Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Τεχνικές Περιορισμού στην Αναγωγή σε Δυναμικές Σχέσεις Μερικής Διάταξης

Διπλωματική Εργασία

ΙΩΑΝΝΗΣ-ΠΕΤΡΟΣ ΣΑΧΙΝΟΓΛΟΥ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2018

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Τεχνικές Περιορισμού στην Αναγωγή σε Δυναμικές Σχέσεις Μερικής Διάταξης

Διπλωματική Εργασία

ΙΩΑΝΝΗΣ-ΠΕΤΡΟΣ ΣΑΧΙΝΟΓΛΟΥ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Ιουλίου 2018.

.....	.....	.....
Κωνσταντίνος Σαγώνας Αναπληρωτής Καθηγητής ΕΜΠ	Νικόλαος Σ. Παπασπύρου	Νεκτάριος Κοζύρης Καθηγητής ΕΜΠ

Αθήνα, Ιούλιος 2018

Ιωάννης-Πέτρος Σαχίνογλου
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Η διεξοδική δοκιμή και επαλήθευση προγραμμάτων γραμμένων στο πρότυπο του ταυτοχρονισμού είναι μία τόσο σημαντική όσο και απαιτητική εργασία. Προκειμένου να επαληθεύσουμε την ορθότητα ενός τέτοιου προγραμμάτος θα πρέπει να εξετάσουμε όλες τις δυνατές δρομολογήσεις που αυτό μπορεί να παράξει. Το Stateless Model Checking με τη χρήση αναγωγής σε δυναμικές σχέσεις μερικής διάταξης (Dynamic Partial Order Reduction ή DPOR) είναι μια τεχνική, η οποία αντιμετωπίζει το πρόβλημα της έκρηξης του μεγέθους του χώρου καταστάσεων. Παρολα αυτά, η επαλήθευση μεγάλων προγραμμάτων μπορεί να διαρκέσει περισσότερο από όσο θα επιθυμούσαν οι προγραμματιστές. Σε αυτές τις περιπτώσεις ο περιορισμός (οριοθέτηση) της αναζήτησης (bounded search) με τη χρήση κάποιου ορίου μπορεί να αποδειχθεί αρκετά χρήσιμος. Η περιορισμένη αναζήτηση, σε αντίθεση με την DPOR απαλείφει το πρόβλημα της έκρηξης του μεγέθους του χώρου καταστάσεων αγνοώντας δρομολογήσεις που ξεπερνούν κάποιο όριο.

Στην παρούσα διπλωματική εργασία περιγράφεται η υλοποίηση του Preemption Pounded DPOR (BPOR) στο Nidhugg ένα εργαλείο για την ανεύρεση σφαλμάτων (bugs) που προκύπτουν από το μοντέλο του ταυτοχρονισμού και τα χαλαρά μοντέλα μνήμης (relaxed memory models). Συγκεκριμένα, υλοποιήθηκαν τρεις τεχνικές περιορισμού: ο Naive-BPOR, ο Nidhugg-BPOR και ο Source-BPOR. Οι τεχνικές αυτές αξιολογήθηκαν τόσο σε συνθετικά προγράμματα, όσο και σε λογισμικό που χρησιμοποιείται στην βιομηχανία. Πιο αναλυτικά, η ορθότητα του μηχανισμού Read-Copy-Update του πυρήνα του Linux επαληθεύτηκε ξανά με τη χρήση αυτών των τεχνικών. Επιπροσθέτως, εξετάστηκε κατά πόσο οι βελτιστοποιήσεις που εφαρμόζονται στον μη περιορισμένο DPOR μπορούν να βελτιώσουν την επίδοση του BPOR.

Λέξεις κλειδιά

Τυπικές Μέθοδοι Επαλήθευσης, Stateless Model Checking, Συστηματικός Έλεγχος Ταυτοχρονισμού, RCU, Read-Copy-Update, Περιορισμένη Αναγωγή σε Δυναμικές Σχέσεις Μερικής Διάταξης

Abstract

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

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

Key words

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

Ευχαριστίες

Πρώτα από όλα, θα ήθελα να ευχαριστήσω τον επιβλέποντά μου, Κωστή Σαγώνα, για τη βοήθεια και την υποστήριξη κατά την εκπόνηση της παρούσας διπλωματικής εργασίας. Εκτός από την υποστήριξη που μου παρείχε, με ενέπνευσε με τον ενθουσιασμό του και το ενδιαφέρον του για τη δουλειά μου. Επιπλέον, θα ήθελα να ευχαριστήσω όλη την ομάδα του Κωστή Σαγώνα τόσο στο Εθνικό Μετσόβιο Πολυτεχνείο όσο και στο Πανεπιστήμιο της Ουψάλα και ειδικά τον Σταύρο Αρώνη και τον Παναγιώτη Φυτά για την πολύτιμη βοήθειά τους.

Τέλος, θα ήθελα να ευχαριστήσω την οικογένεια μου που με υποστήριξε όλα αυτά τα χρόνια και που χωρίς αυτήν δεν θα ήμουν σε θέση να πετύχω τους στόχους μου.

Ιωάννης-Πέτρος Σαχίνογλου,

Αθήνα, 6η Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

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

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 11

Κατάλογος πινάκων 15

Κατάλογος σχημάτων 17

Κώδικες 19

Κατάλογος Αλγορίθμων 21

1. Εισαγωγή 23
1.1 Δοκιμή, Έλεγχος και Επαλήθευση Ταυτόχρονων Προγραμμάτων 23
1.2 Σκοπός της Διπλωματικής 24
1.3 Επισκόπηση 24

2. Θεωρητικό Υπόβαθρο 27
2.1 Ταυτόχρονος Προγραμματισμός 27
2.2 Σφάλματα στο Μοντέλο του Ταυτοχρονισμού (Concurrency Errors) 27
2.3 Δοκιμή, Model Checking και Επαλήθευση 28
2.4 Stateless Model Checking και Partial Order Reduction 29
2.5 Διανυσματικά ρολόγια 30
2.6 Συμβολισμός 31
2.7 Εξαρτήσεις μεταξύ Γεγονότων 31
2.8 Ανεξαρτησία και Ανταγωνισμός 32
2.9 Dynamic Partial Order Reduction 33
2.10 Επίμονα Σύνολα - Persitent Sets 33
2.11 Πηγαία Σύνολα - Source Sets 34
2.12 Κοιμώμενα Σύνολα - Sleep Sets 35
2.13 Σύγκριση Persistent Sets με Source Sets 37
2.14 Περιορισμένη Αναζήτηση και Περιορισμός ως προς την Προτίμηση 37
2.15 Φραγμένα ως προς την Προτίμηση Επίμονα Σύνολα - Preemption Bounded Persistent Sets 38

3. DPOR χωρίς Περιορισμούς 41
3.1 Source-DPOR 41
3.2 Classic-DPOR 42
3.3 Σύγκριση μεταξύ Classic-DPOR και Source-DPOR 43

3.4 Συνδιάζοντας Classic-DPOR και Source-DPOR . . . 43

4. Τεχνικές περιορισμού για DPOR . . . 47
4.1 Naive-BPOR . . . 47
4.2 BPOR . . . 48
4.3 Nidhugg-BPOR . . . 49
4.4 Source-BPOR . . . 51
4.5 Προκλήσεις από την προσθήκη Συντηρητικών Διακλαδώσεων . . . 52
4.5.1 Η Αύξηση του Χώρου Καταστάσεων . . . 52
4.5.2 Η Κατάργηση των Sleep Sets . . . 53
4.5.3 Το Αντιστάθμισμα μεταξύ Επίδοσης και Ακρίβειας . . . 54

5. Υλοποιήσεις Αλγορίθμων . . . 55
5.1 Η Ροή Προγράμματος του Nidhugg . . . 55
5.2 Προσθήκη διακλαδώσεων στο Nidhugg . . . 56
5.3 Υλοποίηση του Nidhugg-DPOR . . . 57
5.4 Υλοποίηση του Naive-BPOR . . . 58
5.5 Υλοποίηση του Nidhugg-BPOR . . . 60
5.6 Υλοποίηση του Source-BPOR . . . 61

6. Αξιολόγηση των Υλοποιηθέντων Αλγορίθμων . . . 63
6.1 Συνθετικά Προγράμματα . . . 63
6.2 RCU . . . 64
6.3 Αξιολόγηση των Μη Φραγμένων Αλγορίθμων . . . 64
6.3.1 Αξιολόγηση των Persistent Sets στα Συνθετικά Προγράμματα . . . 64
6.3.2 Αξιολόγηση των Persistent Sets στο RCU . . . 65
6.4 Σύγκριση με τα αποτελέσματα του Concuerror . . . 66
6.5 Αξιολόγηση Τεχνικών Περιορισμού . . . 67
6.5.1 Αξιολόγηση Τεχνικών Περιορισμού στα Συνθετικά Προγράμματα . . . 67
6.5.2 Αξιολόγηση των Τεχνικών Περιορισμού στο RCU . . . 68
6.5.3 Ένα Γνωστό Σφάλμα . . . 70
6.6 Ισοδυναμία μεταξύ Classic-BPOR και Source-BPOR (Ορθότητα του Source-BPOR) . . . 70

7. Επιπλέον Συζήτηση για το Πρόβλημα του Περιορισμού . . . 71
7.1 Τεχνικές χωρίς την Προσθήκη Συντηρητικών Διακλαδώσεων . . . 71
7.1.1 Κίνητρο . . . 71
7.1.2 Ένας Αλγόριθμος χωρίς Συντηρητκές Διακλαδώσεις . . . 72
7.1.3 Υπολογισμός του Ελάχιστου Preemption Count . . . 72
7.1.4 Προσεγγίζοντας το Preemption Count . . . 74
7.1.5 Αξιολόγηση των Αλγορίθμων Προσέγγισης . . . 75
7.1.6 Υλοποίηση του Lazy-BPOR . . . 75
7.1.7 Αξιολόγηση του Lazy-BPOR στο RCU . . . 76
7.2 Συμπεράσματα . . . 81

8. Επίλογος . . . 83

Βιβλιογραφία . . . 85

Παράρτημα 91

A. 91
A.1 Modifications in test suite 91

Κατάλογος πινάκων

6.1 Source-DPOR vs Nidhugg-DPOR for synthetic tests 65
6.2 Traces for various bound limits 68
6.3 RCU results without bound 69
6.4 RCU results for bound $b = 0$ 69
6.5 RCU results for bound $b = 1$ 69
6.6 RCU results for bound $b = 2$ 69
6.7 RCU results for bound $b = 3$ 69
6.8 RCU results for bound $b = 4$ 69
6.9 Comparison between DPOR and BPOR 70
6.10 Comparison between DPOR and BPOR with the bug 70

7.1 Traces for the first estimation algorithm for various bound limits 76
7.2 Comparison between DPOR and Lazy-BPOR 77
7.3 Comparison between DPOR and Lazy-BPOR without the bug 78
7.4 Comparison between BPOR and Lazy-BPOR 80

Κατάλογος σχημάτων

2.1 Comparing Testing, Model Checking and Verification . . . 29
2.2 Clock example . . . 30
2.3 Construction of persistent sets . . . 34
2.4 Program with non-minimal persistent sets . . . 37
2.5 Example of blocks . . . 39

3.1 Construction of persistent sets in Nidhugg when there is a write process . . 45
3.2 Construction of persistent sets in Nidhugg when both are read processes . . 45

4.1 Naive-BPOR for bound=0 . . . 48
4.2 Usage of non-conservative branches . . . 50
4.3 Example of BPOR execution . . . 51
4.4 Following source sets for conservative branches . . . 51
4.5 writer-3-readers explosion . . . 53
4.6 Sleep set contradiction . . . 54

5.1 Nidhugg's Flow Chart . . . 56
5.2 Execution without the scheduling optimization . . . 60

6.1 writer-N-readers . . . 65
6.2 Lastzero Concuerror . . . 66
6.3 Scheduling Effect reader-writer-reader . . . 67
6.4 Scheduling Effect writer-reader-reader . . . 67
6.5 writer-N-readers bounded . . . 68

7.1 An example of avoidable preemption-switch . . . 71
7.2 Graph example . . . 73
7.3 writer-N-readers bounded by the first estimation algorithm . . . 75

Κώδικες

2.1 Example of non-concurrency error 28
2.2 Example of concurrency error 28
2.3 Vector Clock example 30
2.4 Vector Clock output 30
2.5 Sleep set example 36
5.1 Example of bound counter 59
5.2 Naive-BPOR output 59

List of Algorithms

1 General form of DPOR 33
2 Bounded-DPOR 38

3 Source-DPOR 41
4 DPOR using Clock Vectors (Classic-DPOR) 42
5 Nidhugg-DPOR 44

6 Naive-BPOR 47
7 BPOR 49
8 Nidhugg-BPOR 50
9 Source-BPOR 52

10 see_events() 56
11 add_branch() 57
12 includes() routine 58
13 add_branch() for persistent sets 58
14 Should we increase the bound count? 58
15 see_events() for BPOR 60
16 try_to_add_conservative_branches() 61
17 add_branch() routine for Source-BPOR 61

18 General form of the BPOR without branch addition 72
19 Adding a new block to the dependencies' graph 73
20 First Approximation Algorithm 74
21 Second Approximation Algorithm 75
22 Lazy-BPOR 76

Κεφάλαιο 1

Εισαγωγή

Ο νόμος του Moore, που πήρε το όνομά τους από τον συνιδρυτή της Intel, Gordon Moore, αναφέρει ότι ο αριθμός των τρανζίστορ που μπορούν να τοποθετηθούν σε ένα ολοκληρωμένο κύκλωμα διπλασιάζεται περίπου κάθε δυο χρόνια. Για δεκαετίες οι κατασκευαστές πετύχαιναν την συρρίκνωση των διαστάσεων των chip, επιτρέποντας στο νόμο του Moore να συνεχίζει να επαληθεύεται ενώ οι τελικοί καταναλωτές συνέχιζαν να απολαμβάνουν ακόμα πιο ισχυρούς φορητούς υπολογιστές, tablets και έξυπνα τηλέφωνα. Από την άλλη οι μηχανικοί λογισμικού μπορούσαν απλώς να περιμένουν το νόμο του Moore να συνεχίζει να ισχύει ώστε να μπορούν να κατασκευάσουν ακόμα πιο απαιτητικά προγράμματα. Παρολ' αυτά περιορισμοί όπως η αύξηση της θερμοκρασία και η συχνότητα των ρολογιών εμποδίζουν την περεταίρω βελτίωση στην επίδοση του λογισμικού. Προκειμένου οι προγραμματιστές να ικανοποιήσουν την απαίτηση για αποδοτικό λογισμικό, πρότυπα προγραμματισμού όπως αυτό του ταυτοχρονισμού έχουν γίνει αναγκαία. Η χρήση αυτού του προτύπου όμως δημιουργεί νέες προκλήσεις καθώς ο προγραμματισμός γίνεται πιο δύσκολος και πιο επιρρεπής σε λάθη από το ακολουθιακό πρότυπο.

Συγκεκριμένα, συχνά προβλήμα που εμφανίζονται στο μοντέλο του ταυτοχρονισμού είναι:

Race conditions Όπου μία δρομολόγηση έχει μη αναμενόμενο αποτέλεσμα.

