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

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

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 Vanilla-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 ever 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.

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 [Mich18]. 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, Edmu99, Paro18] 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, Paro18]. 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, Mich18]. 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 [Paul16] 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 was to:

- Implement a preemption bounding technique [Kath13] 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.
- Explore alternative algorithms that can perform bounded partial order reduction.

1.3 Overview

In Chapter 2 the theoretical background utilized in Nidhugg is given for both unbounded and bounded DPOR is given. In Chapter 3 the implementation of the methods for Nidhugg is discussed. In Chapter 4 the different techniques implemented are evaluated using both synthetic tests and RCU. In Chapter 5 further discussion on possible optimatizations is made. In Chapter 6 we summarize the previous chapters and the conclusions we drew from this thesis, and present some possible extensions to our work.

Chapter 2

Background Knowledge

2.1 Cuncurrent 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, and resource starvation. The scheduler is usually responsible for running a thread. Due to this scheduling indeterminism the programmer can not always be aware of which thread will be scheduled next and thus concurrent programs may seem to run randomly.

An important aspect of a concurrent program is the notion of the set of interleavings. 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 those process.

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 or cannot be traced during debugging. There may be only a limited number of interleavings that produce an error.

2.2 Stateless model checking and partial order reduction

In order to find an error of a concurrent algorithm, one must examine every possible interleaving this algorithm can produce. Usually the error would occur only under some unexpected interleaving, 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: partial order reduction and bounded search. Partial order reduction is aiming to reduce the number of interleavings explored by eliminating equivalent interleavings. 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 affected by the order under which

each operation occurs. There are two ways that a partial order reduction algorithm can be implemented. The first is a static partial order reduction algorithm where the dependencies between two threads are tracked before the execution of the concurrent program. The second is the Dynamic partial order reduction (DPOR) which observes the program's dependencies on runtime. It is important to notice that the size of the state space still grows exponentially even if it is reduced by the DPOR.

For larger programs DPOR often runs longer than 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 [Paul16]. There have been proposed many bounded techniques such as preemption bounded search or delay bounded search. 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.

2.3 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:

The algorithm consists of the following steps:

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. (for every element).

An example execution of the algorithm is in Figure 2.1 where both the source code and an explored trace are given. As we can easily notice for each command 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 2.1: Clock example

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

2.4 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 happen-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 happen-before relations. In practice, the happens-before assignment is implemented with the use of vector clocks that demonstrate the relating accesses to the same variable.

Definition 2.1. (happen-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 one would be 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 could 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.

2.5 Indipendence 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 $E' | = p \diamondsuit w$ to denote that $E \not\models p \diamondsuit w$ does not hold.
For a sequence w and $p \in w$, let w
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 $e_E e'$ 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 0 ' E and e occurs immediately before e' in E' , then $proc(e')$ was not blocked before the occurrence of e.

2.6 Unbounded dynamic partial order reduction

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

Definition 2.2. Sufficient Sets A set of transitions is sufficient in a state s if any relevant state reachable via an enabled transition from s is also reach able 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 all most of these techniques share in common is the following basic structure:

Algorithm 1: General form of DPOR

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

where $final(S)$ represents the state that will be reached when the scheduling S 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.
An obvious property that the sufficient sets must hold is that Sufficient_set$(final(S)) \subseteq enabled(S)$.

Definition 2.3. $enabled(s)$ Given a state s, $enabled(s)$ represents the set of all the threads that can be scheduled immediately after s.

Intuitively $enabled(s)$ represents the threads that are not blocked or have already finished their execution.
In bibliography many types of sets can be found [Gode96]. In this thesis we mainly focus on persistent sets and on source sets.

2.7 Persistent Sets

Definition 2.4. (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 [P97]. A selective search using persistent sets explores a persistent set of transitions from each state s where $enabled(s) \neq$ and prunes enabled transitions that are not persistent in s. In a more formal way:

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 followed: If a $t \in T$ and there is another thread t' that can be executed until a command which is in a race with the t then t' belongs in the persistent set.
Notice that the definition of persistent sets suggests a way to construct them.
Let a concurrent program contain 3 threads. The first thread changes the value of the variable (writer) and and the other just read this variable (readers). Let w.r1.r2 be an interleaving. According to the definition of the persistent sets r1 and r2 are in a race with w, thus, r1 and r2 must also be on the persistent set of the first command of the interleaving. In Figure 2.2 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. At the second example, again both r and q are added. However, there is no conflict between p and r. The reason why the thread r is added is the conflict that will be produced by the q's write operation.

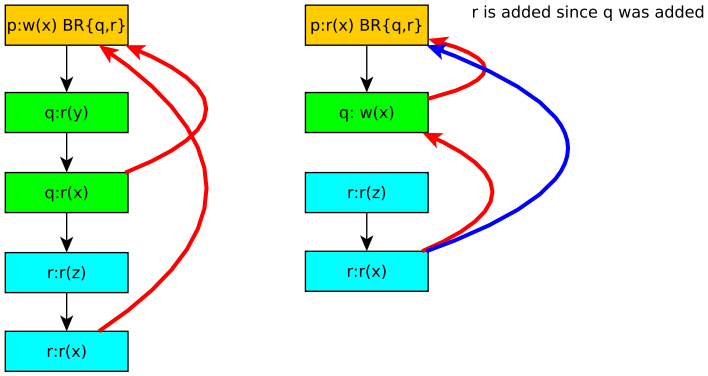

Figure 2.2: Construction of persistent set

2.8 Source sets

Before defining source sets, it is crucial to give some other useful definitions.

Definition 2.5. ($dom(E)$) The set of events-transitions happening in during the scheduling of E.

Definition 2.6. (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 $e \rightarrow_{E.w} next_{[E]}(p)$.

Definition 2.7. (Source Sets) Let S 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. It actually suggests a property for the sufficient sets to hold.

2.9 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 us the concurrent program of one writer and two readers. 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 2.3.

```
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 2.3: 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 2.3: Program with non-minimal persistent sets

As we can see 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 wakeup 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.

2.10 Comparing Persistent sets with Source Sets

It is transparent 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 2.3 an example is given were sleep sets and persistent sets differentiate.
From the example, it is clear 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 p1 does not contain the initials of execution q1.q2.p1.r1.r2, 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 p1, q1 as the source set. Obviously, p1, q1 contains an initial of any execution that starts with either p1 or q1. Any execution sequence which starts with r1 is equivalent to an execution obtained by moving the first step of either p1 or q1 to the beginning:

- If q1 occurs before r2, then q1 is an initial, since it does not conflict with any other transition.
- If q1 occurs after r2, then p1 is independent of all steps, so p1 is an initial. We claim that p1, q1 cannot be a persistent set. The reason is that the execution sequence r1.r2 does not contain any transition in the persistent set, but its second step is dependent with q1. 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.

An interesting question is then whether there are situations where any persistent set contains a strictly smaller source set. We note that the program in Fig. 1 does not illustrate such a situation, since the smallest persistent sets and the smallest source sets coincide: they are either q or r. Nevertheless, the answer to this question is yes, and we formulate this as a theorem.

2.11 Bounded search - preemption bounded search

Bounded search explores only executions that do not exceed a bound [Kath13, Paul16]. The bound may be any property of a sequence of transitions. A bound evaluation function $Bv(S)$ computes the bounded value for a sequence of transitions S. A bound evaluation function Bv 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(S)
3      T = Sufficient_set(final(S)) for all t ∈ T do
4          if Bv(S.t) ≤ c then
5              Explore(S.t)
```

The only difference between the unbounded and the bounded version of the algorithm is the if statement 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, 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 [M07]. The preemption bound is defined recursively as follows.

Definition 2.8. Preemption bound $P_b(t) = 0$

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

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 an other thread. Hence, a preemptive switch will increase the preemption bound.

2.12 Preemption-bound persistent sets

A set that has been proposed as a sufficient for preemption bounded search is the preemption bounded persistent set [Kath13].
An important observation is that the execution of a thread until it gets blocked or terminates will not increase the bound count.

Definition 2.9. (ext(s,t)) Given a state $s = final(S)$ 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. $t.tid \notin enabled(final(S.\beta))$

Definition 2.10. (Preemption bounded persistent set)
A set $T \subseteq \mathcal{T}$ of transitions enabled in a state $s = final(S)$ 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. $Pb(S.t) \leq Pb(S.a_1)$
2. if $Pb(S.t) < Pb(S.a_1)$, then $t \leftrightarrow last(a)$ and $t \leftrightarrow next(final(S.a), last(a).tid)$
3. if $Pb(S.t) = Pb(S.a_1)$, then $ext(s,t) \leftrightarrow last(a)$ and $ext(s,t) \leftrightarrow next(final(S.a), last(a).tid)$

where $A_G(P_b, c)$ is a generic bounded state space with bound function P_b and bound c.
Let us assume that P is a 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 a p was scheduled. These threads are called conservative threads 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 set extends 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

Nidhugg

Nidhugg is a bug-finding tool which targets bugs caused by concurrency and relaxed memory consistency in concurrent programs. It works on the level of LLVM internal representation, which means that it can be used for programs written in languages such as C or C++.
By the time this thesis was written Nidhugg had been supporting the SC, TSO, PSO, POWER and ARM memory models. Target programs should use pthreads for concurrency, and each thread should be deterministic when run in isolation.