Deadlocks Δύο οι περισσότερες διεργασίες σταματούν και περιμένουν η μία την άλλη.

Livelocks Δύο οι περισσότερες διεργασίες συνεχίζουν να εκτελούνται χωρίς να σημειώνουν πρόοδο.

Resource starvations Δύο οι περισσότερες διεργασίες σταματούν να εκετελούνται περιμένοντας για πόρους.

Το χειρότερο όμως είναι ότι αυτάτ τα προβλήματα χαρακτηρίζονται συνήθως ως Heisenbugs [Musu08]; Δηλαδή, μπορεί να αλλάξουν συμπεριφορά ή να εξαφανιστούν τελείως όταν κάποιος προσπαθήσει να τα απομονώσει καθώς σχετίζονται άμεσα με τη σειρά που οι διεργασίες εκτελούνται.

1.1 Δοκιμή, Έλεγχος και Επαλήθευση Ταυτόχρονων Προγραμμάτων

Η δοκιμή και η επαλήθευση ταυτόχρονων προγραμμάτων είναι μια απαιτητική διαδικασία. Μια τεχνική που χρησιμοποιείται για την εξερεύνηση του χώρου καταστάσεων είναι ο έλεγχος του μοντέλου (model checking) [Wikib]. Το model checking είναι μια μέθοδος για την τυπική επιβεβαίωση ταυτόχρονων προγραμμάτων και συστημάτων μέσω απαιτήσεων για το σύστημα εκφρασμένων σε λογική φόρμουλα και την χρήση αποδοτικών αλγορίθμων που μπορούν να ελέγξουν το ορισθέν μοντέλο όπως έχει οριστεί από το

σύστημα ώστε να ελεγχθεί αν οι απαιτήσεις εξακολουθούν να ισχύουν. Το μεγάλο πρόβλημα με τα εργαλία που κάνουν model checking είναι ότι πρέπει να αντιμετωπίσουν την εκθετική αύξηση του χώρου καταστάσεων καθώς ένας μεγάλος αριθμός καταστάσεων πρέπει να αποθηκευτεί. Πολλές τεχνικές έχουν προταθεί προκειμένου να αντιμετωπιστεί αυτό το πρόβλημα. Το Stateless Model Checking, για παράδειγμα, αποφεύγει την αποθήκευση καθολικών καταστάσεων. Αυτή η τεχνική για παράδειγμα έχει υλοποιηθεί σε εργαλεία όπως το Verisoft [Gode97, Gode05], CHESS [Musu08], Concuerror [Chri13], Nidhugg [Abdu15] και RCMC [Koko17a]. Η παρατήρηση ότι δύο δρομολογήσεις είναι ισοδύναμες αν η μία μπορεί μπορεί να προκύψει από την άλλη με την εναλλαγή εκτέλεσης ανεξάρτητων βημάτων είναι ο πυρήνας της ιδέας της αναγωγής σε δυναμικές σχέσεις μερικής διάταξης (Partial Order Reduction) [Valm91, Pele93, Gode96, Clar99, Abdu17b] και χρησιμοποιείται σε διάφορα εργαλεία. Το Dynamic Partial Order Reduction (DPOR) εντοπίζει εξαρτήσεις μεταξύ ενεργειών διαφορετικών νημάτων ενώ το πρόγραμμα εκτελείται [Flan05, Abdu17b]. Η εξερεύνηση ξεκινάει με μια αυθαίρετη δρομολόγηση της οποίας τα βήματα στη συνέχεια χρησιμποιούνται για την αναγνώρηση λειτουργιών όπου εναλλακτικές δρομολογήσεις πρέπει να εξερευνηθούν προκειμένου να συλληφθεί η συνολική συμπεριφορά του προγράμματος. Μία άλλη προσέγγιση είναι το οριοθετημένο (bounded) model checking [Bier03] όπου ένας πεπερασμένος αριθμός από βήματα ξεδιπλώνεται και και οι απαιτήσεις ελέγχονται γι αυτό τον αριθμό βημάτων. Το bounded model checking μπορεί να συνδιαστεί με το partial order reduction για να μοντελοποιήσει εκετλέσεις και έχει υλοποιηθεί αποδοτικά σε εργαλία όπως τα CBMC [Clar04], Nidhugg [Abdu14] and RCMC [Koko17a]. Δυστυχώς, όλες αυτές οι τεχνικές εξακολουθούν να πάσχουν σε κάποιο βαθμός από το πρόβλημα της έκρηξης του state space. Προκειμένου να αντιμετωπιστείς αυτό το πρόβλημα επιπλέον οριοθέτηση απαιτείται. Πόλλες και διαφορετικές τεχνικές έχουν εξεταστεί [Thom16], όπως το preemption bounding, delay bounding, ένας ελεγχόμενος τυχαίος δρομολογιτής και η πιθανοτική δοκιμή του ταυτοχρονισμού (probabilistic concurrency testing, PCT).

1.2 Σκοπός της Διπλωματικής

Στόχος της παρούσας διπλωματικής είναι:

- Η υλοποίηση preemption bounding τεχνικών για τον DPOR [Coon13] στο εργαλείο Nidhugg.
- Να εξεταστεί κατα πόσο τεχνικές όπως το Source-DPOR και Optimal-DPOR [Abdu14] μπορούν να ενισχύσουν την υλοποίηση του bounded partial order reduction, κατα προτίμηση με την παροχή κάποιου είδους εγγυήσεων.
- Η επιβεβαίωση ή όχι της δυνατότητας του bounded dynamic partial order reduction να εντοπίζει λάθη γρηγορότερα από το unbounded partial order reduction.
- Να ελεγχθεί κατα πόσο η εμπειρική παρατήρηση ότι τα λάθη μπορούν να εμφανιστούν σε μικρό αριθμό απο preemptions [Musu07] είναι σωστή.
- Η εξερεύνηση εναλλακτικών προσεγγίσεων στο προβλήμα του preemption-bounded partial order reduction.

1.3 Επισκόπηση

Στο Κεφάλαιο 2 παρουσιάζουμε το θεωρητικό υπόβαθρο τόσο του unbounded όσο και του bounded DPOR. Στο Κεφάλαιο 3 και 4 παρουσιάζονται οι αλγόριθμοι για unbounded

DPOR και bounded DPOR. Στο Κεφάλαιο 5 συζητόνται οι τεχνικές λεπτομέριες των υλοποιήσεων. Η αξιολόγηση κάθε αλγορίθου δίνεται στο Κεφάλαιο 6 όπου οι αλγόριθμοι δοκιμάζονται τόσο με τη χρήση συνθετικών τεστ όσο και με τη χρήση κώδικα σημαντικού μεγέθους (RCU) που είναι μέρος του πυρήνα του Linux. Στο Κεφάλαιο 7 παρουσιάζεται μια εναλλακτική προσέγγιση στο bounding problem. Τελος στο Κεφάλαιο 8 συνοψίζουμε τα προηγούμενα κεφάλαια και καταλήγουμε σε κάποια συμπεράσματα ενώ παρουσιάζουμε και κάποιες πιθανές προεκτάσεις για την παρούσα δουλειά.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Ταυτόχρονος Προγραμματισμός

Ο ταυτόχρονος (concurrent) υπολογισμός, που υλοποιείται με το πρότυπο του ταυτόχρονου προγραμματισμού, είναι μια μορφή υπολογισμού όπου ξεχωριστές υπολογιστικές μονάδες εκτελούν υπολογισμούς σε επικαλυπτόμενα χρονικά διαστήματα αντί να τους εκτελούν ακολουθιακά (όπου ένας υπολογισμός ολοκληρώνεται, προτού αρχίσει ένας άλλος). Τέτοιους υπολογισμούς μπορεί να εκτελεί ένα σύστημα, ένα πρόγραμμα ή ακόμα κι ένα δίκτυο. Σε ένα concurrent σύστημα, ένας υπολογισμός μπορεί να προχωρήσει χωρίς να περιμένει κάποιον προηγούμενο να ολοκληρωθεί. Οι βασικές προκλήσεις στο σχεδιασμό ενός τέτοιου συστήματος είναι το concurrency control: δήλαδη η διασφάληση της σωστής αλληλουχίας των υπολογισμών, η αλληλεπίδραση ή η επικοινωνία μεταξύ των διαφορετικών υπολογιστικών μονάδων και ο συντονισμός στην πρόσβαση σε πόρους που μοιράζονται μεταξύ των διαφόρων υπολογισμών. Στα πιθανά προβλήματα περιλαμβάνονται τα race conditions, deadlocks, livelocks και resource starvation. Ο δρομολογιτής είναι συνήθως υπεύθυνος για την εκετέλεση ένος νήματος ή μιας διεργασίας. Λόγω του μη ντετερμινισμού ο προγραμματιστής δεν μπορεί πάντοτε να είναι σε θέση να γνωρίζει ποιό νήμα θα δρομολογηθεί στη συνέχεια.

Μια σημαντική έννοια στον ταυτόχρονο προγραμματισμμό είναι η ιδέα του συνόλου των interleaving δηλαδή του συνόλου όλων την δυνατών εκτελέσεων που μπορεί να ακολουθήσει ένα πρόγραμμα. Διαισθητικά, αν φανταστούμε μια διεργασία (πιθανότατα άπειρη) ως μια ακολουθία από statetements (που μπορεί να έχουν προκύψει από την "ξεδίπλωση" βρόχων ή loop unfolding), τότε το σύνολο όλων των δυνατών interleaving των διεργασιών αποτελείται από όλες τις δυνατές ακολουθίες από αυτά τα statements.

Όπως μπορούμε να συμπεράνουμε η αποσφαλμάτωση (debugging) τέτοιων προγραμμάτων μπορεί να γίνει πάρα πολύ δύσκολη. Η πρόκληση προκύπτει κυρίως από το γεγονός ότι δεν είναι πάντα ξεκάθαρο ποιό νήμα ή διεργασία θα εκτελεστεί. Επιπλέον, τα σφάλματα δεν εμφανίζονται πάντα κατα τη διάρκεια του debugging καθώς μόνο ένας πολύ μικρός αριθμός από interleavings μπορεί να οδηγεί στην εκδήλωση του σφάλματαος.

2.2 Σφάλματα στο Μοντέλο του Ταυτοχρονισμού (Concurrency Errors)

Σε αυτό το σημείο είναι σημαντικό να εισάγουμε την έννοια του concurrency error και να εξηγήσουμε πως αυτό διαφέρει από τα υπόλοιπα σφάλματα.

Definition 2.1. (Concurrency Error) Ένα concurrency error είναι ένα σφάλμα που προκύπτει από τον μη ντετερμινισμό του δρομολογητή.

Ένα παράδειγμα προγράμματος που περιέχει concurrency error δίνεται στον Κώδικα 2.2. Σε αυτό το πρόγραμμα η μεταβλητή x ισούται με 1 στην αρχή της εκτέλεσης του προγράμματος. Παρολ' αυτά, αν το thread $zero$ δρομολογηθεί πριν το thread $divider$ μια

διαίρεση με το μηδέν θα λάβει χώρα. Από την άλλη μεριά, μια απλή διαίρεση με το 0 δεν μπορεί να θεωρηθεί concurrency error σε μια περίπτωση, όπως φαίνεται στο Κώδικα 2.2 όπου η διαίρεση με το μηδέν θα γίνει χωρίς την παρέμβαση του δρομολογητή.

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

Listing 2.1: Example of non-concurrency error

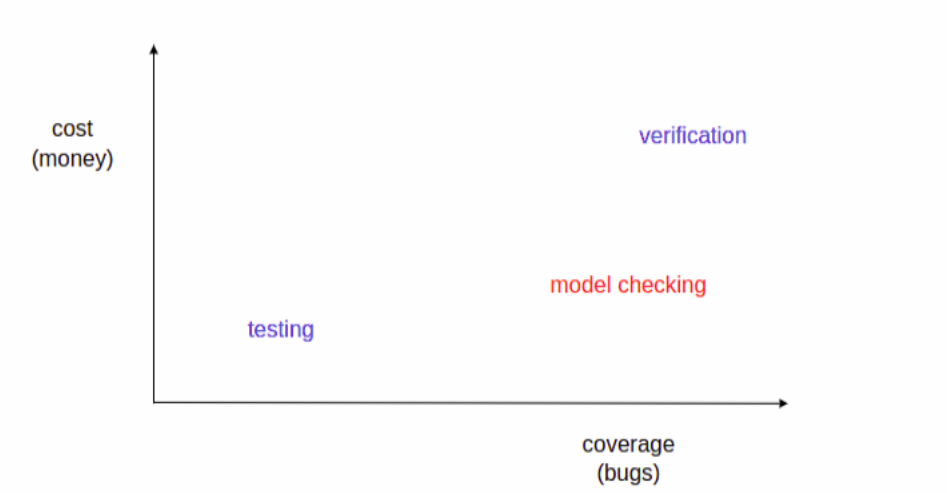
```
volatile int x = 1;

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

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

Listing 2.2: Example of concurrency error

2.3 Δοκιμή, Model Checking και Επαλήθευση

Το Dynamic software model checking είναι μια μορφή συστηματικού testing που είναι εφαρμόσιμο σε βιομηχανικού μεγέθους λογισμικό. Πολλά εργαλεία έχουν αναπτυχθεί τις τελευταίες δεκαετίες που χρησιμοποιούν αυτή την τεχνική με στόχο concurrent και data-driven software. Το model checking είναι πιο απαιτητικό υπολογιστικά από το παραδοσιακό software testing καθώς προσφέρει μεγαλύτερη κάλυψη (coverage) του προγράμματος. Παρολ' αυτά είναι φθηνότερος από γενικότερες μορφές επαληθευσης όπως το interactive theorem proving, το οποίο προσφέρει πολύ μεγαλύτερες εγγυήσεις. Επομένως, το dynamic software model checking προσφέρει μια ελκυστική πρακτική που μπορεί να συγκεράσει ως ένα βαθμό το testing και formal verification.

Το γράφημα που δίνεται στο Σχήμα 2.1 [Gode15] παρουσιάζει με πολύ σωστό τρόπο τις διαφορές μεταξύ testing, model checking και verification.



Σχήμα 2.1: Comparing Testing, Model Checking and Verification

2.4 Stateless Model Checking και Partial Order Reduction

Προκειμένου να βρούμε σφάλματα σε concurrent προγράμματα, πρέπει να ελέγξουμε όλα τα δυνατά interleaving, (όλους τους δυνατούς τρόπους που ένα πρόγραμμα μπορεί να εκτελεστεί) που το πρόγραμμα μπορεί να παράξει. Συνήθως αυτά τα λάθη προκύπτον υπό συγκεκριμένες συνθήκες τις οποίες ο προγραμματιστής δεν έλαβε υποψιν, κάνοντας τον εντοπισμό και διόρθωσή τους πολύ δύσκολες. Το Stateless model checking βασίζεται στην ιδέα της οδήγησης του προγράμματος σε όλα τα δυνατά interleavings. Δυστυχώς αυτή η προσέγγιση υποφέρει από το state explosion, δηλαδή ο αριθμός όλων των δυνατών interleavings αυξάνει εκθετικά σε σύγκριση με το μέγεθος του προγράμματος και τον αριθμό τον νημάτων ή διεργασιών. Πολλές προσεγγίσεις έχουν προταθεί προκειμένου να λύσουνν το συγκεκριμένο πρόβλημα όπως το partial order reduction [Gode96] και οι τεχνικές περιορισμού-οριοθέτησης (bounding techniques) [Coon13].

Το Partial order reduction στοχεύει την μείωση του αριθμου των interleavings που εξερευνώνται με την εξάλειψη ισοδύναμων interleavings. Κάθε interleaving μπορεί να παρουσιαστεί σαν ένα ίχνος (trace). Αυτά τα ισοδύναμα ίχνη παράγονται από την αντιστροφή ανεξάρτητων γεγονότων τα οποία δεν επηρεάζουν το αποτέλεσμα του προγράμματος. Για παράδειγαμ η δρομολογήση δύο νημάτων τα οποία διαβάζουν μια τοπική μεταβλητή (local variable) μπορεί να αντιστραφεί καθώς το αποτέλεσμα αυτών των ενεργειών δεν επηρεάζεται από τη σειρά που αυτές θα γίνουν. Υπάρχουν δύο τρόπου με του οποίους μπορούμε να κάνουμε partial order reduction. Ο πρώτος είναι το static partial order reduction [Kurs98] όπου η εξαρτήσεις μεταξύ των νημάτνω εντοπίζονται πρην την εκτέλεση του concurrent προγράμματος. Η δεύτερη προσέγγιση είναι το Dynamic partial order reduction (DPOR) [Flan05] το οποίο παρατηρεί τις εξαρτήσεις του προγράμματος κατά την εκτλέση του.

Για μεγάλα προγράμματα ο DPOR διαρκεί περισσότερ από όσο θα ήταν επιθυμητό. Σε αυτές τις περιπτώσεις τεχνικές περιορισμού μπορεί να φανούν χρήσιμες. Οι τεχνικές περιορισμού σε αντίθεση με τον DPOR, αντιμετωπίζουν το πρόβλημα του state space explosion με το να μην ελέγχουν δρομολογήσεις που ξεπερνούν ένα όριο [Thom16]. Έχουν προταθεί πολλές τέτοιες τεχνικές όπως το preemption bounded exploration [Coon13] ή το delay bounded exploration [Emmi11]. Όλες αυτές οι τεχνικές βασίζονται στην ιδέα ότι τα περισσότερα σφάλματαα μπορούν να εντοπιστούν ακόμα και με ένα μικρό όριο κάνοντας τον εντοπισμό των σφαλμάτων πολύ πιο γρήγορο.

Σε ότι αφορά αυτές τις τεχνικές, νέες προκλήσεις δημιουργούναι [Coon13] καθώς

εισάγονται περισσότερες εξαρτήσεις, που σχετίζονται με το όριο. Πολλά περιττά traces πρέπει να εξερευνηθούν προκειμένου να σεβαστούμε το όριο δεδομένου ότι οι αλγόριθμοι δεν είναι σε θέση να γνωρίζουν αν ισοδύναμα traces που αντιστοιχούν σε μεγαλύτερο bound έχουν ήδη ελεγχθεί.

Είναι σημαντικό να σημειωθεί ότι οι bounded techniques μπορούν να θεωρηθούν ως testing, με την έννοια ότι ελέγχουν μόνο ένα υποσύνολο του χώρου καταστάσεων αλλά και ως verification, με την έννοια ότ μπορούν επιβεβαιώσουν την απουσία σφαλμάτων για δεδομένο όριο.

2.5 Διανυσματικά ρολόγια

Ένα διανυσματικό ρολόι είναι ένας αλγόριθμος που προσδίδει μερική διάταξη σε κατανεμημένα ή ταυτόχρονα συστήματα και εντοπίζει παραβιάσεις αιτιότητας (causality violations). Ακριβώς όπως και στις χρονοσφραγίδες του Lamport (Lamport's timestamps) [Lamp78], διαδιεργασιακά μηνύματα περιλαμβάμνουν πληροφορίες για την πρόοδο του λογικού ρολογιού μιας διεργασίας. Το διανυσματικό ρολό ενός συστήματος μεf N διεργασίες έιναι ένα διάνυσμα από N λογικά ρολόγια, ένα ρολόι ανά διεργασία. Οι κανόνες με τους οποίους ενημερώνονται τα ρολογία δίνονται παρακάτω:

1. Κάθε διεργασία που εκτελεί μια δική της εντολή αυξάνει το λογικό ρολόι της κατά ένα.

2. Κάθε φορά που μια διεργασία λαμβάνει ένα μήνυμα ή εκτελεί μια ενέργεια σε μια μοιραζόμενη μεταβλήτη, αυξάνει το ρολόι της κατά ένα και ενημερώνει κάθε πεδίο του διανύσματς παίρνοντας το μέγιστο από τις τιμές από το δικό της διάνυσμα και τη μέγιστη τιμή από όλες τις διεργασίες που μοιράζονται αυτή τη μεταβλητή.

Ένα παράδειγμα εκτέλεσης του αλγορίθμου δίνεται στο Σχήμα 2.2 όπου δίνονται και ο πηγαίος κώδικας αλλά και το υπό εξερεύνηση ίχνος. Μπορούμε εύκολα να διαπιστώσουμε ότι σε κάθε έντολή του thread <0> (main thread) το ρολόι του main thread αυξάνεται. Όταν το thread <0.0> ξεκινάει να εκτελείται το ρολόι του για το thread <0> είναι 8 καθώς εκείνη τη στιγμή δημιουργήθηκε από το main thread. Όταν η τιμή του y διαβάζεται το ρολόι για το <0> αυξάνεται ξανά ώστε να αντιστοιχεί με το γεγονός y=1. Στη συνέχει το πρώτο νήμα δρομολογείται ξανά και το ρολόι του για το <0.0> είναι 7 καθώς η εντολή `pthread_join()` εκτελείται.