3.1 Source-DPOR - The Nidhugg's algorithm

The algorithm that Nidhugg is based on is presented here:

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 ;
```

Explanation of the algorithm: Each step of the algorithm consists of two separate phases. Initially an arbitrary enabled and not sleeping process is chosen and added to the $backtrack(E)$.
During the first step of the algorithm the race detection takes place. The algorithm finds another event e which is already contained in the explored trace and can be reversed with the next step of p. In order to explore the execution sequence where p is before e e.i. a sequence equivalent to the form $E'.u.proc(e).z$ where u is obtained by appending p after the sequence $not(e, E)$ of events that occur after e in $E.p$ but not happen after e and z and continuation of the execution. By appending p at the end of $notdep(e, E)$ we make sure that all the events that happen before p at the first execution still happen before p. 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 calculation of the new sleep set at that step since some processes may have waken 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 threads 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 Implementation of Nidhugg

Nidhugg works on the level of LLVM internal. 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 which is the pid of the thread that was executed. Branches which cause the exploration of different interleaving are also stored in the event object. The scheduling is regulated by the Tracebuilder object which differentiates with 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 3.1. 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. After the execution of an event 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. Any modifications on the algorithm should mainly focus on the procedures taking place during the dependency detection.

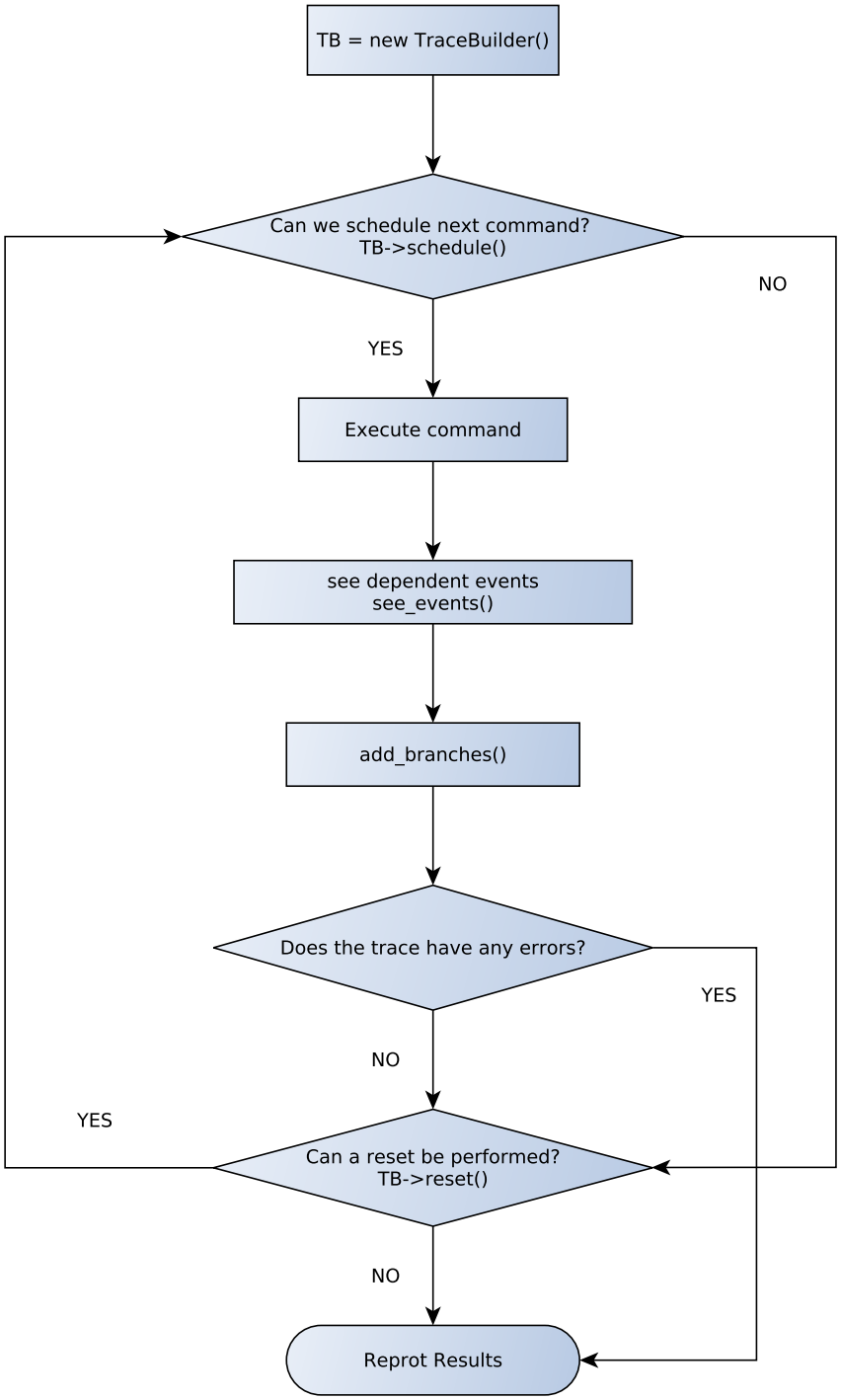

Figure 3.1: Nidhugg's Flow Chart

3.3 Branch addition by 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.
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

Algorithm 4: see_events() routine

```
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) ;
```

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 function 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 happens after the seen_access events.
The add_branch function is the most crucial for the whole Nidhugg infrastructure as it is responsible for the addition of branches.

Algorithm 5: add_branch() routine

```
1  Function add_branch(b)
2      candidates = ∅ ;
3      lc := null ;
4      E′ := E starting from next(b) ;
5      foreach eindom(E′) do
6          if b happens before e or ∃c ∈ candidates | c happens before e then
7              continue ;
8          lc := e.pid;
9          if lc ∈ candidates then
10             continue ;
11         if e.pid ∈ backtrack(b)ore.pid ∈ sleep_setb 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 array 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.
Even though it is not clear from a first look Nidhugg calculates subset of the initials that hold the property of source sets.

3.4 Implementation of the preemption-bound counter

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 trace builder as well, since it is responsible for the scheduling. Such an attribute will be proved 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 an preemptive switch happened. The other occasion when 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. 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 result. The overhead would be caused by both the memory that would be required and by that 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 psuedo code is given in Algorithm 6.

Algorithm 6: 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 Figure 3.1.

```
 === 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 3.1: Example of bound counter

3.5 The Vanilla-BPOR algorithm

3.5.1 Description of Vanilla-BPOR

The first bounded technique to be implemented is the vanilla-bpor. The purpose of the algorithm is to block threads that exceed the bound limit. The algorithm is presented at Algorithm 7.

Algorithm 7: Vanilla-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)\SleepandB_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 ;
```

As it is obvious the algorithm remains the same. The only additions made are related to the thread scheduling. Apparently such an algorithm is far from being sound.
Lets take for example the writer-2 readers example with $b = 0$.

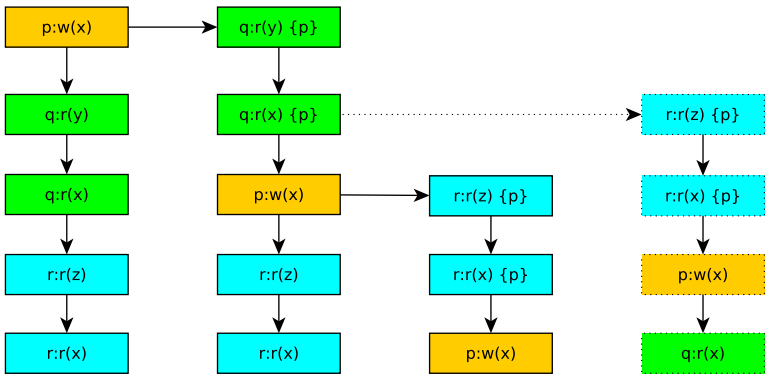

Figure 3.2: Vanilla-BPOR for bound =0

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 vanilla-BPOR was not able to explore them all. r.r.p.q.q was 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 it 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.

3.5.2 Implementation of Vanilla-BPOR

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 3.2.

```
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 3.2: Vanilla-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 was a exceeded a bound blocked trace. 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.
An simple example can be demonstrated here:

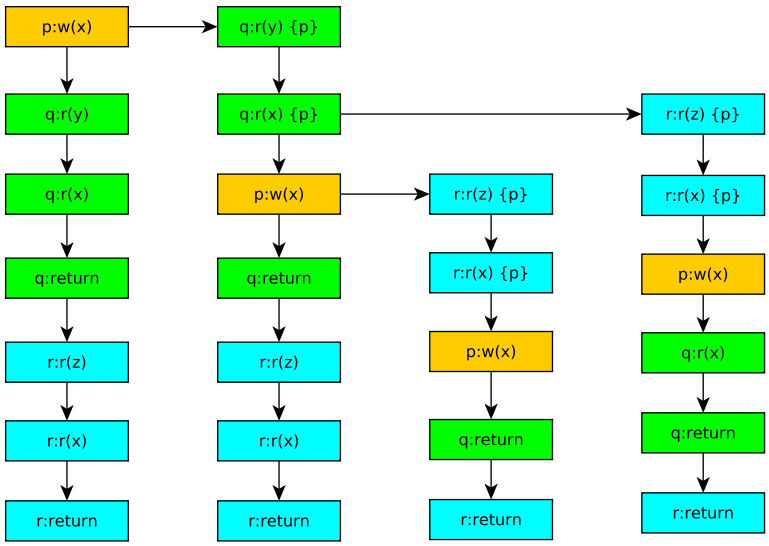

Figure 3.3: Execution without the scheduling optimization

Let us assume only two threads a writer and a thread that reads the same variable twice. As we can infer from the example, had the most recently running thread given the highest priority at least one more interleaving would have been explored.

3.6 The DPOR using persistent sets

The implementation of persistent sets proved to be one of the most challenging tasks in Nidhugg. As it will become clear later, such an implementation of persistent sets is a prerequisite for any other bpor implementation aiming to be sound. Many options had been considered. The definition of persistent sets implies one way to implement them i.e. for every execution step we should check all the other threads and add them to the branch if they contain a command which conflicts with the execution step. This kind of approach completely contradicts with the whole philosophy of Nidhugg which stems from the nature of source sets and, thus, would not be feasible. The option that was finally chosen was an implementation based on the DPOR using Vector Clocks.

3.6.1 Description of DPOR using persistent sets

Since Nidhugg uses vector clocks to track events the DPOR using Clock Vectors variation will be used.

Algorithm 8: DPOR using Clock Vectors

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

A subtle variation of the algorithm is used. As a result there is no need to add all available threads when p is not enabled. If no sufficient candidate was found a candidate suggested by the Source-DPOR algorithm will be added. This way of calculating persistent sets is considered to be more complex and it is usually rejected. However, source sets algorithm is closer to this approach.
It is clear that persistent algorithm implemented differentiates from Source-DPOR in the calculation of the initials. Specifically a subset of the initials that happen before p is used. Intuitively in the case of a writer and 2 readers both readers will be added to the branch since the first read does not happen before the second read. To generalize this idea: since Nidhugg does not enable us to create branches for $last(E)$ when it is scheduled we add the branches later as in DPOR. When a race is considered usually only the thread that causes the race will be added since CI contains this thread only.
We will prove that Nidhugg's DPOR calculates a persistent set or that when the algorithm finishes a persistent set will have been calculated in each step.
Let us assume two processes that are in race with $last(S)$.

- Case 1: at least one of them is a write process. We know that the Nidhugg's DPOR should calculate a superset of the Source DPOR branches, thus, we know that the read and the write processes at some point will be inverted. Moreover we will the CI set will consider both ignoring padding (see Figure 3.4.As a result both processes will be considered and will be added to the persistent set.

Algorithm 9: Nidhugg BPOR