```
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]
```

Σχήμα 2.2: Clock example
Κάθε αλγόριθμος που παρουσιάζεται σε αυτή τη θέση βασίζεαται στα διανυσματικά ρολόγια.

2.6 Συμβολισμός

Πριν προχωρήσουμε βαθύτερα στο dynamic partial order reduction είναι πολύ σημαντικό να εξηγήσουμε τη σημειογραφία που θα χρησιμοποιήσουμε. Μια ακολουθία εκτέλεσης E ενός συστήματος που αποτελείται από περατό αριθμό βημάτων των διεργασιών του εκτελείται από την αρχική κατάσταση s_0. Δεδομένου ότι κάθε βήμα είναι ντερμινιστικό, μια ακολουθία εκτελέσεων E χαρακτιρίζεται κατά μοναδικό τρόπο απο την ακολουθία των διεργασίων που εκτελούν εντολές στο E. Για παράδειγμα, το $p.p.q$ συμβολίζει την εκτέλεση της ακολουθίας όπου το p εκτελεί δύο βήματα, ακολουθούμενο από ένα βήμα του q. Αυτή η ακολουθία από βήματα στο E ορίζει μόναδικά και το global state του συστήματος μετά την εκτέλεση του E, που συμβολίζεται με $s_{[E]}$. Για ένα state s, το $enabled(s)$ συμβολίζει το σύνολο των διεργασιών p που είναι ενεργές στο s (για τις οποίες η εκτέλεση $p(s)$ ορίζεται). Χρησιμοποιούμε το . για να συμβολίσουμε την ένωση (concatenation) ακολουθιών απο διεργασίες. Έτσι, αν η p δεν είναι μπλοκαρισμένη μετά το E, τότε $E.p$ είναι μια ακολουθία εκτελέσεων. Ένα γεγονός στο E είναι μια συγκεκριμένη εμφάνιση μια διεργασίες στο E. Χρησιμοποιούμε το $\langle p, i\rangle$ για να συμβολίζουμε το i-οστό γεγονός μιάς διεργασίας p στην εκτέλεση E. Με άλλα λόγια, το γεγονός $\langle p, i\rangle$ είναι το i-οστό βήμα τη διεργασίας p στην εκτέλεση E. Με το $dom(E)$ συμβολίζουμε το σύνολο των γεγονότων $\langle p, i\rangle$ τα οποία ανήκουν στο E, i.e., $\langle p, i\rangle \in dom(E)$ ανν E περιέχει τουλάχιστον i βήματα του p. Θα χρήσιμποιούμε τα $e, e', \ldots$, για αναφερόμαστε σε διάφορα γεγονότα. Το $proc(e)$ συμβολίζει τη διερτασία p ενός γεγονότος $e = \langle p, i\rangle$. Αν $E.w$ είναι μια εκτέλεση, που προέκυψε από τη συνένωση του E και του w, τότε το $dom_{[E]}(w)$ θα είναι $dom(E.w)$ $dom(E)$, δηλαδή τα γεγονότα στο $E.w$ τα οποία ανήκουν στο w. Μια ειδική περίπτωση είναι η εξής: χρησιμοποιούμε το $next_{[E]}(p)$ για να συμβολίσουμε το $dom_{[E]}(p)$. Το $<_E$ συμβολίζει τη συνολική διάταξη μεταξύ των γεγονότων του E, δηλαδή, το $e <_E e'$ συμβολίζει ότι το e συμβαίνει πριν από το e' στο E. Τέλος χρησιμοποιούμε το $E' \leq E$ για να συμβολίσουμε ότι η ακολουθία E' είναι πρόθεμα της ακολουθίας E.

2.7 Εξαρτήσεις μεταξύ Γεγονότων

Μια από τις πιο σημαντικές έννοιες όταν χρησιμοποιούμε έναν αλγόριθμο που κάνει αναζήτηση σε όλο τον χώρο καταστάσεων των διαφόρων δρομολογήσεων είναι η σχέση happens-before σε μια ακολουθία εκτέλεσης. Συνήθως αυτή η σχέση συμβολίζεται με $\rightarrow$. Για παράδειγμα η σχέση $\rightarrow$ για δύο γεγονότα e, e' στο $dom(E)$ είναι αληθής τότε το γεγονός e συμβαίνει πριν το e'. Αυτή η σχέση συνήθως εμφανίζεται στην ανατλλαγή μηνυμάτων όταν το e είναι η μετάδοση μηνύματος και το e' είναι το γεγονός της λήψης του μηνύματος. Για παράδειγμα στο Nidhugg το $e \rightarrow e'$ δε θα ήταν αληθές αν τουλάχιστον ένα από τα δύο γεγονότα δεν ήταν write operation στην ίδια μοιραζόμενη μεταβλητή. Είναι λογικό κάθε DPOR αλγόριθμος να μπορεί να προσδώσει τέτοιες happens-before σχέσεις. Πρακτικά η happens-before ανάθεση υλοποιείται με τη χρήση vector clocks.

Definition 2.2. (happens-before ανάθεση) Μια happens-before ανάθεση, η οποία αναθέτει μια μοναδική σχέση happens-before $\rightarrow E$ σε κάθε ακολουθία εκτέλεσης E, είναι έγκυρη αν ικανοποιεί τις ακόλουθες ιδιότητες για κάθε E.

1. Το $\rightarrow_E$ είναι μια μερική διάταξη στο $dom(E)$, που περιλαμβάνεται στο $<_E$. Με άλλα λόγια κάθε δρομολόγηση είναι μέρος μια μερικής διάταξης που μπορεί να παράξει το πρόγραμμα.

2. Τα βήματα εκτέλεση κάθε διεργασίας είναι πλήρως διατεταγμένα, δηλαδή $\langle p, i\rangle \rightarrow_E \langle p, i+1\rangle$ όποτε ισχύει $\langle p, i+1\rangle \in dom(E)$.

3. Αν E' είναι πρόθεμα του E τότε $\rightarrow_E$ και $\rightarrow_{E'}$ είναι ίδια στο $dom(E')$.

4. Κάθε γραμμικοποίηση (linearization) E' of $\rightarrow_E$ στο $dom(E)$ είναι μια ακολουθία εκτέλεσης ακριβώς ίδια με τη "happens-before" σχέση. $\rightarrow_{E'}$ as $\rightarrow_E$. Αυτό σημαίνει ότι η σχέση $\rightarrow_E$ επάγει ένα σύνολο από ισοδύναμες ακολουθίες εκτέλεσης, όλες με την ίδια "happens-before" σχέση. Το $E \simeq E'$ συμβολίζει ότι τα E και E' είναιε γραμμικοποιήσεις της ίδιας "happens-before" σχέσης, και το $[E] \simeq$ συμβολίζει την ισοδυναμία στην περίπτωση του E.

5. Αν $E \simeq E'$ τότε $s_{[E]} = s_{[E']}$ (δύο ισοδύναμα traces θα οδηγήσουν στο ίδιο state).

6. Για μια ακολουθία E, E' και w, ώστε η $E.w$ είναι μια ακολουθία εκτέλεσης, έχουμε ότι $E \simeq E'$ ανν $E.w \simeq' E'.w$.

2.8 Ανεξαρτησία και Ανταγωνισμός

Μπορούμε πλέον να ορίσουμε την ανεξαρτησία μεταξύ υπολογισμών. Αν $E.p$ και $E.w$ είναι δύο ακολουθίες εκτέλεσης, τότε το $E \models p\diamondsuit w$ συμβολίζει ότι το $E.p.w$ είναι μια ακολουθία εκτέλεσης τέτοια ώστε $next_{[E]}(p) \not\rightarrow_{E.p.w} e$ για κάθε $e \in dom([E.p])(w)$. Με άλλα λόγια, το $E \models p\diamondsuit w$ δηλώνει ότι το επόμενο γεγονό του p δε θα "συμβεί πριν" από κάποιο άλλο στο w στην ακολουθία εκτέλεσης $E.p.w$. Διαισθητικά, αυτό σημαίνει ότι το p είναι ανεξάρτητο από το w μετά το E. Στην ειδική περίπτωση όπου το w περιέχει μόνο μια διεργασία q, τότε το $E \models p\diamondsuit q$ συμβολίζει ότι τα επόμεντα βήματα των p και q είναι ανεξάρτητα μετά το E. Το $E' \models p\diamondsuit w$ συμβολίζει ότι το $E \not\models p\diamondsuit w$ δεν ισχύει.
Για μια ακολουθία w με $p \in w$, let $w\backslash p$ συμβολίζουμε την ακολουθία w με την πρώτη εμφάνσιση του p να έχει αφαιρεθεί, και το $w \uparrow p$ συμβολίζει το πρόθεμα του w μέρχι αλλά χωρίς να συμπεριλαμβάνει την πρώτη εμφάνιση του p. Για μια ακολουθία εκτέλεσης E και ένα γεγονός $e \in dom(E)$, έστω ότι το $pre(E, e)$ συμβολίζει το πρόθεμα του E μέχρι αλλά χωρίς να συμπεριλαμβάνει το e. Για μια ακολουθία εκτέλεσης E και ένα γεγονός $e \in E$, το $notdep(e, E)$ είναι η υπακολουθία του E που αποτελείται από τα γεγονότα που συμβαίνουν μετά το e αλλά δε "συμβαίνουν μετά" το e (δηλαδή τα γεγονότα e' που συμβαίνουν μετά το e για τα οποία ισχύει $e \not\rightarrow_E e'$).
Μια κεντρική έννοια στους περισσότερους DPOR αλγορίθμου είναι αυτή του ανταγωνισμού. Διαισθητικά, δύο γεγονότα e και e' σε μια ακολουθία εκτέλεσης E, όπου το e συμβαίνει πριν το e' στο E, συναγωνίζοναι αν

- Το e συμβαίνει πριν το e' στο E, και
- τα e και e' είναι ταυτόχρονα, δηλαδή υπάρχει μια ισοδύναμη ακολουθία εκτέλεσης $E' \simeq E$ στην οποία τα e και e' είναι γειτονικά.

Τυπικά, έστω ότι τα $e \lessdot_E e'$ συμβολίζουν ότι $proc(e) \neq proc(e')$, ότι $e \rightarrow_E e'$, και ότι δεν υπάρχει γεγονός $e'' \in dom(E)$, διαφορετικό από τα e' και e, τέτοιο ώστε $e \rightarrow_E e'' \rightarrow_E e'$. Όποτεδήποτε ο DPOR εντοπίζει συναγωνισμό, ελέγχει αν τα γεγονότα που συναγωνίζονται μπορούν να εκτελεστούν σε αντίστροφη σειρά. Δεδομένου ότι τα γεγονότα συνδέονται με σχέσεις happens-before, μπορεί να οδηγηθούμε σε διαφορετικά global state: έτσι ο αλγόριθμος πρέπει να προσπαθήσει να εξερευνήσει την αντίστοιχη ακολουθία εκτέλεσης Έστω ότι το $e \lesssim_E e'$ συμβολίζει ότι $e \lessdot_E e'$, και ότι ο συναγωνισμός μπορεί να αντιστραφεί. Τυπικά, αν $E' \lesssim E$ και το e συμβαίναι ακριβώς πριν το e' στο E', τότε η $proc(e')$ δεν ήταν μπλοκαρισμένη πριν την εμφάνιση του e.

2.9 Dynamic Partial Order Reduction

Πριν εξηγήσουμε τον DPOR αλγόριθμο είναι σημαντικό να όρισουμε τα επαρκή σύνολα (sufficient sets).

Definition 2.3. (Επαρκές Σύνολο) Ένα σύνολο από μεταβάσεις είναι επαρκές σύνολο στην κατάσταση s αν κάθε σχετική κατάσταση που είναι προσβάσιμη μέσω μιας δυνατής μεταβάσης από το s είναι προσβάσιμη και από το s μέσω τουλάχιστον μίας μετάβασης στο επαρκές σύνολο. Μια αναζήτηση χρειάαζεται να εξερευνήσεις μόνο μεταβάσεις που ανήκουν στο επαρκές σύνολο από το s επείδή όλα τα σχετικά states θα εξακολουθήσουν να είναιν προσβάσιμα. Το σύνολο που περιέχει όλα τα ενεργά threads είναι τετριμένα επαρκές στο s, αλλά μικρότερα επαρκή σύνολα επιτρέπουν μεγαλύτερο περιορισμό του χώρου καταστάσεων.

Πολλές τεχνικές έχουν προταθεί για την υλοποίηση ενός DPOR αλγορίθμου. Αυτό που έχουν όλες οι τεχνικές κοινό είναι η παρακάτω μορφή.
1.

Algorithm 1: General form of DPOR

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

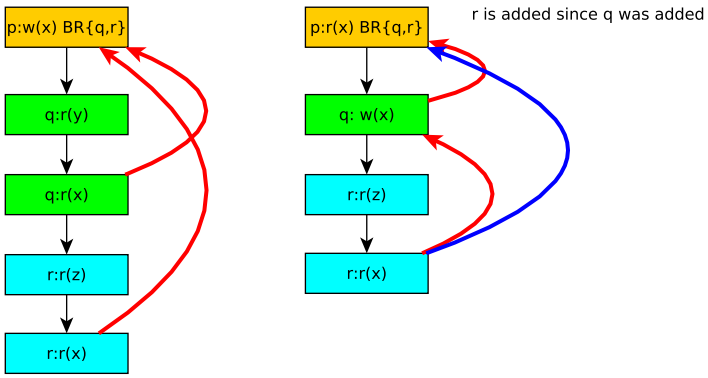
όπου το $final(E)$ αντιπροσωπεύει την κατάσταση που θα φτάσουμε ότι η ακολουθία E εκτελεστεί.
Ο αλγόριμος περιγράφει μια DFS αναζήτηση στο χώρο καταστάσεων όλων των πιθανών interleavings. Όπως μπορούμε να υποθέσουμε το πιο σημαντικό κομμάτι του αλγορίθμου είναι ο υπολογισμός του συνόλου T.
Μια προφανής ιδιότητα που πρέπει να ισχύει είναι ότι Sufficient_set($final(E)$) $\subseteq$ $enabled(E)$.
Διαισθητικά τα $enabled(s)$ αντιπροσωπεύουνν τα νήματα εκείνα που δεν είναι μπλοκαρισμένα και η εκτέλεση τους δεν έχει ολοκληρωθεί.
Στη βιβλιογραφία πολλά είδη επαρκών συνόλων μπορούν να βρεθούν [Gode96]. Σε αυτή τη διπλωματική εστιάζουμε στα επίμονα σύνολα (persistent sets) και στα πηγαία σύνολα (source sets).

2.10 Επίμονα Σύνολα - Persitent Sets

Definition 2.4. (Επίμονα Σύνολα - Persistent Sets) Έστω s ένα state, και έτσω $W \subseteq E(s)$ ένα σύνολο από ακολουθίες εκτέλεσης από το s.Ένα σύνολο T μεταβάσεων είναι επίμονο σύνολο για το W μετά το s αν για κάθε πρόθεμα του w από κάποιες ακολουθίες στο W, οι οποίες δεν πειρέχουν εμφανίσεις από μεταβάσεις στο T, έχουμε ότι $E \vdash t \diamondsuit w$ για κάθε $t \in T$.

Ο παραπάνω ορισμό μπορεί να περιγραφέι και ώς εξής: Αν $t \in T$ και δεν υπάρχει άλλο thread t' το οποίο μπορεί να εκτελεσθεί μέχρι μια εντολή η οποία βρίσκεται σε συναγωνισμό με το t, τότε το t' ανήκει στο επίμονο σύνολο.
Πρέπει να σημειώσουμε ότι αυτός ορισμός προτείναι και ένα τρόπο κατασκευής του.
Στο Σχήμα 2.3 παρουσιάζονται δύο διαφορετικά παραδείγματα κατασκευής επίμονων συνόλων. Συμβολίζουμε το επίμονο σύνολο από διακλαδώσεις που μπορεί να πάρει η εκτέλεση με BR. Στο π□ώτο παράδειγμα, έστω ένα concurrent program που περιέχει 3

threads p, q, και r. Το thread p μεταβάλλει την τιμή της μεταβλητής(writer) και τα άλλα (q και r) απλά διαβάζουν την τιμή αυτών των μεταβλητών (readers). Έστω $p.q.q.r.r$ ένα interleaving. Σύμφωνα με τον ορισμό των επίμονων συνόλων, τα q και r συναγωνίζονται με το p, επομένως, τα q και r πρέπει να ανήκουν και αυτά στο επίμονο σύνολο. Στο Σχήμα 2.3 παρατηρούμε ότι και το r και το q thread προστίθενται στο επίμονο σύνολο της πρώτης εντολής καθώς και τα δύο έχουν conflict με το write operation. Στο δεύτερο παράδειγμα, έστω τα p και r είναι αναγνώστες και q είναι εγγραφέας. Παρατηρούμε ότι και το r και το q προστίθενται. Παρολ' αυτά δεν υπάρχει conflict μεταξύ των p και r καθώς και τα δύο είναι αναγνώστες της μεταβλητής x. Ο λόγος είναι ότι το thread r έχει conflict που προκύπτει από την έγγραφή που θα εκτελέσει το q.



Σχήμα 2.3: Construction of persistent sets

2.11 Πηγαία Σύνολα - Source Sets

Πριν δώσουμε των ορισμό των πηγαίων συνόλων (source sets), πρέπει να δώσουμε κάποιους άλλους χρήσιμους ορισμούς.

Definition 2.5. ($dom(E)$) Το σύνολο των μεταβάσεων που συμβαίνει σε μια δρομολόγηση του E.