```
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 ;
```

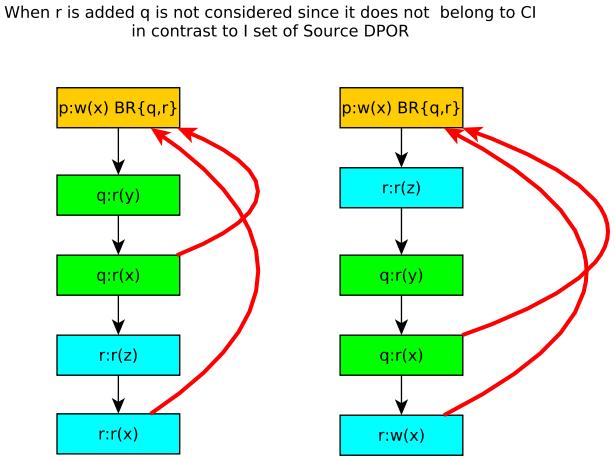

Figure 3.4: Construction of persistent sets in Nidhugg

- 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 $backtrack$.

It is clear that any process no process that does not belong to the backtrack(S) has race with a process that belongs to backtrack(S).

3.6.2 Implementation of DPOR using persistent-sets

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 the $i \rightarrow p$ holds true.

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 10.

Algorithm 10: add_branch() routine 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.

3.7 The BPOR

Having implemented persistent sets correctly the next task is the implementation of 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.

Definition 3.1. (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.

3.7.1 Description of BPOR

The algorithm implemented is presented here [Kath13] in detail. A modification of this algorithm is used in order to take advantage of the Nidhugg's infrastructure. The algorithm is presented at Algorithm 11.
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.

Algorithm 11: 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 ;
```

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:

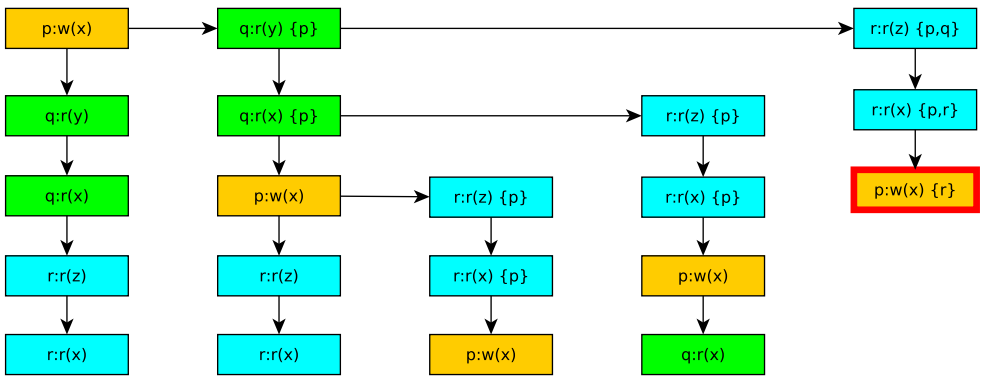

Figure 3.5: Usage of non-conservative branches

We notice that the last trace is sleep set blocked while it should be examined. The algorithm is unaware that the thread 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 where added at this certain point of the trace so no thread is added twice. 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 Source-DPOR with the addition of the conservative branches. The solution is based on the notion of the conservative sets where every thread that was added to the branch is recorded. However many challenges araise which are discussed in the implementation section.

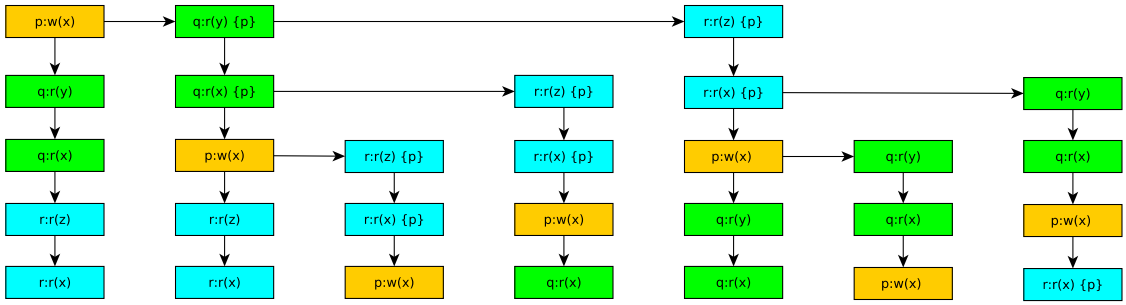

Figure 3.6: BPOR

3.7.2 Implementation of 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 demonstrated at Algorithm 12.
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 12: see_events() routine for BPOR

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

In case the add_branch() invokes directly then the add_conservative_branch() is called which works as the "conservative" part of the see_events(). The pseudo code for this procedure is given here:
During add_branch procedure we add branches at the appropriate places using the algorithm for the persistent sets suggested in the previous section. It was made clear that 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.

3.8 The Source-BPOR algorithm

Having implemented a BPOR algorithm 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-2readers 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 3.7.

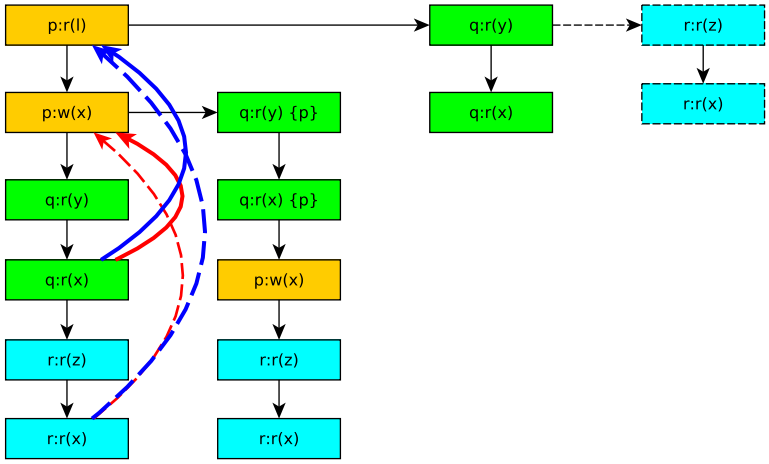

Figure 3.7: Following source sets for conservative branches

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 the following:

3.8.1 Description of Source-BPOR

We can notice that the algorithm is equivalent to the source-DPOR for non-conservative traces and equivalent to BPOR for conservative traces.

3.8.2 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 given at Algorithm 13.