Definition 2.6. (Initials μιας ακολουθίας εκτέλεσης $E.w$, $I_{[E]}(w)$) Για μια ακολουθία εκτέλεσής $E.w$, έστω ότι το $I_{[E]}(w)$ συμβολίζει το σύνολων των διεργασιών που προκαλούν ένα γεγονός e στο $dom_{[E]}(w)$ τα οποία δεν έχουν "happens-before" πρόγονο στο $dom_{[E]}(w)$. Πιο τυπικά, $p \in I_{[E]}(w)$ αν $p \in w$ και δεν υπάρχει άλλο γεγονός $e \in dom_{[E]}(w)$ για το οποίο ισχύει $e \rightarrow_{E.w} next_{[E]}(p)$.

Χαλαρώνοντας τον ορισμό για τα Initials παίρνουμε τον ορισμό των Weak Initials, WI.

Definition 2.7. (Weak Initials μετά από μια ακολουθία εκτέλεσης $E.w$, $WI_{[E]}(w)$) Για μια ακολουθία εκτέλεσης $E.w$, έστω ότι $WI_{[E]}(w)$ είναι η ένωση των $I_{[E]}(w)$ με το σύνολο των διεργασιών που μπορούν να εκτελέσουν μια εντολή δηλαδή αν $p \in enabled(s_{[E]})$.

Το νόημα αυτών των δύο εννοιών για μια ακολουία $E.w$ είναι το εξής:

- $p \in I_{[E]}(w)$ ανν υπάρχει μια ακολουθία w' τέτοια ώστε $E.w \simeq E.p.w'$, και
- $p \in WI_{[E]}(w)$ ανν υπάρχει μια ακολουθία w' και v τέτοια ώστε $E.w.v \simeq E.p.w'$.

Definition 2.8. (Πηγαία Σύνολα - Source Sets) Έστω E μια ακολουθία εκτέλεσης, και έστω W ένα σύνολο από ακολουθίες, τέτοιο ώστε $E.w$ είναι μια ακολουθία για κάθε $w \in W$. Το σύνολο T των διεργασιών είναι πηγαίο σύνολο του W μετά το E αν για κάθε $w \in W$ έχουμε ότι $WI_{[E]}(w) \cap P = \emptyset$.

Το πηγαίο σύνολο είναι ένα σύνολο από threads το οποίο μας εγγυάται ότι όλος ο χώρος καταστάσεων θα εξερευνηθεί. Είναι σημαντικό να παρατηρήσουμε ότι τα πηγαία σύνολα δε συνδέονται με συναγωνισμό μεταξύ γεγονότων Αυτό που υπονοεί ο ορισμός είναι ότι πηγαίο σύνολο πρέπει να θεωρείαι κάθε σύνολο από νήματα που περιέχει αυτά τα threads που μπορούν να καλύψουν όλο το χώρο καταστάσεων Στην πραγματικότητα ο ορισμός δίνει μια ιδιότητα για τα επαρκή σύνολα.

2.12 Κοιμώμενα Σύνολα - Sleep Sets

Μια ακόμα τεχνική συμπληρωματική με τα επίμονα και τα πηγαία σύνολα που στοχεύει να μειώσει την εξερεύνηση περιττών interleavings είναι η τεχνική των κοιμώμενων συνόλων (Sleep Sets) Τα sleep sets εμποδίζουν μια είδη χρησιμοποιημένη μετάβαση να ξαναχρησιμοποιηθεί μέχρι η εξερεύνηση να φτάσει σε μια μετάβαση που έχει εξάρτηση από τη μετάβαση που βρίσκεται ήδη στο sleep set. Ας υποθέσουμε ότι η εξερεύνηση χρησιμοποιεί μια μετάβαση t από ένα state s, προσθέτει στο backtrack t, και στη συνέχεια εξευρευνά το t_0 από s. Μέχρι η αναζήτηση συναντήσει μια μετάβαση που είναι που είναι σε συναγωνισμό με το t, κανένα state δεν είναι προσβάσιμο μέσω του t_0 που δεν θα ήταν ήδη προσβάσιμες μέσω του t από το s. Επομένως, το t "κοιμάται" μέχρι μια μετάβαση με συναγωνισμό εξερευνηθεί.
Ένα μικρό παράδειγμα είναι αυτό που ακολουθεί: Έστω ένα concurrent πρόγραμμα που αποτελείται από έναν εγγραφέα (w1) και δύο αναγνώστες (r1,r2). Έστω w1 <0.0>: w(x) r1 <0.1>: (local operations), r(x) και r2 <0.2>: (local operations), r(x).
Το trace που παίρνουμε σαν αποτέλεσμα φαίνεται στον Κώδικα 2.5. Ας σημειωθεί ότι όποτε μια διεργασία εκτελεί μια εντολή η οποία βρίσκεται σε συναγωνισμό με μια εντολή απο άλλη διεργασία η οποία "κοιμάται", ξυπνάει. Όπως μπορούμε να συμπεράνουμε από την εκτέλεση του DPOR το interleaving που ξεκίνησε από το r2 μπλόκαρε καθώς θα οδηγούσε σε ένα interleaving το οποίο έχουμε ήδη εξερευνήσει. Ας σημειώσουμε ότι λόγω του ότι το r1 δεν "ξυπνάει" καθώς η πρώτη μετάβαση (local operations) δεν έχει conflict με κάποια άλλη μετάβαση.
Αποδεικνύεται [Gode96] ότι τα sleep sets μπορούν να μπλοκάρουν κάθε περιττή μετάβαση και επομένως μονο κατάλληλα interleavings θα εξερευνηθούν μέχρι το τέλος.

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

Listing 2.5: Sleep set example

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


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

Σχήμα 2.4: Program with non-minimal persistent sets

2.13 Σύγκριση Persistent Sets με Source Sets

Ας σημειωθεί ότι ο ορισμός των source sets είναι πολύ πιο χαλαρός από τον αντίστοιχ των persistent sets. Αυτή η χαλάρωση επιτρέπει στα source sets να είναι πολύ πιο αποδοτικά από τα persistent sets. Στο Σχήμα 2.4 ένα παράδειγμαα δίνεται που δείχνει τη διαφορά των δύο.
Από το παράδειγμα, είναι σαφές ότι ο λόγος που τα source sets είναι καλύτερα από τα persistent sets είναι το γεγονό ότι τα minimum source μπορούν να εξαλείψουν τελείως sleep set blocked traces δηλαδή traces που θα μπλοκάρονταν από τα sleep sets. Ένας αλγόριθμος που θα μπορούσε να υπολογίσει minimal source sets θα ήταν βέλτιστος [Abdu14], και επομένως δε θα εξερευνούσε ποτέ περιττά interleavings.
Είναι προφανές ότι μια μόνο μετάβαση δεν μπορει να είναι source set. Για παράδειγμα, το σύνολο $\{p_1\}$ δεν περιέχει initials της εκτέλεσης $q_1.q_2.p_1.r_1.r_2$, καθώς τα q2 και p1 εκτελούν conflicting accesses. Από την άλλη μεριά, κάθε υποσύνολο που περιέχει δύο enabled μεταβάσεις είναι source set. Για να το δούμε αυτό ας διαλέξουμε το $\{p_1, q_1\}$ ως source set. Προφανώς, το $\{p1, q1\}$ περιέχει ένα initial για κάθε εκτέλεση που ξεκινά είτε με το p_1 ή το q_1. Κάθε εκτέλεση που ξεκινά με το r_1 είναι ισοδύναμη με την εκτέλεση που παίρνουμε αν θέσουμε ώς πρώτο βήμα είτε το p_1 ή το q_1:

- Αν το q_1 συμβεί πριν το r_2, τότε το q_1 είναι initial, καθώς δεν έχει conflict με κάποια άλλη μετάβαση.
- Αν το q_1 συμβαίνει πριν το r_2, τότε το p_1 είναι ανεξάρτητο από όλα τα βήματα και p_1 είναι initial. Ισχυριζόμαστε ότι το $\{p_1, q_1\}$ δεν μπορεί να είναι persistent set. Ο λόγος είναι ότι η εκτέλεση $\{r_1.r_2\}$ δεν περιέχει καμία μετάβαση που να είναι στο persistent set, αλλά το βήμα είναι εξαρτόμενο από το q_1.

Με άλλα λόγια, τα persistent sets έχουν τη δυσάρεστη ιδιότητα όταν προσθέτουμε μία διεργασία στο persistent set να προστίθενται επιπλέον διεργασίες.
Συνεχίζοντας τη σύγκρισή μεταξύ source sets και persistent sets, πρέπει να σημειώσουμε δύο ιδιότητες.

- Κάθε persistent set είναι source set.
- Κάθε one-process source είναι persistent set.

2.14 Περιορισμένη Αναζήτηση και Περιορισμός ως προς την Προτίμηση

Η περιορισμένη αναζήτηση - bounded search εξερευνά μόνο εκτελέσεις που δεν ξεπερνούν κάποιο όριο [Coon13, Thom16]. Το όριο μπορεί να είναι μια ιδιότητα της ακολουθίας.

Μια συνάρτηση υπολογισμού του ορίου $B_v(E)$ δίνει μια τιμή σε μια ακολουθία E. Η συνάρτηση υπολογισμού B_v και το όριο c είναι είσοδη στην περιορισμένη αναζήτηση. Προφανώς το bounded search μπορεί να μην έχει πρόσβαση σε όλα τα προσβάσιμα states της unbounded αναζήτησης. Αντίθετα επισκέπτεται μόνο states που είναι προσβάσιμα μέσα στο bound.
Ένας αλγόριθμος που περιγράφει το bounded search θα ήταν ο ακόλουθος:

Algorithm 2: Bounded-DPOR

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

Η μόνη διαφορά μεταξύ unbounded και bounded εκδοχής είναι το if-statement στη γραμμή 4 που επιτρέπει την εξερεύνηση interleaving μόνο αν το bound δεν έχει ξεπεραστεί.
Το μόνο που μένει είναι ο ορισμός μια κατάλληλης B_v που υπολογίζει την τιμή που ο bounded-DPOR προσπαθεί να κρατήσει κάτω από ένα όριο καθώς και το sufficient set.
Σε αυτή τη διπλωματική μας απασχολεί το preemption-bounded search.
Το preemption-bounded search περιορίζει τον αριθμό των preemptive context switches που συμβαίνουν σε μια εκτέλεση [Musu07]. Το preemption bound δίνεται αναδρομικά από τον ακόλουθο τύπο.

Definition 2.9. (Preemption bound)
$P_b(\emptyset) = 0$

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

Ο ορισμός περιγράφει τί είναι ένα preemptive context switch. Preemptive context switch έχουμε όταν ένα thread που έτρεχε σταματήσει για να εκτελεστεί ένα άλλο

2.15 Φραγμένα ως προς την Προτίμηση Επίμονα Σύνολα - Preemption Bounded Persistent Sets

Ένα σύνολο που έχει προταθεί ως sufficient για preemption bounded search είναι το preemption bounded persistent set [Coon13].
Μια σημαντική παρατήρηση που πρέπει να κάνουμε είναι ότι ένα thread που μπλοκάρει είν τερματίζει δε θα αυξήσει το count.

Definition 2.10. ($ext(s,t)$) Given a state $s = final(E)$ and a transition $t \in enabled(s)$, $ext(s,t)$ returns the unique sequence of transitions β from s such that

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

Στη συνέχει ορίζουμε τα preemption bounded persistent sets. Συμβολίζουμε με $A_G(P_b, c)$ Το generic bounded state space την bound function με P_b και το bound με c. Το $last(a)$ συμβολίζει το τελευταίο βήμα εκτέλεσης της ακολουθίας a.

Definition 2.11. (Preemption bounded persistent set)

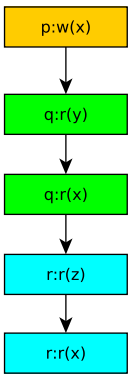
Ένα σύνολο $T \subseteq \mathcal{T}$ από δυνατές μεταβάσεις στο state $s = final(e)$ είναι preemption-bound persistent στο s ανν για κάθε μη κενή ακολουθία a απο μεταβάσεις από s στο $A_G(P_b, c)$ τέτοιο ώστε $\forall i \in dom(a), a_i \notin T$ για κάθε $t \in T$,

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

Όταν έχουμε να κάνουμε με preemption bounded DPOR είναι χρήσιμο να εισάγουμε την έννοια του block μιας ακολουθίας.

Definition 2.12. (Block μια ακολουθίας εκτέλεσης) Block μιας ακολουθίας εκτέλεσης είναι κάθε κομματι της ακολουθίας που αποτελείται από εκτελέσεις του ίδου thread.

Στο Σχήμα 2.5 υπάρχουν τρία blocks. Το πρώτο block είναι το κίτρινο, το δεύτερο το πράσινο και το τρίτο το μπλέ.



Σχήμα 2.5: Example of blocks

Έστω P ένα persistent set. Preemption bounded persistent set είναι ένα σύνολο που περιέχει όλα τα $p \in P$ με την προσθήκε όλων των threads που θα προστιθεντο στο block που θα δημιουργούνταν όταν το p δρομολογούνταν. Αυτές οι διακλαδώσεις ονομάζοντια συντηρητικές (conservative) και ο στόχος τους είναι να επιτρέψουν την ανακάλυψη interleaving που δεν ξεπερνούν το bound. Σημειώστε ότι μια διακλάδωση μπορεί να είναι και conservative και non-conservative. Τα preemption bounded persistent sets επεκτείνουν τα persistent set βαζοντας όλα τα threads που θα δημιουργήσουν καινουρια block στη συνέχεια.

Κεφάλαιο 3

DPOR χωρίς Περιορισμούς

Σε αυτό το κεφάλαιο συζητόνται οι αλγόριθμοι Classic-DPOR και Source-DPOR. Αυτοί οι αλγόριθμοι διαφέρουν μεταξύ τους σε ό,τι αφορά τα επαρκή σύνολα που χρησιμοποιούν προκειμένου να εξερευνήσουν το χώρο καταστάσεων.Ο DPOR βασίζεται στα persistent sets ενώ ο Source-DPOR στα source sets.

3.1 Source-DPOR

Σε αυτή την ενότητα παρουσιάζεται ο Source-DPOR [Abdu14] αλγόριθμος. Παρόλο που ο Source-DPOR είναι μεταγενέστερος του Classic-DPOR, όπως φαίνεται και από το όνομα, επιλέγουμε να τον παρουσιάσουμε πρώτο καθώς όλοι οι αλγόιρθμοι που παρουσιάζονται σε αυτή τη διπλωματική βασίζονται στον Source-DPOR.

Algorithm 3: Source-DPOR

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

Αρχικά μια αυθαίρετη ενεργή διεργασία που δεν κοιμάται προστίθεται στο $backtrack(E)$. Σε κάθε βήμα ο αλγόριθμος αποτελείται από δύο διακριτά βήματα. Κατα τη διάρκεια του πρώτου βήματος ο εντοπισμός συναγωνισμών (races) λαμβάνει χώρα. Ο αλγόριθμος επιλέγει μια διεργασία p η οποία μπορεί να εκτελέσει το επόμενό της βήμα, δηλαδή, $p \in enabled(s_{[E]})$, και την προσθέτει στο τέλος του υπο εξερεύνηση trace E. Στη συνέχεια ο αλγόριθμος βρίσκει κάθε γεγονός e το οποίο περιέχεται ήδη στο υπο εξερεύνηση trace ($e \in dom(E)$) και το αντιμεταθέτει με το επόμενο βήμα του p.
Αυτό συμβαίνει προκειμένου να εξερευνηθεί μια ακολουθία εκτέλεσης για την οποία το p συμβαίνει πριν το e. Έχουμε ότι η ακολουθία αποτελείται από τα:

- $E' \equiv pre(E, e)$: την υπακολουθία E' που αποτελείται από γεγονότα δρομολογημένα πριν το e στο E.

- $u \equiv notdep(e, E).p$: την συνένωση u από όλα τα γεγονότα που είναι δρομολογημένα μετά το e στο E αλλά είναι ανεξάρτητα από τα e και p.

- $proc(e)$ που είναι το id της διεργασιά που προκάλεσε το e.