Algorithm 13: 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 orlc → 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
```

3.9 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.

Chapter 4

Evaluation of Bounding Techniques

In this chapter the performance of each implemented technique will be discussed. Firstly the performance of persistent sets is demonstrated in order to prove that is indeed differentiated from source sets. The evaluation happens in two parts. In the first part, short synthetic programs are used, while in the second part real world software is tested. All the programs tested will be Added in the appendix section. One area where Nidhugg is tested is the verification of the Read Copy Update technique of the Linux kernel.

4.1 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 optimization of the Source-DPOR compared to the DPOR.

- The writer-Nreaders test: In this test N threads read (reader) the same global variable and one threads (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 were 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: program whose pseudo code is shown in Figure 4. Its N+1 threads 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 table at the same order and as a result many conflicts emerge.

4.2 RCU

Read-Copy-Update is a synchronization mechanism invented by McKenney and Slingwine [McKe98] that is a part of the Linux kernel since 2002. The key feature of RCU is the good scalability it provides by allowing concurrent reads and updates. While this may seem counter-intuitive or impossible at first, RCU allows this in a very simple yet extremely efficient way: by maintaining multiple data versions. RCU is carefully orchestrated in a way that not only ensures that reads are coherent and no data will be deleted until it is certain that no one holds references to them, but also uses efficient and scalable mechanisms which make read paths extremely fast. Most notably, in non-preemptible kernels, RCU imposes zero overhead to readers.
The basic idea behind RCU is to split updates in two phases: the removal phase and the reclamation phase. During the removal phase, an updater removes references to data either by destroying them (i.e., setting them to NULL), or by replacing them with references to newer versions of these data. This phase can run concurrently with reads due to the fact that modern microprocessors guarantee that a reader will see either the old or the new reference to an object, and not a weird mash-up of these two or a partially updated reference. During the reclamation phase, the updater frees the items removed in the removal phase, i.e., these items are reclaimed. Of course, since RCU allows concurrent reads and updates, the reclamation phase must begin after the removal phase and, more specifically, when it is certain that there are no readers accessing or holding references to the data being reclaimed.
The typical update procedure using RCU looks as follows [McKe98].

1. Ensure that all readers accessing RCU-protected data structures carry out their references from within an RCU read-side critical section.

2. Remove pointers to a data structure, so that subsequent readers cannot gain a reference to it (removal phase).

3. Wait until all pre-existing readers complete their RCU read-side critical section, so that there no one holding a reference to the item being removed.

4. At this point, there cannot be any readers still holding references to the data structure, which may now be safely freed.

- -DASSERT_0 : An `assert(0)` statement is inserted after `synchronize_rcu()`. Obviously, this results in a test failure. What this assertion does, however, is that it shows that the grace period can end, and that there are some explored executions in which it does; i.e., it provides liveness guarantees. We will use this injection in conjunction with some of the next bug injections in order to determine whether the grace period can end or not.

- -DFORCE_FAILURE_1 : This injection forces the reader to pass through and report a quiescent state during its read-side critical section. Of course, this is not permitted and, as expected, results in a failure.

- -DFORCE_FAILURE_2 : A return statement is placed at the beginning of `synchronize_rcu()`. Of course, this results in a test failure since the updater does not wait for pre-existing readers to complete their RCU read-side critical sections, and such critical sections are not permitted to span a grace period.

- -DFORCE_FAILURE_3 : This injection makes `rcu_gp_init()` clear the node mask `(->qsmask)` variables instead of setting them appropriately. The `rcu_gp_init()`

function is invoked from the RCU grace-period kthread at the beginning of each grace period in order to initialize it. Obviously, since the `->qsmask` variables are cleared from the start of the grace period, the grace period can end immediately. In other words, the grace-period kthread does not wait for pre-existing readers to complete. (This can be considered a more complex variant of injection #2.) As expected, this injection results in a test failure.

- -DFORCE_FAILURE_4 : In this injection the `rcu_gp_fqs()` function is made to clear the `->qsmask` variables instead of waiting for the CPUs to clear their respective bits. Of course, in order for `rcu_gp_fqs()` to clear the `->qsmask` variables, the respective CPUs (in our case, the reader) have to be in dynticks-idle mode (or the CPU must have passed through a quiescent state at some point, since the respective dynticks counters are sampled). Consequently, in our code, CPU0 calls the `rcu_gp_fqs()` function, and CPU1 enters and exits dynticks-idle mode within its RCU read-side critical section, which enables CPU0 to prematurely end the grace period. This can be considered an even more complex variant of injection #2, and results in a test failure, as expected.

- -DFORCE_FAILURE_5 : This injection makes the function `__note_gp_changes()` clear the bit of the respective node's mask for this CPU (`rnp->qsmask &= ~rdp->grpmask`). This function is called when a CPU enters RCU core in order to record the beginnings and ends of grace periods. However, instead of just recording a grace period beginning, `__note_gp_changes()` is now made to also clear the `->qsmask` bit, which implies that this CPU reported a quiescent state for the new grace period. This results in test failure.

- -DFORCE_FAILURE_6 : Essentially, what this injection does is delete the if statement checking whether a node's mask is zero and calling `rcu_preempt_blocked_readers_cgp()`, in the `rcu_report_qs_rnp()` function. This if statement just checks whether the bitmask for this node is cleared in order for a node to acquire its parent's lock. In a real kernel, this should result in too short grace periods, since a signal that will prematurely awake the grace-period kthread is sent, if there are multiple CPUs. In our case, however, it does not lead to too-short grace periods since, in our modeling, `wake_up()` boils down to a no-op – there is no need to wake up someone who is just spinning. However, if we were dealing with a two-level tree, the caller of `rcu_report_qs_rnp()` would move up one level and trigger a `WARN_ON_ONCE()` statement that checks whether the child node's bits are cleared. Hence, this test automatically sets the number of CPUs to `CONFIG_RCU_FANOUT_LEAF + 1` (i.e., to 17, since the default value of `CONFIG_RCU_FANOUT_LEAF` is 16 in these kernels). Also, this test requires the use of a higher unroll value because there are some loops that need to be unrolled at least as many times as the number of CPUs used plus one. So, we used an unroll value of 19 for this case.

- -DLIVENESS_CHECK_1 : This eliminates the need for a CPU to pass through a quiescent state by setting rdp->qs_pending to zero in __note_gp_changes() . This function updates the per-CPU rcu_data structure and, since rdp->qs_pending is set to zero, there is no need for a CPU to report a quiescent state to RCU, which prevents grace periods from completing. When the injection is used in conjunction with -DASSERT_0 , no execution triggers the assert(0) statement after synchronize_rcu() .

- -DLIVENESS_CHECK_2 : A return statement is placed at the beginning of the rcu_sched_qs() function. In effect, this means that CPUs cannot record their passing

through a quiescent state in the respective rc_data structures, something that also prevents grace periods from completing. Used in conjunction with -DASSERT_0 this bug injection also results in no executions triggering the assertion, thus signifying a liveness violation.

- -DLIVENESS_CHECK_3 : A return statement is placed at the beginning of rcu_report_qs_rnp(). This means that CPUs cannot report their passing through a quiescent state to RCU, which in turn means that grace periods cannot complete. This injection also needs to be used together with -DASSERT_0 to discover the liveness violation.

4.3 Evaluation of Persistent sets

As it was established in the previous chapter the implementation of the persistent sets is crucial since they are utilized in every bounding technique. As a result it is vital to able to verify that source-DPOR is indeed an optimization over the DPOR. The comparison of the results is demonstrated below.

4.3.1 Evaluation of Persistent sets on Synthetic tests

It is clear from these testcases that there are indeed some different results. As it was expected source-DPOR explores less traces than the 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 implication of the Source-DPOR is not the same for all test cases. It varies due to the different approaches as well as the size of the state space. The results are presented with two different ways. The writer-Nreaders example is given with a graph 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 a table so their can be an easily comparison.

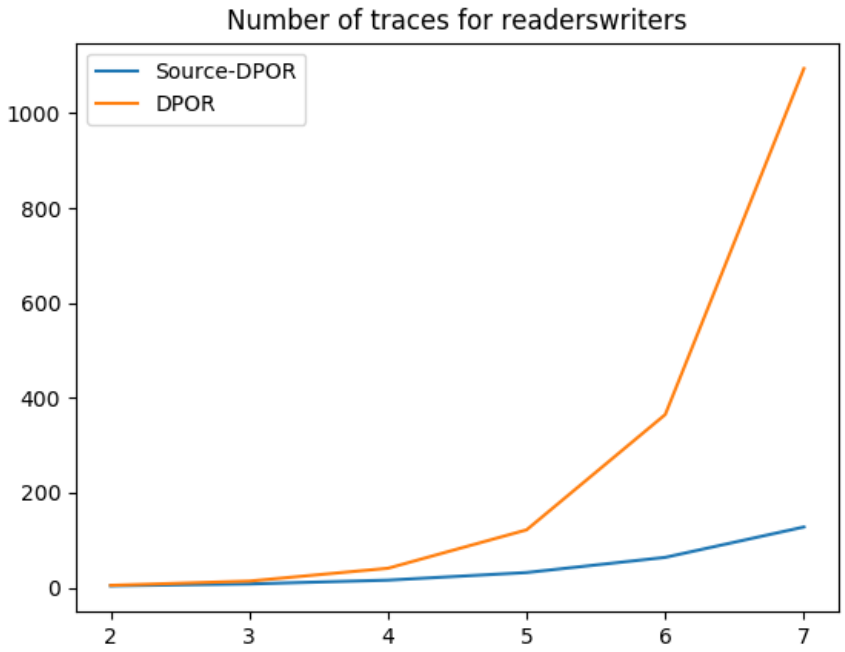

Figure 4.1: writer-N-readers

Test case	Source-DPOR	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 4.1: Source-DPOR vs DPOR for synthetic tests

4.3.2 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 [Koko17]. The reason why the results of DPOR and Source-DPOR may be due to the operations that take place which not allow for the optimization of the Source-DPOR to be effective. There is another reason based on the LLVM code the Nidhugg produces which is explained in the next section.

4.4 Comparison with Concuerror results - Why DPOR may be enough for LLVM

There are many cases where Concuerror results do not seem to be verified. We have found out that both DPOR and Source-DPOR produce the same results. However, taking a look on the code that Nidhugg tests this behavior seems rational. LLVM produces much more code than the code in the source file. Due to this padding of the dependent events, the number of conflicting events is less than the number of conflicting events in an Erlang program.
At Figure 4.2 the reason is visualized. As shown in the figure we would expect that both p and r would be added in the persistent set. However when the p thread is added the write event is no longer visible, according to the persistent set definition. As a the r thread is not added. We can compare the LLVM code with the relative Erlang code.

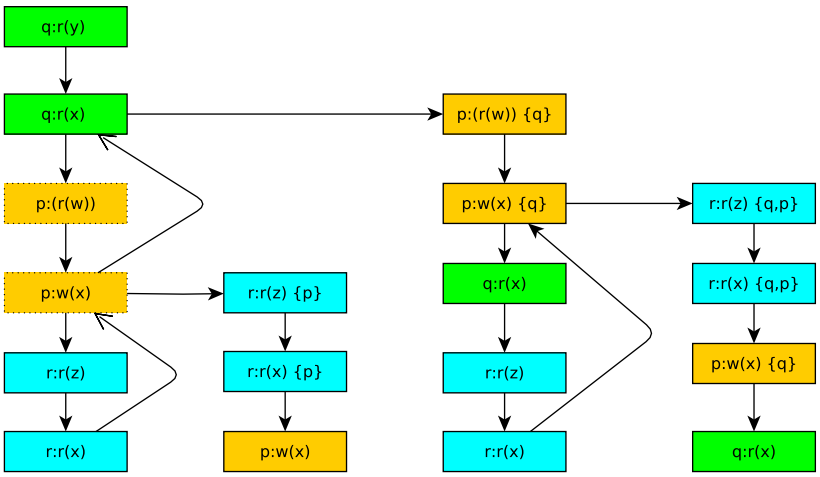

Figure 4.2: Padding impact on persistent sets

```
readers_rwr() ->
    ets:new(table, [public, named_table]),
    ets:insert(table, {x, 0}),
    ets:insert(table, {y, 0}),
    Writer =
        fun() ->
                ets:insert(table, {x, 1})
        end,
    Reader =
        fun() ->
                ets:lookup(table, y),
                ets:lookup(table, x)
        end,
    spawn(Reader),
    spawn(Writer),
    spawn(Reader),
    receive
    after
        infinity -> ok
    end.
```

Listing 4.1: Erlang code for rwr

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

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

```
@c = global i32 0, align 4

; Function Attrs: nounwind uwtable
define i8* @writer(i8* %arg) #0 {
  %1 = alloca i8*, align 8
  store i8* %arg, i8** %1, align 8
  store volatile i32 2, i32* @c, align 4
  ret i8* null
}