Στη συνέχεια ο αλγόριθμος εξετάζει αν υπάρει κάποια διεργασία που ανήκει στο $I_{[E']}(u)$ και είναι ήδη στο $backtrack(E')$. Αν όχι, τότε μια διεργσία στο $I_{[E']}(u)$ προστίθεται στο backtrack.
Κατά τη φάση της εξερεύνησης, η εξερεύνηση ξεκινά από το $E.p$. Το σημαντικό κομμάτι είναι ο υπολογισμός του sleep set του επόμενου βήματος καθώς κάποιες διεργασίες ίσως πρέπει να ξυπνήσουν. Αν το επόμενο βήμα μιας διεργασίας συγκρούεται με το $next(p)$ τότε αυτή η διεργασία πρέπει να ξυπνήσει. Σαν αποτέλεσμα το sleep set αποτελείται από τις είδη κοιμώμενες διεργασίες των οποίων το επόμενο βήμα είναι ανεξάρτητο από το $next(p)$, δηλαδή ισχύει, $Sleep' := \{q \in Sleep \mid E \models p \diamondsuit q\}$ Μετά το τέλος της εξερεύνησης του $E.p$, το p προστίθεται στο sleep set επειδή θέλουμε να αποτρέψουμε την εξερεύνηση ενός ισοδύναμου trace.

3.2 Classic-DPOR

Επειδή ο Source-DPOR χρησιμποιεί διανυσματικά ρολόγια για να παρακολουθεί τα γεγονότα θα χρησιμοποιήσουμε τον DPOR με χρήση Clock Vectors [Flan05] και μια παραλλαγή του που δίνεται στον Αλγόριθμο 4.

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

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

Ένα clock vector $C(p)$ συντηρείται καθόλη την εκτέλεση του αλγορίθμου για κάθε διεργασία p. Έτσι, το $C(p_i) = \langle c_1, c_2, ..., c_m \rangle$ αντιπροσωπεύει το clock vector μιας διεργασίας p_i. όπου το c_j είναι το index της τελευταίας μετάβασης της διεργασίας p_j για την οποία ισχύει ότι $c_j \rightarrow_s p_i$. Διαισθητικά, το clock vector μιας διεργασίας p_j δίνει πληροφορία

στη διεργασία p_j σχετικά με την εκτέλεση των βημάτων των άλλων διεργασιών που συμβαίνουν πριν από την εκτέλεση των βημάτων της p_j. Τα clock vectors βασίζονται στον αλγόριθμο του Lamport [Lamp78] και περισσότερες λεπτομέριες σχετικά με τη χρήση τους μπορούν να βρεθούν εδώ [Flan05]. Αντιπροσωπεύουμε την αρχική κατάσταση όλων των vector clocks ως $\perp = \langle 0, ..., 0 \rangle$. Με $C(p)(proc(E_i))$ αντιπροσωπεύουμε την τιμή των clock vector μίας διεργασίας p για τη διεργασία στο i-οστό βήμα του E.
Η αρχική κατάσταση του Classic-DPOR είναι μια κενή ακολουθία από γεγονότα και όλα τα vector clocks τίθενται σε $\perp$.
Ο Classic-DPOR αποτελείται από δύο φάσεις. Η πρώτη φάση είναι η ανίχνευση races. Κατα τη διάρκεια αυτής της φάσης η επόμενη μετάβαση από όλες τις διεργασίες p λαμβάνεται υπόψιν. Για κάθε μια μετάβαση $next(s, p)$ (που μπορεί να είναι ενεργή ή όχι στο s), υπολογίζεται η τελευταία εξαρτόμενη με αυτή μετάβαση i στο E. Ο υπολογισμός λαβμάνει χώρα στη γραμμή 4. Αν υπάρχει μια τέτοια μετάβαση i, ίσως υπάρχει και ένα race condition ή μία εξάρτηση μεταξύ i και $next(s, p)$, και επομένως, ίσως θα π□έπει να εισάγουμε ένα "backtracking point" στη θέση του state $pre(S, i)$, δηλαδή στο state ακριβώς πριν την ετκέλεση της μετάβασης i. Αν p είναι ενεργή διεργασία προστίθεται σαν backtrack point. Αλλιώς το σύνολο όλων των ενεργών μεταβάσεων γίνεται backtracked.
Κατά τη διάρκει της φάσης της εξερεύνησης η διεργασία p που ανήκει στο $enabled(s)$ προστίθεται σαν backtrack point όπως και στον Source-DPOR. Στη συνέχεια τα vector clocks ενημερώνονται σύμφωνα με τον αλγόριθμο του Lamport.

3.3 Σύγκριση μεταξύ Classic-DPOR και Source-DPOR

Όπως μπορούμε εύκολα να διαπιστώσουμε και οι δύο αλγόριθμοι αποτελούνται από τις δύο ίδιες διακριτές φάσεις το race detection και τη φάση της εξερεύνησης. Επιπλέον και οι δύο αλγόριθμοι βασίζονται στα διανυσματικά ρολόγια, παρόλο που η χρήση τους απλώς υπονοείται στον Source-DPOR στις γραμμές 6 και 8 του Αλγορίθμου 3.
Η βασική διαφορά έγκειται στη φάση της ανίχνευσης race. Κατά τη διάρκεια αυτής της φάσης στον Classic-DPOR όλες οι διεργασίες p λαμβάνονται υποψιν πριν δρομολογηθούν και έτσι πολλές από αυτές θα γίνουν backtrackede. Από την άλλη, στον Source-DPOR μόνο η τελευταία διεργασία που δρομολογήθηκε λαμβάνεται υπόψιν. Επιπλέον οι δύο αλγόριθμοι διαφέρουν στον τρόπο που αντιμετωπίζουν την περίπτωση της απουσίας ενεργών διεργασιών κατα τη διάρκει προσθήκης backtrack point. Σε αυτή την περίπτωση ο Classic-DPOR προσθέτει όλες της ενεργές διεργασίες ενώ καμία διεργασία δεν προστείθεται από τον Source-DPOR. Συνολικά, ο Source-DPOR εκμεταλλέυεται τα Clock vectors περισσότερο από τον Classic-DPOR.

3.4 Συνδιάζοντας Classic-DPOR και Source-DPOR

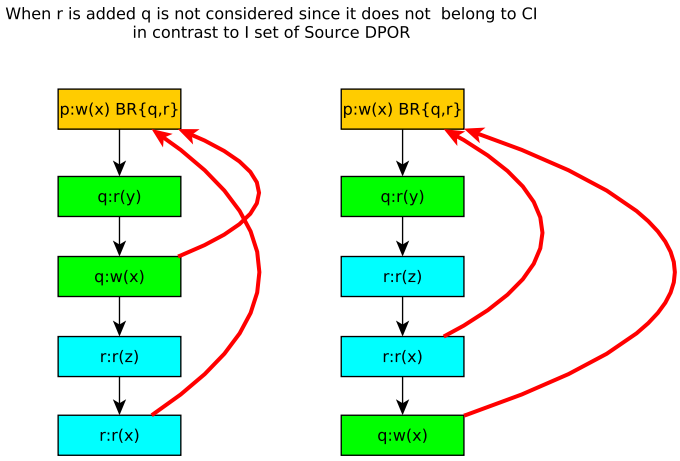
Προκειμένου να εκμεταλλευτούμε την υποδομή του Nidhugg ο Αλγόριθμος 5 θα χρησιμποιηθεί. Σαν αποτέλεσμα δεν υπάρχει ανάγκη να προσθέσουμε όλα τα διαθέσεμα threads όταν το p δεν είναι ενεργό. Αν δεν έχει βρεθεί κατάλληλος υποψήφιος τότε ένας υποψήφιος που προτείνεται από τον αντίστοιχ Source-DPOR αλγόριθμο θα προστεθεί. Αυτός ο τρόπος υπολογισμού persistent sets θεωρείται πιο πολύπλοκος και επομένω κακή επιλογή [Gode05]. Παρολ' αυτά η χρήση source sets είναι πιο κοντά σε αυτή την προσέγγιση.
Ο Αλγόριθμος 5 διαφέρει από τον Source-DPOR (Αλγ. 3) στον υπολογισμό των initials. Συγκεκριμένα ένα υποσύνολο των initials (CI) που συμβαίνει πριν από το p χρησιμοποιείται. Διαισθητικά στην περίπτωση του εγγραφέα και των δύο αναγνωστών και οι δύο αναγνώστες θα προστεθούν στο backtrack καθώς ο πρώτος αναγνώστης δε συμβαίνει

Algorithm 5: Nidhugg-DPOR

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

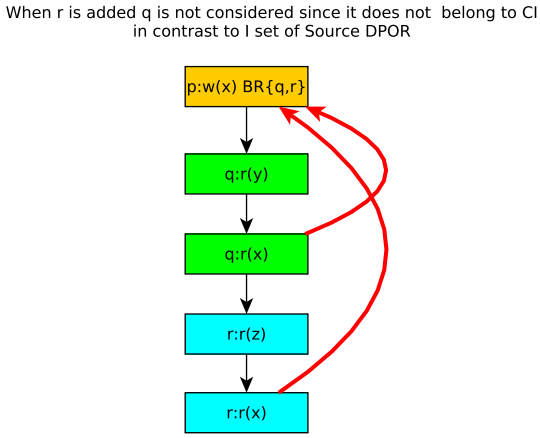
πριν από τον δεύτερο. Για να γενικεύσουμε την ιδέα: Το Nidhugg δεν μας επιτρέπει να δημιουργήσουμε διακλαδώσει για το $last(E)$, κατα τη διάρκεια της φάσης της δρομολόγησης. Οι διακλαδώσεις προστίθενται αργότερα στον Source-DPOR (Αλγ. 3). Όταν ένα race εξετάζεται συνήθως μόνο ένα thread που προκαλέι το race Θα προστεθεί καθώς το CI περιέχει αυτό το thread μόνο.
Θα δώσουμε μια διαισθητική απόδειξη σχετικά με την ορθότητα του Nidhugg-DPOR στον να υπολογίζει persistent set και θα εξηγήσουμε γιατί όταν ο αλγόριθμος τελειώσει ένα persistent θα έχει υπολογιστεί σε κάθε βήμα. Το ενδιαφέρον κομμάτι είναι να δείξουμε ότι για καθε διακλάδωση που είναι μια εντολή εγγραφής όλες οι διεργασίες που συγκρούονται με την εντολή και μπορούν να συμβούν ταυτόχρονα με αυτή θα προστεθούν στο σύνολο των διακλαδώσεων.
Ας υποθέσουμε δύο διεργασίες που βρίσκονται σε race στο $last(S)$.

- Περίπτωση 1: τουλάχιστον μία διεργασία περιέχει μια εντολή εγγραφής. Ξέρουμε ότι οι δύο διεργασίες θα πρέπει να αντιστραφούν σε κάποιο σημείο. Από τη στιγμή που ο Nidhugg-DPOR αγνοεί τα weak initials θα προσθέσει και τις δύο διεργασίες. Στο Figure 3.1 παρατηρούμε ότι οι διεργασίες q και r αντιστρέφονται. Στον Source-DPOR μόνο μία από τις διεργασίες πρέπει να χρησιμποιηθοεί για να γίνει διακλάδωση καθώς μοιράζονται τα ίδια initials. Παρολ' αυτά στον Nidhugg-DPOR αυτό δεν ισχύει καθώς το CI δεν περιέχει βήματα από τις άλλες διεργασίες.



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

- Case 2: και οι δύο διεργασίες είναι αναγνώστες. Αφού δεν υπολογίζουμε το I αλλά το CI η πρώτη ανάγνωση δε θα ληφθεί υπόψιν καθώς δε συμβαίνει πριν από την δεύτερη ανάγνωση και σαν αποτέλεσμα και οι δύο διεργασίες θα προστεθούν στο $backtrack$. Στο Figure 3.2 παρατηρούμε ότι στον υπολογισμό του CI όταν το race μεταξύ των p και r εντοπίζεται η q αγνοείται και, επομένως, το r θα προστεθεί για να γίνει διακλάδωση.



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

Είναι σαφές ότι καμία διεργασίε που δεν ανήκει στο $backtrack(S)$ δεν έχει ανταγωνισμό με το τις διεργασίες στο $backtrack(S)$.

Κεφάλαιο 4

Τεχνικές περιορισμού για DPOR

Σε αυτό το κεφάλαιο Τεχνικές Περιοσμού (Bounding Techniques) για τον DPOR συζητόνται καθώς και προκλήσεις που δημιουργούνται από αυτές τις τεχνικές. Οι προκλήσεις που πρέπει να αντιμετωπίσουμε για να περιορίσουμε τον DPOR έχουν ήδη συζητηθεί.

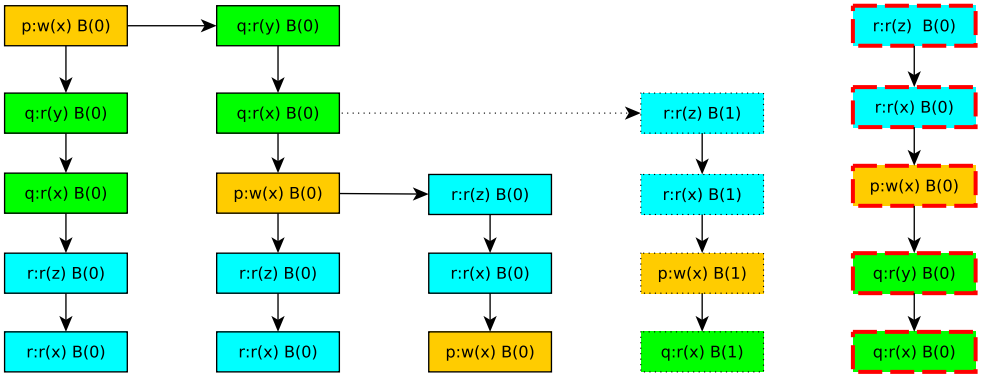
4.1 Naive-BPOR

Οι πρώτη τεχνική περιορισμού που παρουσιάζεται είναι ο Naive-BPOR (Αλγόριθμος 6). Ο σκοπός αυτού του αλγορίθμου είναι να μπλοκάρει traces που ξεπερνούν κάποιο όριο.

Algorithm 6: Naive-BPOR

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

Ο Αλγόριθμος 6 είναι σχεδόν πανομοιότυπος με τον Source-DPOR(Algorithm 3). Η μόνη διαφορά είναι η προσθήκη μια συνθήκης που συνδέεται με τη δρομολόγηση διεργασίων. Όταν ένα βήμα μιας διεργασίας p προστίθεται στο E με αποτέλεσμα το trace $E.p$: $B_v(E.p) > b$ τότε η διεργασία δεν επιτρέπεται να δρομολογηθεί. Ο Αλγόριθμος δεν είναι sound δηλαδή, δεν εξετάζει όλα τα traces τα οποία ανήκουν έχουν bound count μικρότερο από μία τιμή.
Υποθέτουμε το παραάδειμα του ενός εγγραφέα και των δύο αναγνωστών για bound $b = 0$. Ένα παράδειγμα αναζήτησης για bound $b = 0$ δίνεται στο Figure 4.1.



Σχήμα 4.1: Naive-BPOR for bound=0

Όπως μπορούμε να παρατηρήσουμε υπάρχουν 4 traces τα οποία δεν ξεπερνούν το όριο. Αυτά είναι: $p.q.q.r.r$, $q.q.p.r.r$, $r.r.p.q.q$, $q.q.r.r.p$. Παρολ' αυτά ο Naive-BPOR δεν είναι σσε θέση να τα εξερευνήσει όλα; $r.r.p.q.q$ δεν εξερευνάται. Όπως δείξαμε στη σύγκριση μεταξύ persistent και source sets, το r δεν θα γίνει ποτέ το πρώτο γεγονός στο trace καθώς αυτό θα οδηγούσε σε sleep set blocked trace. Η διακλάδωση που θα οδηγούσε στο ισοδύναμο trace με το $r.r.p.q.q$ απορρίπτεται καθώς θα έχει μεγαλύτερο bound count από το ζητούμενο. Αξιοπρόσεκτο είναι ότι ένας Naive-BPOR που βασιζόταν σε persistent sets θα εξερευνούσε ένα μεγαλύτερο state-space και θα εξερευνούσε και το ζητούμενο trace $r.r.p.q.q$.

4.2 BPOR

Το επόμενο βήμα είναι να υλοποιήσουμε έναν preemption bounded Αλγόριθμο (BPOR) [Coon13]. Αυτός ο Αλγόριθμος θα βασίζεται στα persistent sets. Καθώς έχουμε ήδη ένα ορθά υλοποιημένο αλγόριθμο που χρησιμοποιεί persistent sets στον Nidhugg-DPOR μπορούμε εύκολα να υλοποιήσουμε έναν BPOR αλγόριθμο. Η καινοτμία του BPOR είναι η εισαγωγή συντηρητικών διακλαδώσεων. Αυτές είναι διακλαδώσει που εισάγονται προκειμένου να εγγυηθούν την εξερεύνηση όλου του state space. Είναι αρκετά συνηθισμένο ένα trace να ξεπερνάει το bound limit αλλά ένα ισοδύναμό του να μην το ξεπερνάει. Οι συντηρητικές διακλαδώσεις χρησιμοποιούνται γι αυτό το σκοπό.
Για να κάνουμε την ιδέα πιο καθαρή εισάγουμε την έννοια Trace Block.

Definition 4.1. (Trace block) Για ένα trace T η ακολουθία B από συνεχόμενα γεγονότα είναι ένα Trace Block ανν όλα τα γεγονότα πραγματοποιούντια από την ίδια διεργασία δηλαδή, όλα τα γεγονότα έχουν το ίδιο thread id.

Η ιδέα πίσω από τις συντηρητικές διακλαδώσεις είναι αρκετά απλή. Μια συντηρητική διακλάδωση προστίθεται στην αρχή του αντίστοιχου Block όταν μια διακλάδωση δημιουργείται. Συνήθως τα ταυτόχρονα γεγονότα συμβαίνουν μέσα σε ένα block. Έτσι όταν επιλέγεται μια εναλλακτική διακλάδωση το preemption count θα αυξηθεί. Αν η διακλάδωση αυτή είχε εξερευνηθεί από την αρχή του block το preemption count δε θα είχε αυξηθεί. Ο BPOR παρουσιάζεται με λεπτομέρια στον Αλγόριθμο 7. Ο BPOR διαφέρει από τον Classic-BPOR στον διπλά εμφολευμένο βρόγχο που εισάγεται στη γραμμή 3. Ο εσωτερικός βρόγχος εισάγεται προκειμένου ο BPOR να υπακούει στον ορισμό των preemption-bounded persistent sets που εισάγαμε στο Κεφάλαιο 2 Επιπλέον, όπως ισχύει και στον Naive-BPOR, η εξερεύνηση σταματάει όταν το trace ξεπερνάει το όριο που εμείς έχουμε θέσει.

Algorithm 7: BPOR

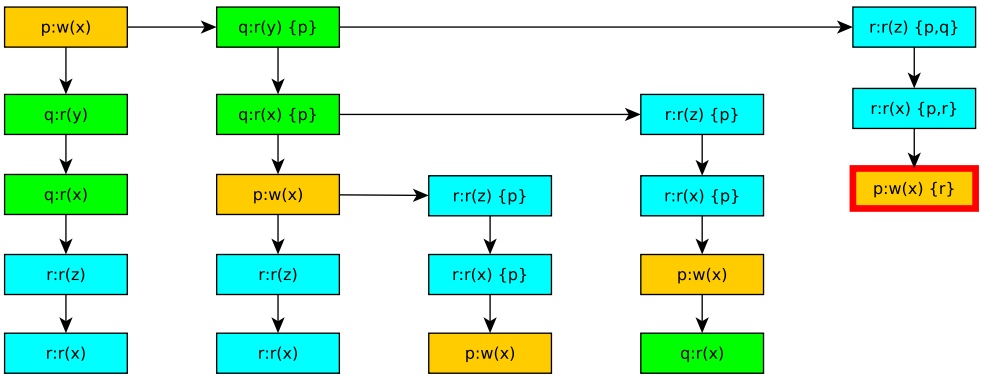
```
1  Function Explore(E)
2      let s := last(E);
3      for all process p do
4          for all process q ≠ p do
5              if ∃i = max({i ∈ dom(E) | E_i is dependent and may be co-enabled with
                next(s, p) and E_i.tid = q}) then
6                  if p ∈ enabled(pre(E, i))) then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