; Function Attrs: nounwind uwtable
define i8* @reader(i8* %arg) #0 {
  %1 = alloca i8*, align 8
  %local = alloca i32, align 4
  store i8* %arg, i8** %1, align 8
  %2 = load volatile i32, i32* @c, align 4
  store i32 %2, i32* %local, align 4
  ret i8* null
}
```

Figure 4.3: Comparison between C and LLVM

4.5 Evaluation of Bounding Techniques

The evaluation of the 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. It is intelligible that a faster algorithm may compromise the soundness of the state space.

Technique:	Vanilla-BPOR			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	10610
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 4.2: Traces for various bound limits

4.5.1 Evaluation of Bounding Techniques on Synthetic tests

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

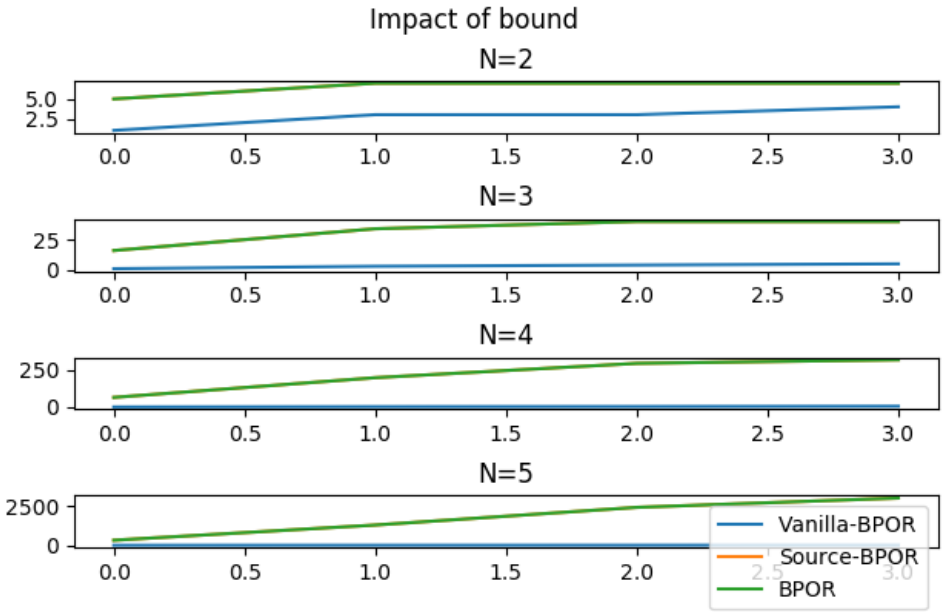

Figure 4.4: writer-N-readers bounded

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

4.5.2 Evalution of Bounding Techniques on RCU

The results are demonstrated below. Notice that since the Source-DPOR did not resulted less traces than the DPOR we could not expect from the Source-BPOR and BPOR to

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 4.3: RCU results without bound

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	1	0.19	NF	2	0.19	NF	1	0.3	NF	2	0.31	NF	1	0.44	NF	2	0.48	NF	1	0.57	NF	2	0.59	NF	1	0.59	NF	2	0.62	NF
-DASSERT_0	1	0.18	NF	2	0.19	NF	1	0.29	NF	2	0.31	NF	1	0.44	NF	2	0.49	NF	1	0.56	NF	2	0.59	NF	1	0.59	NF	2	0.61	NF
-DFORCE_FAILURE_1	1	0.18	NF	2	0.19	NF	1	0.29	NF	2	0.32	NF	1	0.46	NF	2	0.48	NF	1	0.56	NF	2	0.59	NF	1	0.59	NF	2	0.6	NF
-DFORCE_FAILURE_2	1	0.18	NF	2	0.19	NF	1	0.29	NF	2	0.31	NF	1	0.44	NF	2	0.47	NF	1	0.56	NF	2	0.57	NF	1	0.56	NF	2	0.6	NF
-DFORCE_FAILURE_3	1	0.18	NF	2	0.2	NF	1	0.3	NF	2	0.34	NF	1	0.45	NF	2	0.5	NF	1	0.56	NF	2	0.61	NF	1	0.58	NF	2	0.63	NF
-DFORCE_FAILURE_4	1	0.18	NF	2	0.19	NF	1	0.29	NF	2	0.33	NF	1	0.45	NF	2	0.52	NF	1	0.56	NF	2	0.63	NF	1	0.57	NF	2	0.63	NF
-DFORCE_FAILURE_5	1	0.18	NF	2	0.19	NF	1	0.3	NF	2	0.31	NF	1	0.44	NF	2	0.48	NF	1	0.56	NF	2	0.59	NF	1	0.57	NF	2	0.61	NF
-DFORCE_FAILURE_6	1	0.94	F	1	0.95	F	1	1.36	NF	2	2.74	F	1	2.85	NF	2	4.47	F	1	6.23	NF	2	8.7	F	1	6.58	NF	2	8.73	F
-DLIVENESS_CHECK_1	1	0.17	NF	2	0.19	NF	1	0.29	NF	2	0.31	NF	1	0.44	NF	2	0.47	NF	1	0.55	NF	2	0.6	NF	1	0.58	NF	2	0.61	NF
-DLIVENESS_CHECK_2	1	0.17	NF	2	0.2	NF	1	0.29	NF	2	0.32	NF	1	0.44	NF	2	0.48	NF	1	0.57	NF	2	0.59	NF	1	0.57	NF	2	0.62	NF
-DLIVENESS_CHECK_3	1	0.18	NF	2	0.19	NF	1	0.29	NF	2	0.31	NF	1	0.44	NF	2	0.48	NF	1	0.56	NF	2	0.58	NF	1	0.57	NF	2	0.6	NF

Table 4.4: RCU results for bound $b = 0$

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	3	0.19	NF	44	0.67	NF	2	0.3	NF	28	0.73	NF	2	0.46	NF	24	0.98	NF	2	0.58	NF	24	1.12	NF	2	0.58	NF	24	1.16	NF
-DASSERT_0	3	0.19	NF	44	0.67	NF	2	0.3	NF	28	0.73	NF	2	0.46	NF	24	0.98	NF	2	0.57	NF	24	1.11	NF	2	0.58	NF	24	1.18	NF
-DFORCE_FAILURE_1	3	0.19	NF	44	0.68	NF	2	0.31	NF	28	0.72	NF	2	0.47	NF	24	0.97	NF	2	0.57	NF	24	1.12	NF	2	0.58	NF	24	1.15	NF
-DFORCE_FAILURE_2	1	0.18	NF	6	0.35	F	3	0.31	NF	5	0.54	F	3	0.47	NF	5	0.75	F	3	0.58	NF	5	0.91	F	3	0.6	NF	5	0.95	F
-DFORCE_FAILURE_3	3	0.19	NF	44	0.66	NF	2	0.3	NF	33	1.01	NF	2	0.46	NF	41	1.75	NF	2	0.57	NF	41	1.9	NF	2	0.59	NF	41	1.98	NF
-DFORCE_FAILURE_4	4	0.21	NF	55	0.8	NF	2	0.3	NF	43	1.1	NF	2	0.46	NF	25	1.11	NF	2	0.58	NF	25	1.28	NF	2	0.58	NF	25	1.33	NF
-DFORCE_FAILURE_5	3	0.2	NF	44	0.67	NF	2	0.31	NF	18	0.53	NF	2	0.45	NF	16	0.75	NF	2	0.57	NF	16	0.86	NF	2	0.58	NF	16	0.91	NF
-DFORCE_FAILURE_6	1	0.94	F	1	0.94	F	2	1.49	NF	2	2.75	F	2	2.97	NF	2	4.41	F	2	6.36	NF	2	8.22	F	2	6.73	NF	2	8.55	F
-DLIVENESS_CHECK_1	3	0.19	NF	44	0.66	NF	2	0.3	NF	28	0.7	NF	2	0.46	NF	24	0.96	NF	2	0.57	NF	24	1.09	NF	2	0.58	NF	24	1.13	NF
-DLIVENESS_CHECK_2	3	0.19	NF	52	0.79	NF	2	0.3	NF	28	0.69	NF	2	0.45	NF	24	0.96	NF	2	0.57	NF	24	1.11	NF	2	0.58	NF	24	1.16	NF
-DLIVENESS_CHECK_3	3	0.19	NF	44	0.66	NF	2	0.3	NF	28	0.71	NF	2	0.45	NF	24	0.97	NF	2	0.57	NF	24	1.11	NF	2	0.57	NF	24	1.14	NF

Table 4.5: RCU results for bound $b = 1$

differentiate. Moreover tests did not show any difference. For these reasons only the performance of Vanilla-BPOR and 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 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

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	9	0.27	NF	353	4.68	NF	5	0.35	NF	153	3.44	NF	5	0.52	NF	153	5.39	NF	5	0.64	NF	153	5.74	NF	5	0.66	NF	153	6.01	NF
-DASSERT_0	9	0.28	NF	353	4.68	NF	5	0.36	NF	153	3.47	NF	5	0.52	NF	153	5.44	NF	5	0.64	NF	153	5.73	NF	5	0.66	NF	153	5.98	NF
-DFORCE_FAILURE_1	9	0.28	NF	353	4.67	NF	5	0.35	NF	153	3.47	NF	5	0.54	NF	153	5.45	NF	5	0.65	NF	153	5.71	NF	5	0.66	NF	153	5.94	NF
-DFORCE_FAILURE_2	4	0.32	F	7	0.36	F	3	0.33	NF	5	0.55	F	3	0.49	NF	5	0.77	F	3	0.6	NF	5	0.91	F	3	0.62	NF	5	0.94	F
-DFORCE_FAILURE_3	9	0.28	NF	188	2.45	NF	5	0.37	NF	201	6.49	F	5	0.55	NF	258	12.11	F	5	0.67	NF	258	12.59	F	5	0.68	NF	258	12.84	F
-DFORCE_FAILURE_4	20	0.45	NF	47	0.91	F	6	0.38	NF	41	1.78	F	6	0.56	NF	21	1.89	F	6	0.67	NF	24	2.3	F	6	0.7	NF	24	2.39	F
-DFORCE_FAILURE_5	9	0.28	NF	306	3.88	NF	5	0.36	NF	105	2.29	NF	5	0.51	NF	90	2.96	NF	5	0.62	NF	90	3.21	NF	5	0.66	NF	90	3.38	NF
-DFORCE_FAILURE_6	1	0.93	F	1	0.93	F	2	2.74	F	2	2.79	F	2	4.31	F	2	4.32	F	2	8.24	F	2	8.2	F	2	8.59	F	2	8.6	F
-DLIVENESS_CHECK_1	9	0.27	NF	182	2.36	NF	5	0.35	NF	94	1.82	NF	5	0.53	NF	79	2.34	NF	5	0.64	NF	79	2.53	NF	5	0.65	NF	79	2.66	NF
-DLIVENESS_CHECK_2	10	0.3	NF	216	2.86	NF	5	0.35	NF	94	1.83	NF	5	0.52	NF	97	2.94	NF	5	0.64	NF	97	3.21	NF	5	0.66	NF	97	3.28	NF
-DLIVENESS_CHECK_3	9	0.27	NF	201	2.55	NF	5	0.35	NF	105	2.08	NF	5	0.53	NF	79	2.36	NF	5	0.63	NF	88	2.83	NF	5	0.66	NF	88	2.97	NF

Table 4.6: RCU results for bound $b = 2$

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	17	0.48	NF	1627	22.22	NF	8	0.41	NF	603	14.51	NF	8	0.58	NF	659	24.59	NF	8	0.7	NF	609	23.21	NF	8	0.74	NF	659	26.53	NF
-DASSERT_0	17	0.47	NF	183	2.65	F	8	0.4	NF	106	2.96	F	8	0.59	NF	128	5.39	F	8	0.71	NF	118	5.28	F	8	0.73	NF	128	5.91	F
-DFORCE_FAILURE_1	17	0.47	NF	1627	21.99	NF	8	0.4	NF	603	14.41	NF	8	0.59	NF	659	25.71	NF	8	0.7	NF	609	23.03	NF	8	0.73	NF	659	26.31	NF
-DFORCE_FAILURE_2	4	0.34	F	7	0.36	F	3	0.54	F	5	0.55	F	3	0.78	F	5	0.82	F	3	0.93	F	5	0.91	F	3	0.96	F	5	0.92	F
-DFORCE_FAILURE_3	17	0.47	NF	634	8.0	NF	8	0.53	NF	1091	34.12	F	8	0.78	NF	1481	66.72	F	8	0.91	NF	1481	68.81	F	8	0.96	NF	1481	71.3	F
-DFORCE_FAILURE_4	43	1.22	NF	87	1.42	F	10	0.6	NF	70	2.75	F	10	0.84	NF	25	2.06	F	10	1.04	NF	32	2.69	F	10	1.07	NF	32	2.78	F
-DFORCE_FAILURE_5	17	0.46	NF	1157	14.6	NF	8	0.37	NF	386	8.8	NF	8	0.56	NF	324	10.61	NF	8	0.68	NF	324	11.3	NF	8	0.71	NF	324	11.78	NF
-DFORCE_FAILURE_6	1	0.94	F	1	0.94	F	2	2.79	F	2	2.8	F	2	4.39	F	2	4.25	F	2	8.07	F	2	8.02	F	2	8.67	F	2	8.58	F
-DLIVENESS_CHECK_1	17	0.46	NF	597	7.57	NF	8	0.39	NF	251	4.78	NF	8	0.58	NF	198	5.49	NF	8	0.7	NF	198	5.81	NF	8	0.72	NF	198	6.12	NF
-DLIVENESS_CHECK_2	20	0.59	NF	767	9.8	NF	8	0.4	NF	251	4.74	NF	8	0.58	NF	258	7.39	NF	8	0.71	NF	258	7.78	NF	8	0.74	NF	258	8.2	NF
-DLIVENESS_CHECK_3	17	0.46	NF	665	8.2	NF	8	0.4	NF	292	5.7	NF	8	0.6	NF	198	5.57	NF	8	0.7	NF	232	6.86	NF	8	0.74	NF	232	7.22	NF

Table 4.7: RCU results for bound $b = 3$

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR			VAN			BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	50	0.99	NF	5634	78.41	NF	10	0.46	NF	2083	54.54	NF	10	0.72	NF	2469	100.03	NF	10	0.89	NF	1979	84.04	NF	10	1.02	NF	2469	110.91	NF
-DASSERT_0	50	0.97	NF	274	3.79	F	10	0.47	NF	179	4.93	F	10	0.79	NF	118	4.77	F	10	0.82	NF	108	4.87	F	10	1.12	NF	118	5.43	F
-DFORCE_FAILURE_1	50	0.96	NF	275	3.74	F	10	0.52	NF	182	5.02	F	10	0.75	NF	300	12.69	F	10	0.85	NF	220	9.73	F	10	1.04	NF	300	13.93	F
-DFORCE_FAILURE_2	4	0.32	F	7	0.35	F	3	0.54	F	5	0.53	F	3	0.82	F	5	0.72	F	3	0.97	F	5	0.89	F	3	1.05	F	5	0.92	F
-DFORCE_FAILURE_3	50	0.95	NF	1627	20.57	NF	15	0.68	NF	100000	0.0	NF	15	0.98	NF	100000	0.0	NF	15	1.14	NF	100000	0.0	NF	15	1.16	NF	100000	0.0	NF
-DFORCE_FAILURE_4	140	2.92	NF	127	1.94	F	29	1.0	NF	112	4.08	F	20	1.18	NF	26	2.1	F	21	1.45	NF	40	3.1	F	21	1.58	NF	40	3.21	F
-DFORCE_FAILURE_5	49	0.96	NF	4155	53.32	NF	9	0.43	NF	60	2.26	F	9	0.63	NF	60	3.12	F	9	0.77	NF	60	3.47	F	9	0.92	NF	60	3.61	F
-DFORCE_FAILURE_6	1	0.92	F	1	0.91	F	2	2.77	F	2	2.92	F	2	4.51	F	2	4.19	F	2	9.42	F	2	8.22	F	2	10.12	F	2	8.75	F
-DLIVENESS_CHECK_1	48	0.93	NF	1493	19.1	NF	10	0.53	NF	517	10.37	NF	10	0.68	NF	404	10.86	NF	10	0.92	NF	404	11.6	NF	10	0.94	NF	404	12.26	NF
-DLIVENESS_CHECK_2	61	1.15	NF	2105	28.37	NF	10	0.6	NF	517	9.38	NF	10	0.71	NF	582	16.25	NF	10	0.89	NF	582	17.34	NF	10	1.02	NF	582	18.14	NF