4.3 Nidhugg-BPOR

Προκειμένου να εκμεταλλευτούμε την υποδομή του Nidhuggs, μια παραλλαγή του αλγορίθμου χρησιμοποιείται. Ο αλγόριθμος παρουσιάζεται εδώ 8.
Παρατηρούμε ότι τα persistent sets χρησιμοποιείται και για τα conservative και για τα non-conservative branches.
Μια σημαντική πρόκληση δημιουργείται όταν ο DPOR Αλγόριθμος χρησιμοποιείται μαζί με τα sleep sets. Αυτό προκύπται από το γεγονός ότι οι οι συντηρητικές διακλαδώσεις δεν δημιουργούνται προκειμένουν αν επιλύσουν races. Στον Αλγόριθμο των sleep set παρατηρούμε ότι αν ακολουθήσουμε την ίδια στρατηγική με τις μη συντηρητικές διακλαδώσεις πολλά traces θα είχαν μπλοκαριστεί.
Ας πάρουμε και πάλι το παράδειγμα του εγγραφέα και των δύο αναγνωστών, όπως φαίνεται στο Figure 4.2.

Algorithm 8: Nidhugg-BPOR

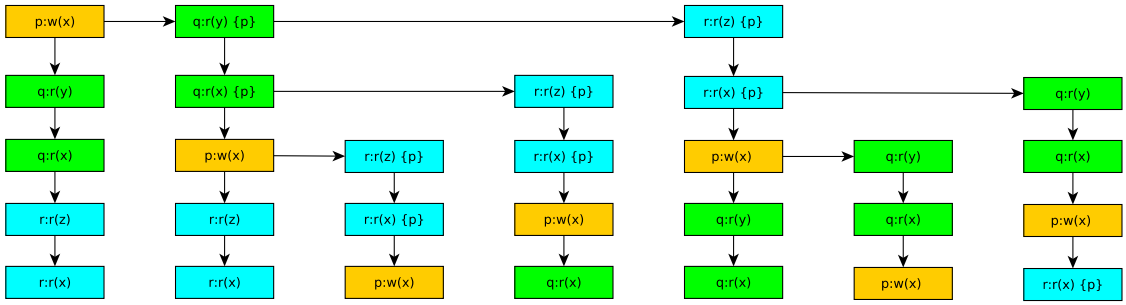
```
1  Explore(⟨⟩,∅,b);
2  Function Explore(E,Sleep,b)
3      if ∃p ∈ ((enabled(s_[E])\Sleep) and B_v(E.p) <= b then
4          backtrack(E) := p ;
5          while ∃p ∈ (backtrack(E)\Sleep and B_v(E.p) <= b do
6              foreach e ∈ dom(E) such that e ≲_{E.p} next_[E](p) do
7                  let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



Σχήμα 4.2: Usage of non-conservative branches

Παρατηρούμε ότι το τελευταίο trace θα μπολοκαριστεί από τα sleep set ενώ θα έπρεπε να εξεταστεί. Ο Αλγόριθμος δεν γνωρίζει ότι η διεργασία r πρέπει να αφαιρεθεί από το sleep set καθώς δεν υπάρχει και δεν πρόκειται να υπάρξει ποτέ conflict με την πρώτη εντολή της διεργασίας καθώς αυτή σχετίζεται με μια μη μοιραζόμενη μεταβλητή. Προκει-

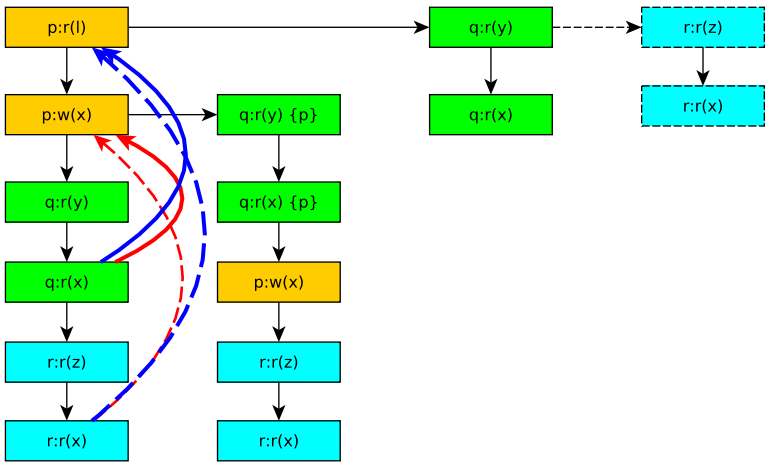
μένου να αντιμετωπίσουμε αυτό το πρόβλημα επιλέχθηκε τα conservative branch να μην προστίθενται στο sleep set. Παρολ' αυτά πρέπει να είμαστε σε θέση να καταγράφουμε όλες τις διακλαδώσεις που προστίθενται σε ένα συγκεκριμένο σημείο προκειμένου καμία διακλάδωση να μην προστεθεί δύο φορές. Χωρίς αυτό το σύνολο που κάνει αυτή την καταγραφή η εκτέλεση δε θα τελείωνε. Η λύση είναι η εισαγωγή του συντηρητικού συνόλου (conservative sets) όπου καταγράφεται η προσθήκη των διακλαδώσεων.
Διαισθητικά, ο αλγόριθμος είναι ίδιος με τον Nidhugg-DPOR με την προσθήκη των συντηρητικών διακλαδώσεων. Ο Αλγόριθμος βασίζεται στην των conservative sets. Τα υπόλοιπα προβλήματα που αντιμετωπίστηκαν δίνονται στην ενότηταα σχετικά με την υλοποίηση.



Σχήμα 4.3: Example of BPOR execution

4.4 Source-BPOR

Έχοντας συζητήσει τον BPOR Αλγόριθμο και τον Nidhugg-BPOR, το επόμενο βήμα είναι να προσπαθήσουμε να συνδιάσουμε τον αλγόριθμο με τα source sets. Η πρώτη παρατήρηση που πρέπει να κάνουμε είναι ότι τα source sets και επομένως ο αλγόριθμος που τα δημιουργεί δεν είναι κατάλληλα στην προσθήκη συντηρητικών διακλαδώσεων. Μια γρήγορη επεξήγησ δίνεται στο επόμενο παράδειγμα με τον έναν εγγραφέα και τους δύο αναγνώστες. Ας υποθέσουμε έναν αλγόριθμο που χρησιμποιεί source sets για την προσθήκη συντηρητικών διακλαδώσεων. Το αποτέλεσμα φαίνεται στο Figure 4.4.



Σχήμα 4.4: Following source sets for conservative branches

Είναι σαφές ότι κάποια traces δεν εξερευνόνται. Συγκεκριμένα, το trace που ξεκινά με τη r θα απορριφθεί. Ο λόγος είναι ότι μοιράζεται τα ίδια initials με το r1 ακόμα και στην

αρχή αυτού του block. Σαν αποτέλεσμα ο Αλγόριθμος πρέπει να δημιουργεί persistent sets όταν συντηρητικές διακλαδώσεις προστίθενται.
Έχοντας υπόψιν την προηγούμενη παρατήρηση ο Αλγόριθμος Source-BPOR δίνεται εδώ 9.

Algorithm 9: Source-BPOR

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

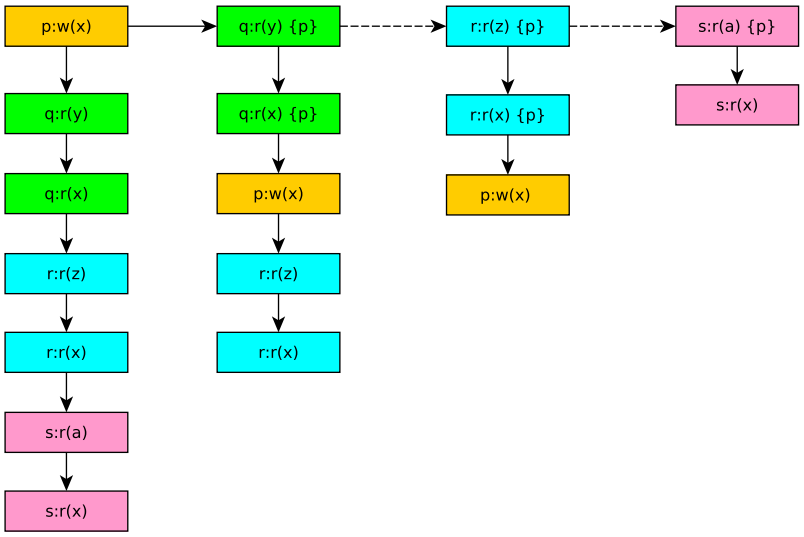
Παρατηρούμε ότι ο Αλγόριθμος 9 χρησιμοποιεί Source-DPOR (Alg. 3) για τη μη συντηρητικές διακλαδώσεις και τον BPOR (Alg. 8) για τις συντηρητικές διακλαδώσεις.

4.5 Προκλήσεις από την προσθήκη Συντηρητικών Διακλαδώσεων

4.5.1 Η Αύξηση του Χώρου Καταστάσεων

Στην προηγούμενη ενότηταη είδαμε ότι τα conservative sets δεν μπορούν να χρησιμποιήσουν τα sleep sets. Αυτό συμβαίνει επειδή αυτές οι διακλαδώσεις δνε προκύπτουν από conflicts και σαν αποτέλεσμα είναι αδύνατο να "wake up" άλλες διεργασίες των οποίων τα επόμενα βήματα είναι τοπικές λειτουργίες. Το πρόβλημα γίνεται ακόμα πιο πολύπλοκο όταν σκεφτούμε ότι όταν προσθέτουμε συντηρητικές διακλαδώσεις ο αλγόριθμος "ξενχά" τί έχει λάβει χώρα προηγουμένως. Αυτή η έλλειψη μνήμης οδηγεί σε μια έκρηξη του χώρου καταστάσεων. Αυτή η έκρηξη είναι ακόμα πιο έντονη και από την εξερεύνηση όλου του state space Προκειμένου να εξηγήσουμε καλύτερα, ένα παράδειγμα που περιλαμβάνει έναν εγγραφέα και τρεις αναγνώστες δίνεται στο Figure 4.5. Συγκεκριμένα, η διεργασία εγγραφέας που γράφει τη μεταβλητή x ενώ οι διεργασίες αναγνώστες δια-

βάζουν αυτή τη μεταβλητή αφού διαβάσουν μια άλλη μοναδική για την κάθε διεργασία μεταβλητή που μπορεί να θεωρηθεί τοπική λειτουργία.

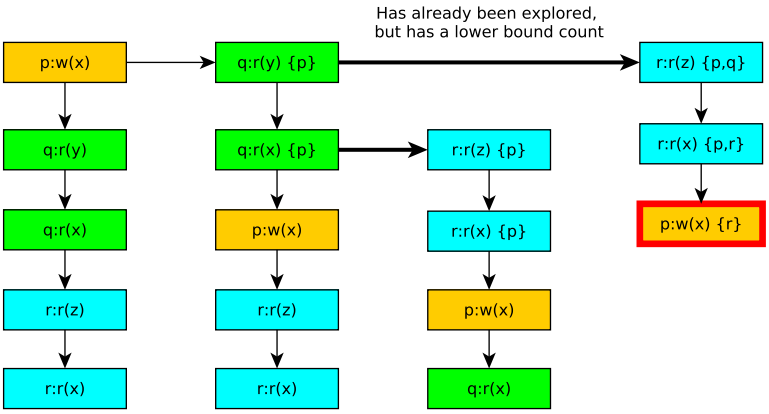


Σχήμα 4.5: writer-3-readers explosion

Όπως μπορούμε να συμπεράνουμε από το παράδειγμα οι εξερευνημένες καταστάσεις είναι περισσότερες από τις αναμενόμενες. Ο αναμενόμενος αριθμός θα έπρεπε να είναι 8 αλλά στην πραγματικότητα είναι $4!$. Παρατηρούμε ότι κάθε εγγραφέας γίνεται baccktracked στην πρώτη εντολή καθώς έχει conflict με το $w(x)$. Αυτές οι διακλαδώσιες θα πρέπει να γίνουν blocked απο την unbounded search. Παρολ' αυτά, αυτές οι διακλαδώσεις είναι συντηρητικές. Υπάρχουν extra traces όπως το $r2.r1.w$ trace που ελέγχει ήδη το εξερευνημένο $r1.r2.w$ trace. Ο Αλγόριθμος δεν είναι σε θέση να γνωρίζει όταν προσθέτει συντηρητικές διακλαδώσεις αν ισοδύναμα traces έχουν ήδη εξερευνηθεί. Η κατάσταση γίνεται χειρότερ όταν ακόμα περισσότεροι εγγραφείς διαβάζουν την ίδια μοιραζόμενη μεταβλητή καθώς αν θέσουμε ένα μεγάλο όριο τότε ο αριθμός των traces προσεγγίζει την τιμή $N!$ όπου N ο συνολικός αριθμόες των διεργασιών.

4.5.2 Η Κατάργηση των Sleep Sets

Τα αποτελέσματα των διαφόρων bounding αλγορίθμων υπονοεί ότι ο αριθμός τον sleep set blocked traces είναι πολύ μικρός σε σχέση με τον αριθμό των εξερευνημένων traces. Αυτό προκύπτει από τα conservative branches. Έχουμε ήδη δείξει όταν μία διεργασία προστείθεται και ως συντηρητική διακλάδωση και ως μη συντηρητική διακλάδωση η συντηρητική φύση της διακλάδωσης υπερισχύει. Traces που θα θεωρούνταν περιττά στην ubounded εκδοχή του αλγορίθμου δεν είναι περιττά στην bounded εκδοχή καθώς οι μη συντηρητικές διακλαδώσεις μπορεί να έχουν απορριφθεί.
Ένα ακόμα "πρόβλημα" με τα sleep sets είναι ότι "ευνοούν" τις διακλαδώσεις που αυξάνουν το bound count ενώ μπλοκάρουν traces με μικρότερο bound count. Στο Figure 4.6 φαίνεται πως μια διακλάδωση που δεν οδηγεί σε αύξηση του bound count απορρίπτεται.



Σχήμα 4.6: Sleep set contradiction

Αυτή η συμπεριφορά είνα αρκετά λογική αν αναλογιστούμε την depth-first φύση του αλγορίθμου. Αποτέλεσμα είναι να επιλέγει πρώτα διακλαδώσει που βρίσκονται χαμηλότερα στο trace όπου το bound count είναι μεγαλύτερο καθώς περισσότερα preemptive switches έχουν συμβεί. Ο σκοπός των sleep sets είναι να μπλοκάρουν traces, δηλαδή traces που έχουν ήδη εξερευνηθεί. Έτσι traces με μικρότερο bound count απορρίπτονται Μια μέθοδος που θα εξερευνούσε το state space με έναν breadth first τρόπο ίσως δε θα ήταν εφικτή καθώς θα απαιτούσε πολλή μνήμη από ημιτελή traces.

4.5.3 Το Αντιστάθμισμα μεταξύ Επίδοσης και Ακρίβειας

Από την προηγούμενη συζήτηση είναι σαφέ ότι όλοι οι bounding algorithms έχουν να αντιμετωπίσουν ένα tradeoff. Κάποια αλγόριθμοι είναι γρηγορότεροι καθώς εξερευνούν ένα μικ□ό κομμάτι του state space, χωρίς να καλύπτουν όλο το state space μέσα στο bound (Αλγόριθμος 6), ενώ άλλοι εξερευνούν όλο το state space (Αλγόριθμος 8) μέσα στο bound αλλά απαιτούν πιο πολλή ώρα.

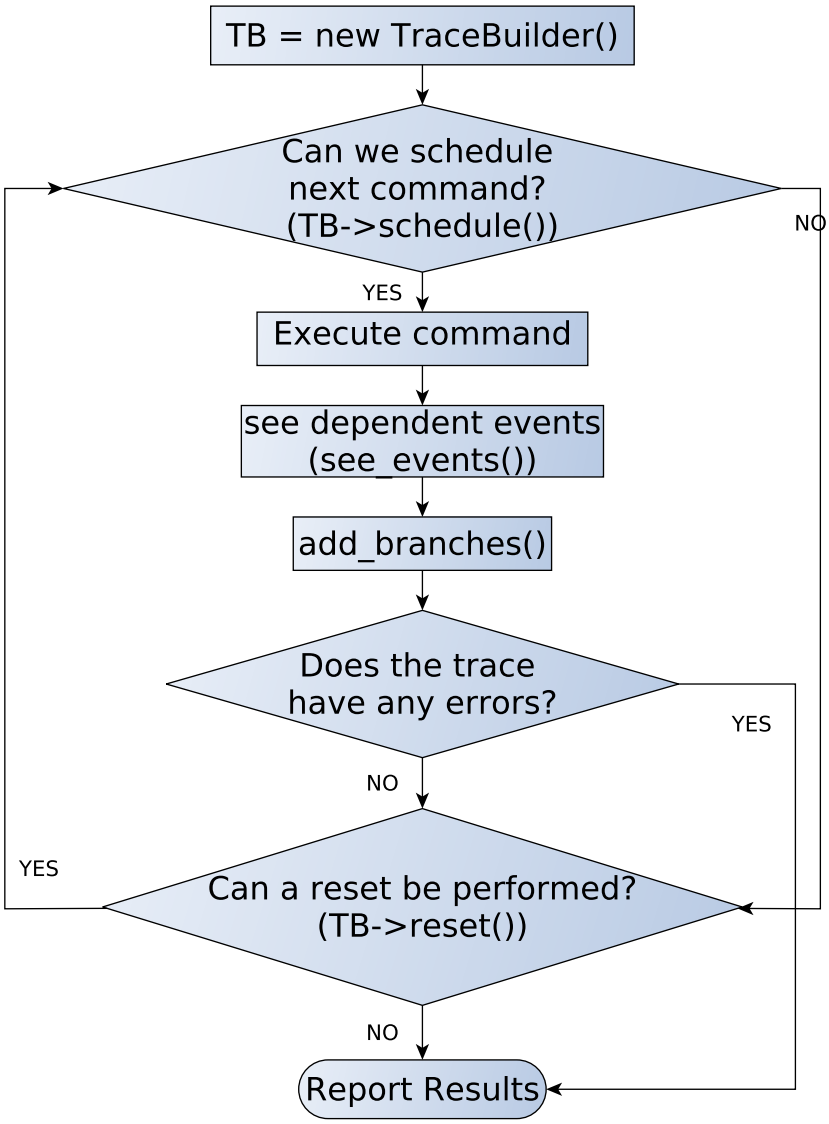
Κεφάλαιο 5

Υλοποιήσεις Αλγορίθμων