-DLIVENESS_CHECK_3	49	0.95	NF	1788	22.26	NF	10	0.47	NF	655	12.44	NF	10	0.74	NF	404	11.04	NF	10	0.91	NF	506	14.83	NF	10	0.94	NF	506	15.64	NF

Table 4.8: RCU results for bound $b = 4$

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	DPOR			BPOR			DPOR			BPOR			DPOR			BPOR			DPOR			BPOR			DPOR			BPOR		
	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	183	2.65	3	37	1.36	F	106	2.96	3	29	1.77	F	128	5.39	3	29	1.97	F	118	5.28	3	29	2.05	F	128	5.91	3
-DFORCE_FAILURE_1	146	2.19	F	275	3.57	4	41	1.48	F	182	5.02	4	33	1.94	F	300	12.69	4	33	2.16	F	220	9.73	4	33	2.23	F	300	13.93	4
-DFORCE_FAILURE_2	4	0.32	F	6	0.35	1	3	0.53	F	5	0.54	1	3	0.74	F	5	0.75	1	3	0.9	F	5	0.91	1	3	0.92	F	5	0.95	1
-DFORCE_FAILURE_3	2372	30.82	NF	-	-	-	13264	464.77	F	201	6.49	2	8114	408.74	F	258	12.11	2	8114	423.19	F	258	12.59	2	8114	440.16	F	258	12.84	2
-DFORCE_FAILURE_4	84	1.39	F	47	0.91	2	79	3.15	F	41	1.78	2	24	1.99	F	21	1.89	2	43	3.32	F	24	2.3	2	43	3.44	F	24	2.39	2
-DFORCE_FAILURE_5	4888	64.83	NF	-	-	-	9	0.85	F	60	2.26	4	9	1.21	F	60	3.12	4	9	1.43	F	60	3.47	4	9	1.46	F	60	3.61	4
-DFORCE_FAILURE_6	1	0.94	F	1	0.95	0	2	2.7	F	2	2.74	0	2	4.21	F	2	4.47	0	2	8.03	F	2	8.7	0	2	8.53	F	2	8.73	0

Table 4.9: Comparison between DPOR and BPOR

found faster with source-DPOR.

4.5.3 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 not fair. Looking at table 4.10 we can clearly see that the minimum traces required for BPOR to track the bug for the first time are always less than DPOR

4.6 Equivalence between BPOR and Source-BPOR (Correctness of Source-BPOR)

Surprisingly the results of 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 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.

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	DPOR			BPOR			DPOR			BPOR			DPOR			BPOR			DPOR			BPOR			DPOR			BPOR		
	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	183	2.65	3	512	17.67	F	106	2.96	3	858	37.31	F	128	5.39	3	338	15.94	F	118	5.28	3	858	40.42	F	128	5.91	3
-DFORCE_FAILURE_1	247	3.55	F	275	3.74	4	515	18.21	F	182	5.02	4	861	37.8	F	300	12.69	4	341	15.9	F	220	9.73	4	861	40.52	F	300	13.93	4
-DFORCE_FAILURE_2	4	0.34	F	6	0.35	1	3	0.55	F	5	0.54	1	3	0.7	F	5	0.75	1	3	0.86	F	5	0.91	1	3	0.88	F	5	0.95	1
-DFORCE_FAILURE_3	2372	32.1	NF	-	-	-	17094	636.25	F	201	6.49	2	15349	736.84	F	258	12.11	2	15349	714.01	F	258	12.59	2	15349	793.75	F	258	12.84	2
-DFORCE_FAILURE_4	78	1.43	F	47	0.91	2	61	2.74	F	41	1.78	2	16	1.67	F	21	1.89	2	27	2.48	F	24	2.3	2	27	2.6	F	24	2.39	2
-DFORCE_FAILURE_5	17426	185.57	NF	-	-	-	118	4.1	F	60	2.26	4	112	5.12	F	60	3.12	4	112	5.51	F	60	3.47	4	112	5.8	F	60	3.61	4
-DFORCE_FAILURE_6	1	0.98	F	1	0.95	0	2	2.93	F	2	2.74	0	2	4.21	F	2	4.47	0	2	8.13	F	2	8.7	0	2	8.62	F	2	8.73	0

Table 4.10: Comparison between DPOR and BPOR with the bug

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 is an execution step which conflicts with b. Hence, there must be a trace where b happens before s. As a result b belongs to the source set. As shown in the Figure 4.5, the branch which seems to be initially rejected, will finally be added by the Source-DPOR and as a result belongs to the source-set.

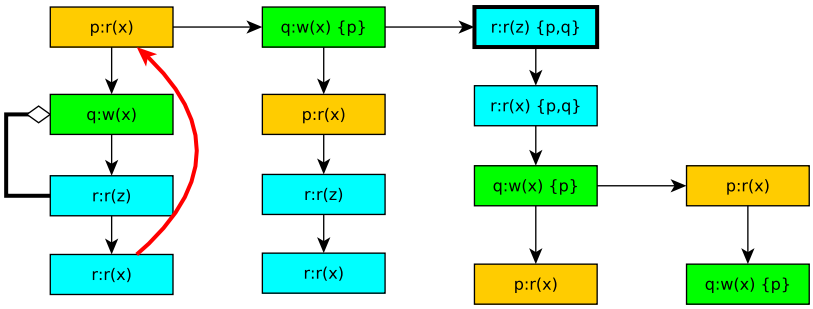

Figure 4.5: Source-BPOR and BPOR equivalence Case 1

- Case 2: s doesn't conflict with b (both b and s are read operations). There must b an 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 4.6, both q and r belong to the persistent set. However, the r thread will be rejected since it shares the same initials with the q 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.

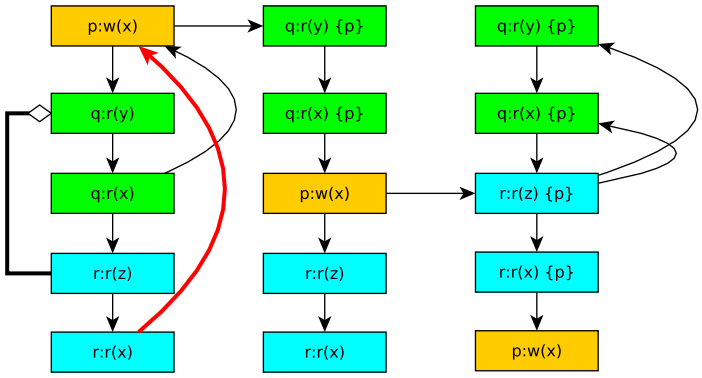

Figure 4.6: Source-BPOR and BPOR equivalence Case 2

A more intuitive explanation of the equivalence of the two techniques would be this: Persistent sets add threads in points higher in the trace. As a result, an equivalent trace

may have already been explored when these branches are scheduled, leading to sleep set blocked sets. However, it obvious that these branches would lead to equivalent traces of lower bound count and thus, they would be add as conservative branches by the BPOR algorithm.
We have proved that Source-Bpor is sound since the traces explored by the bpor are subset of the traces explored by the Source-BPOR.

Chapter 5

Further Discussion on Bounding Problem

In this chapter alternative ideas of approaching the preemption bounding problem of the DPOR are discussed. Firstly, it is explained why a better solution, where conservative branches are utilized, is difficult to be found. Secondly, it is shown that optimizations that have already been used for the DPOR algorithm cannot solve the problem. Finally a new approach is suggested which is shown to be equivalent to the addition of conservative branches. This approach however can be used to better approximate a sound solution of the problem.

5.1 Conservative Branches

It has been shown in a previous chapter 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 unbounded state space. In order to explain this better an example with a writer and 3 readers is given in Figure 5.1.

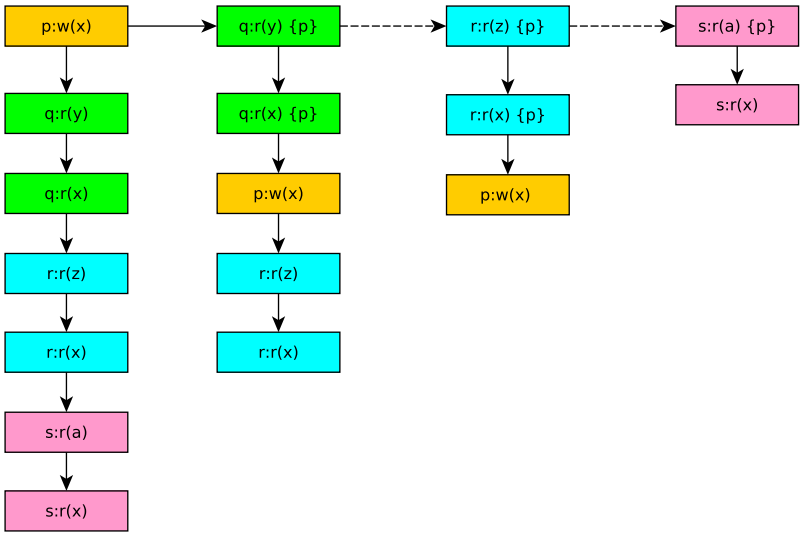

Figure 5.1: writer-3readers explosion

As we can infer from the example there are more states to be explored than the expected ones. This is caused by the addition of conservative branches. The extra trace suchs 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.

5.2 Sleep Sets

The results of the various bounding algorithms suggest that the number of sleep set blocked traces is subtle 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 the example below it is shown that the branch which would not have caused a increase of the bound is rejected.

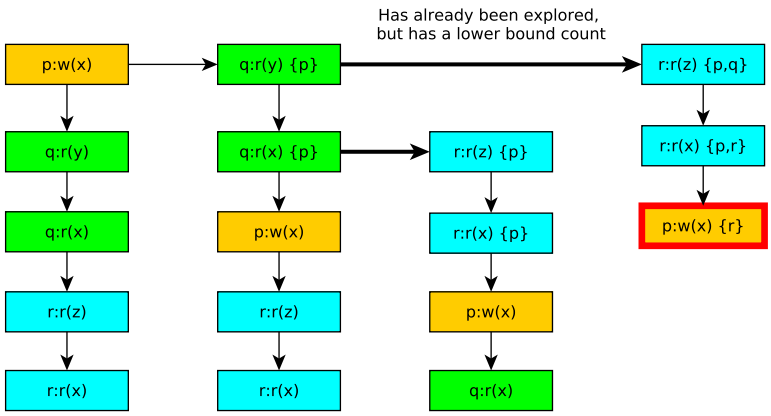

Figure 5.2: Sleep set contradiction

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 DPOR 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.
It is clear that the problem lies on the nature of the DPOR algorithm. 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.

5.3 Source Sets - Optimal DPOR

The source set technique which can lead many times to optimal coverage of the state space manages avoid the exploration of redundant sets. However as it was discussed in the previous section it avoids the scheduling of the traces. Unfortunately it cannot be used in the when conservative branches are added since these branches are not related to the sleep sets. As a result the conservative branches alone will never lead to sleep set blocked traces.
However, in many test cases there are some sleep set blocked traces which are caused by conditional reads and writes. When a technique which does not utilize the sleep sets is to be used these traces would have been easily eliminated. Unfortunately, it was experimentally shown that explored traces outnumber the sleepset blocked traces and, consequently, the implementation of such an algorithm would have a minor impact.
Moreover, we have shown that even if we maintain the source-set optimization for the non-conservative branches the results will be equivalent with using persistent sets. The idea of keeping the rejected traces from Source-DPOR that would have been added from DPOR (with persistent sets) was rejected since it harms the soundness of the algorithm.

5.4 Techniques without the Addition of Conservative Branches

It was shown that no apparent significant improvement can be made with the use of conservative branches. In this section, techniques without the usage of conservative branches are discussed.

5.4.1 Motivation

The only algorithm that does not add any conservative branch is the Vanilla-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 further explaining this idea.

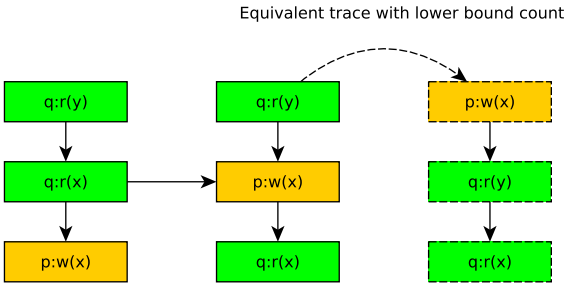

Figure 5.3: Motivation

As it is clear the preemptive switch that takes places would have been easily avoided there is an obvious inversion of the two blocks.
But what allows such an inversion?
The answer lies to the events of each block. 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 happen 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?

5.4.2 An Algorithm without Conservative Branches

An algorithm that would preform such a bounded search would be different from the Vanilla-BPOR only concerning the function calculates the bound count of the traces. This function of 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, $f(E) <= f(E.E')$ for any E'. The general form of the algorithm is given below:

Algorithm 14: 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
```

We notice that instead of calculating the Bv value we calculate minimum of all Bv values of the traces that are equivalent with $S.t$.

5.4.3 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 happen-before relations hold. Each equivalent trace should also compensate to these relations. It is also possible for different instructions in one block different happen before relations hold true. For this section only we will consider that a happen before relation is a relation that happens 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.

The existence of these happen before relations imply the existence of 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 happen 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 preemptive switch have weight 1.
The construction of the graph would not allow to traverse a block A that happens-before B before B, thus, all the happen before relations should still hold true. 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 15. 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 15: 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 ;
```

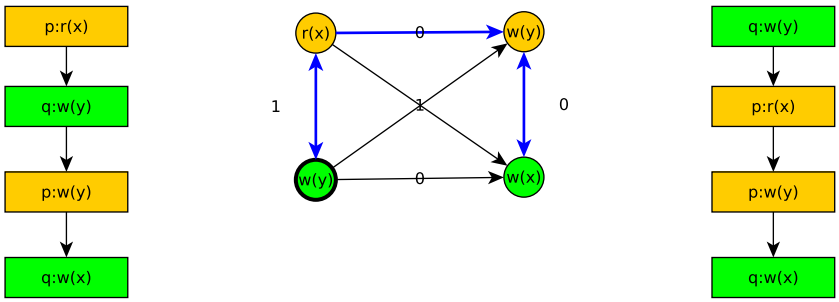

Figure 5.4: Graph example

In Figure 5.4 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 Vanilla-BPOR without the explosion caused by the conservative branches.

5.4.4 Approximating Bound Count

There are two approaches examined in order to approximate a value were considered. The notion of both algorithms is based on this observation: A preemption switch is compulsory when for two blocks of the same thread A a block of another thread B must intervene in order for the happen before relations to hold true. As a result the execution of the first A block should stop so the execution of the B block take 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 happen before relations and, thus construct an equivalent trace with lower bound count.
The algorithm is presented here:

Algorithm 16: First Estimation 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 the above algorithm 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 happen before relation vector clocks can be used. Moreover, it is obvious that more happen before relations can be counted.
Th second algorithm explores more state space than it is required.

Algorithm 17: Second Estimation 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     for i = lower_block + 1 to len(E) − 1 do
6         if E[lower_block] → E[i] → last(E) then
7             current_bound++;
8             return ;
```

This algorithm starts the search for an event that intervenes the two events of the same the immediately previous block with the same thread as the last one.

5.4.5 Evaluation

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.

5.4.6 Evaluation of Approximating algorithms

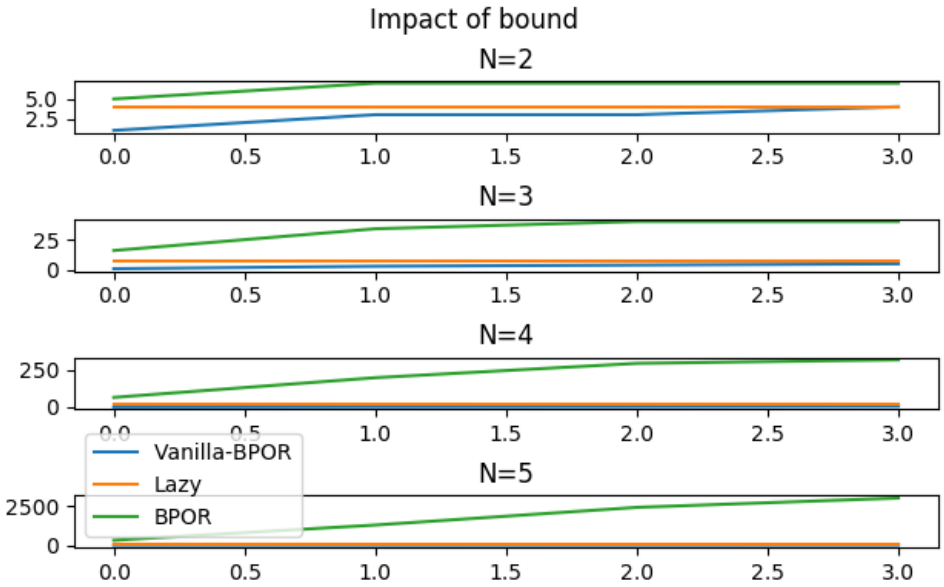

Figure 5.5: writer-N-readers bounded by the first estimation algorithm

Technique:	Vanilla-BPOR			Lazy-BPOR			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 5.1: Traces for the first estimation algorithm for various bound limits

Technique:	Vanilla-BPOR			Lazy-BPOR			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	45	258	883	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	6	8	2	8	14
indexermod.c	1	1	5	120	120	120	120	1320	7920

Table 5.2: Traces for the second estimation algorithm for various bound limits

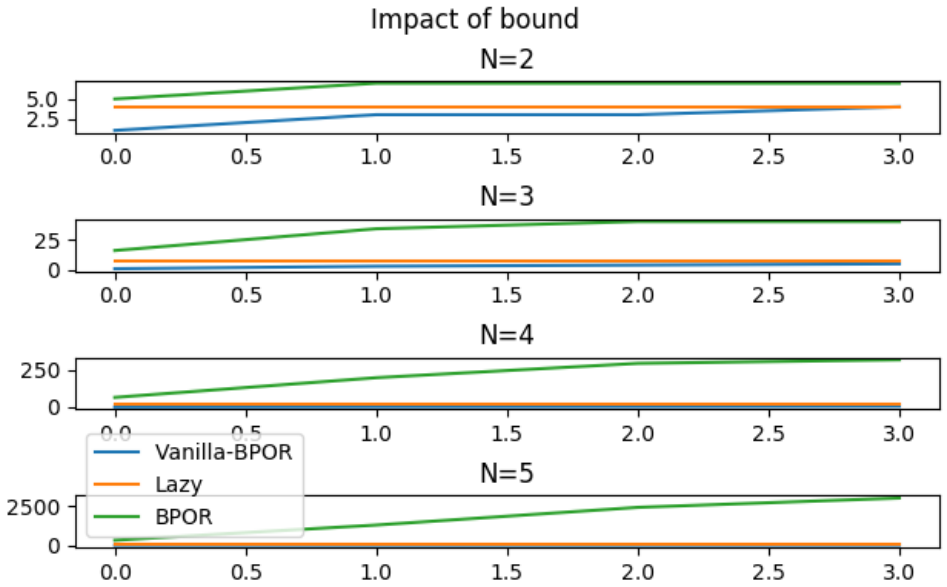

Figure 5.6: writer-N-readers bounded by the second estimation algorithm

5.4.7 Implementation of LBPOR

Some of the testcases 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 LBPOR, 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 Vanilla-BPOR is that the LBPOR 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 18: LBPOR

```
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{Hamiltonian_path(G')} <= b then
17                 Explore(E.p, Sleep, G', b) ;
18                 add p to Sleep ;
```

5.4.8 LBPOR - RCU Evaluation

The results are demonstrated below. Since we have to compare LBPOR 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.
Here we present the evaluation of the algorithm on RCU.
We compare the algorithm with BPOR. We notice that LBPOR examines less traces but requires longer time since the cost of the lazy bound count is significantly increased.

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 5.3: Comparison between DPOR and LBPOR

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 5.4: Comparison between DPOR and LBPOR without the bug

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	BPOR			LBPOR			BPOR			LBPOR			BPOR			LBPOR			BPOR			LBPOR			BPOR			LBPOR		
	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound
-DASSERT_0	183	2.65	3	104	2.79	2	106	2.96	3	73	4.06	2	128	5.39	3	85	8.57	2	118	5.28	3	75	6.28	2	128	5.91	3	85	9.44	2
-DFORCE_FAILURE_1	275	3.74	4	141	3.45	3	182	5.02	4	121	8.68	3	300	12.69	4	163	21.73	3	220	9.73	4	123	11.28	3	300	13.93	4	163	23.54	3
-DFORCE_FAILURE_2	6	0.35	1	4	0.35	1	5	0.54	1	3	0.52	0	5	0.75	1	3	0.71	0	5	0.91	1	3	0.87	0	5	0.95	1	3	0.9	0
-DFORCE_FAILURE_3	2	0.2	NF	38	1.87	NF	201	6.49	2	200	54.62	1	258	12.11	2	233	103.89	1	258	12.59	2	233	107.1	1	258	12.84	2	233	111.37	1
-DFORCE_FAILURE_4	47	0.91	2	51	1.38	2	41	1.78	2	24	2.1	1	21	1.89	2	14	1.79	1	24	2.3	2	17	2.27	1	24	2.39	2	17	2.34	1
-DFORCE_FAILURE_5	2	0.19	NF	38	3.96	NF	60	2.26	4	52	3.58	3	60	3.12	4	52	5.26	3	60	3.47	4	52	5.66	3	60	3.61	4	52	5.92	3
-DFORCE_FAILURE_6	1	0.95	0	1	0.94	0	2	2.74	0	2	2.77	0	2	4.47	0	2	4.33	0	2	8.7	0	2	8.45	0	2	8.73	0	2	8.56	0

Table 5.5: Comparison between BPOR and LBPOR(buged)

Chapter 6

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.

[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.

[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.

[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.

[Edmu99] Orna Grumberg Edmund M. Clarke, Marius Minea and Doron A. Peled, "State Space Reduction Using Partial Order Techniques", *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[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.

[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*, pp. 147–186, 1997.

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

[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.

[Kath13] Kathryn S. McKinley Katherine E. Coons, Madanlal Musuvathi, "Bounded Partial Order Reduction", October 2013. Available: http://lwn.net/Articles/262464/.

[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.

[Koko17] Michalis Kokologiannakis and Konstantinos Sagonas, "Stateless Model Checking of the Linux Kernel's Hierarchical Read-Copy-Update (Tree RCU)", July 2017. Available: https://arxiv.org/abs/1610.03052.

[Lamp78] Leslie Lamport, "Time, Clocks and Ordering of Events in Distributed Systems", 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.

[M07] MUSUVATHI M. and QADEER S., "Iterative context bounding for systematic testing of multithreaded programs", 2007.

[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.

[Mich18] Konstantinos Sagona Michalis Kokologiannakis, Ori Lahav and Victor Vafeiadis, "Effective Stateless Model Checking for C/C++ Concurrency", *Proceedings of the ACM on Programming Languages*, 2018.

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

[P97] GODEFROID P. and PIROTTIN D., "Refining dependencies improves partial-order verification methods", 1997.

[Paro17a] Bengt Jonsson Parosh Abdulla, Stavros Aronis and Konstantinos Sagonas, "Comparing Source Sets and Persistent Sets for Partial Order Reduction", March 2017. Available: https://link.springer.com/chapter/10.1007/978-3-319-63121-9_26.

[PARO17b] BENGT JONSSON PAROSH AZIZ ABDULLA, STAVROS ARONIS and KONSTANTINOS SAGONAS, "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction", September 2017. Available: https://www.researchgate.net/publication/319277772_Source_Sets_A_Foundation_for_Optimal_Dynamic_Partial_Order_Reduction.

[Paro18] Bengt Jonsson Parosh Aziz Abdulla, Stavros Aronis and Konstantinos Sagonas, "Effective Stateless Model Checking for C/C++ Concurrency", *Proceedings of the ACM on Programming Languages*, 2018.

[Paul16] Adam Betts Paul Thomson, Alastair F. Donaldson, "Concurrency Testing Using Controlled Schedulers: An Empirical Study", March 2016. Available: http://www.doc.ic.ac.uk/~afd/homepages/papers/pdfs/2016/TOPC.pdf.

[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.

[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

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;
}
```