Το Nidhugg είναι εργαλείο εντοπισμού σφαλμάτων που στοχεύει στην ανεύρεση σφαλμάτων που προκύπτουν από το μη ντετερμινισμό του δρομολογιτή και σε λάθη που προκύπτουν από τα μοντέλα χαλαρής μνήμης. Δουλεύει στο επίπεδο της ενδιαμέσης αναπαράστασης του LLVM, που σημαίνει ότι μπορεί να χρησιμοποιηθεί για προγράμματα που είναι γραμμένα σε C και C++ ή γλώσσες που μπορούν να γίνουν compile σε LLVM και υλοποιούν τον ταυτοχρονισμό χρησιμοποιώντας τη βιβλιοθήκη pthreads.
Τη στιγμή που γραφόταν η παρούσα διπλωματική το Nidhugg υποστήριζε τα μοντέλα SC, TSO, PSO and POWER. Από την άλλη το Nidhugg δεν μπορεί να χειριστεί τον μη ντετερμινισμό των δεδομένων γι αυτό τα threads θα πρέπει να είναι ντετερμινιστικά όταν τρέχουν σε απομόνωση.

5.1 Η Ροή Προγράμματος του Nidhugg

Το Nidhugg δουλεέυι στο επίπεδο της ενδιάμεσης αναπαράστασης του LLVM (IR). Προκειμένου το Nidhugg να βρέι ένα σφάλμα δημιουργεί ένα διερμηνέα LLVM assembly. Στη συνέχεια δρομολογεί και εκτελεί διαφορετικα traces μέχρι ένα σφάλμα να βρεθεί όπως η παραβίαση ενός assertion. Τα traces παίζουν ένα σημαντικό ρόλο στο Nidhugg καθώς αναπαριστούν τα διαφορετικά schedulings. Αυτά τα traces αντιπροσωπεύονται ως vectors από Events objects. Το Event object συντηρεί όλη την χρήσιμή πληροφορία σχετικά με ένα event όπως το pid του thread που εκτελέστηκε. Οι διακλαδώσεις που προκαλούν την εξερεύνηση διαφορετικών δρομολογήσεων αποθηκεύονται επίσης στο Event object. Οι δρομολογήσεις ρυθμίζονται από το Tracebuilder object με τη σειρά του καθορίζεται από το memory model που χρησιμοποιείται. Ο Tracebuilder είναι επίσης υπεύθυνος για τον ανεύρεση races μεταξύ των διαφόρων threads και τις προσβάσεις στη μνήμη.
Η εκτέλεση ακολουθεί γενικά τη ροή που παρουσιάζεται στο Figure 5.1. Όπως βλέπουμε και στο διάγραμμα ροή ο TraceBuilder προσπαθεί να δρομολογήσει καινούρια γεγονότα με τη χρήση της schedule(). Μετά τη δρομολόγηση τα γεγονότα εκτελούνται και ενημερώνονται τα διανυσματικά ρολόγια. Στη συνέχεια ελέγχεται αν κάποιο γεγονός είναι εξαρτόμενο από κάποιο άλλο, δηλαδή, προσπελαύνει κάποια τοποθεσία της μνήμης που ανήκει και στο άλλο. Μετά από αυτό το Niddhugg προσπαθεί να προσθέσει διακλαδώσεις και να ελέγξει αν υπάρχει κάποιο σφάλμα. Στην περίπτωση σφάλματος ο χρήστης ενημερώνεται και η διαδικασία τερματίζει. Αξίζει να σημειωθεί ότι υπάρχει επιλογή η εξερεύνηση να συνεχίσει ώστε να βρεθούν περισσότερα λάθη. Αν στο τέλος μιας δρομολόγησης δεν έχει βρεθεί κάποιο λάθος ο TraceBuilder κάνει reset στην πιο κοντινή διακλάδωση και συνεχίζει την εξερεύνηση. Αν δεν υπάρχουν άλλες διαθέσιμες διακλαδώσεις, η εξερεύνηση τερματίζει.
Σε ό,τι αφορά του αλγορίθμους που υλοποιήθηκαν είναι ξεκάθαραο ότι το πιο σημαντικό κομμάτι του διαγράμματος ροής είναι ο εντοπισμός εξαρτήσεων.



Σχήμα 5.1: Nidhugg's Flow Chart

5.2 Προσθήκη διακλαδώσεων στο Nidhugg

Μόλις δρομολογηθεί μια εντολή που πιθανώς να προκαλέσει σφάλμα το διάνυσμα see_accesses δημιουργείται και περιλαμβάνει όλες τις προσπελάσεις που λαμβανουν χώρα στη ίδια θέση στη μνήμη και καλείται η διαδικασία `see_events()`.

Algorithm 10: see_events()

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

Είναι σαφές από τον αλγόριθμο ότι ο σκοπός της συνάρτησης είναι εξαιρέσει όλες εκείνες

τις προσβάσεις στη μνήμη που δεν είναι σε race με την συγκεκριμένη πρόσβαση δηλαδή εξαρτήσεις που δεν μπορούν να αναπαρασταθούν σαν traces στα οποία δεν υπάρχουν άλλα γεγονότα που να συμβαίνουν μεταξύ τους. Είναι σημαντικό να ξεκαθαρίσουμε ότι αυτό δε σημαίνει ότι τα γεγονότα δεν μπορούν να είναι ταυτόχρονα σε κάποια άλλη δρομολόγηση. Τα γεγονότα που επιβίωσαν της διαδικασίς προστίθενται σε ένα πίνακα ο οποίος χρησιμοποιείται από την συνάρτηση `add_branch()`.
Μία άλλη λειτουργία της `see_events()` είναι η ενημέρωση των vector clocks. Μετά το τέλος της ρουτίνας για δύο γεγονότα που είναι ταυτόχρονα τα διανυσματικά ρολόγια θα έχουν ενημερωθεί ώστε να φαίνεται η σχέση-πριν (happens-before relation).
Η συνάρτηση `add_branch()` φαίνεται στον αλγόριθμο 11 είναι η πιο σημαντική στην υποδομή του Nidhugg.

Algorithm 11: add_branch()

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

Διαισθητικά, η add_branch κάνει τα ακόλουθα: Ξεκινώντας από ένα γεγονός που έχει conflict με το πιο πρόσφατα δρομολογημένο γεγονός, ξεκινά να διανύει το διάνυσμα που αντιπροσωπεύει το E μέχρι το κοντινότερο στο τέλος του trace, thread βρεθεί (αν αυτό είναι δυνατό). Στην πραγματικότητα αυτό που συμβαίνει είναι ο υπολογισμός των I (initials). Όπως φαίνεται και από τον αλγόριθμο αν έχει ήδη τοποθετηθεί μια κατάλληλη διακλάδωση ή μια κατάλληλη διακλάδωση είναι στο sleep set η διαδικασία τερματίζει.
Το αποτέλεσμα της `add_branch()` είναι ο υπολογισμός ενός source set.

5.3 Υλοποίηση του Nidhugg-DPOR

Η υλοποίηση των persistent sets βασίζεται στην ήδη υλοποιημένη υποδομή των vector clocks. Συγκεκριμένα η `include()` συνάρτηση των vector clocks μας επιτρέπει να καθορίσουμε αν υπάρχουν σχέσεις $i \to p$. Ο αλγόριθμος που αυτή υλοποιεί `includes()` δίνεται στον Αλγόριθμο 12. Ακολουθώντας τον συμβολισμό του Source-DPOR το $\langle e, i \rangle$ είναι ένα ήδη δρομολογημένο γεγονός και είναι το i-οστό βήμα της διεργασίας e και το p είναι το πιο πρόσφατα backtracked γεγονότ της διεργασίας p.
Για να υπολογίσουμε το CI χρειάζεται απλλά να προλάβουμε την `add\_branch()` από το να απορρίψει διακλαδώσει λόγω thread που ανήκουν στο I και όχι στο CI. Η αλλαγές που έγιναν δίνονται παρακάτω στον Αλγόριθμο 13.
Στην περίπτωση που το E είναι κενό μπορούμε να χρησιμοποιήσουμε έναν υποψήφιο που υποδείχτηκε από το I.

Algorithm 12: includes() routine

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

Algorithm 13: add_branch() for persistent sets

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

5.4 Υλοποίηση του Naive-BPOR

Το πρώτο που πρέπει να υλοποιήσουμε για κάθε bounding technique είναι ένα μετρητής του bound count. Στην περίπτωση μας ένα μετρητής που θα μας δίνει πόσα preemptive switces έχουν συμβεί στο trace.
Πρέπει να είμαστε σε θέση να καταλάβουμε πότε ένα thread είναι ενεργό. Αυτήν την πληροφορία μπορούμε να την πάρουμε από παλαιότερες τιμές του ίδιου του bound count. Δοθέντων δύο διαδοχικών γεγονότων a, b, αν $a.bound\_count < b.bound\_count$ τότε a ήταν διαθέσιμο.
Ο ψευδοκώδικας δίνεται στον Αλγόριθμο 14.

Algorithm 14: Should we increase the bound count?

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

Προκειμένου να μπορέσουμε να επιβεβαιώσουμε την επιβεβαίωση της σωστής μέτρηση του bound count, το debug print του Nidhugg τροποποιήθηκε κατάλληλα. Έτσι ο μετρητής πρέπει να δουλεύει όπως φαίνεται στο Listing 5.1.

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

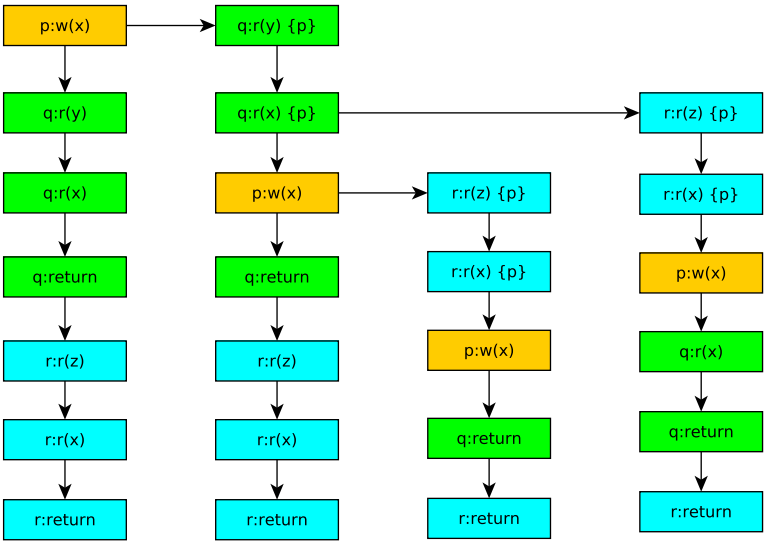
Listing 5.1: Example of bound counter

Το Nidhugg αναμένει τα traces να μπλοκάρονται μόνο λόγω sleep sets. Έτσι έπρεπε να γίνουν οι κατάλληλες αλλαγές ώστε να γνωρίζουμε το λόγο που δρομολόγηση σταμάτησε. Συγκεκριμένα προστέθηκαν flags στην συνάρτηση `schedule()` γι αυτό το σκοπό.
Το αποτέλεσμα μετά την εκτέλεση του προγράμματος φαίνεται στο Listing 5.2.

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

Listing 5.2: Naive-BPOR output

Μια αλλαγή που πρέπει να κάνουμε είναι να αλλάξουμε τη προτεραιότητα με την οποία δρομολογούνται τα threads. Το Nidhugg δρομολογεί τα threads δίνοντας προτεραιότητα στο παλαίοτερο. Έτσι μόλις ένα παλιό thread ξυπνήσει θα δρομολογηθεί αμέσως. Με αποτέλεσμα να αυξηθεί το preemption count. Επομένως πρεπει να αλλάξουμε την προτεραίοτητα δρομολόγησης ώστε να προηγείται πάντα το thread που έτρεχε προηγούμενως ώστε να αποφύγουμε αυξήσεις του bound count.



Σχήμα 5.2: Execution without the scheduling optimization

Στο Figure 5.2 δίνεται ένα παράδειγμα με δύο threads να διαβάζουν μια μεταβλητή και ενα να τη γράφει. Όπως φαίνεται και στην εικόνα όταν το p ξυπνά λόγω του conflict τότε δρομολογείται αμέσως αυξάνοντας το bound count.

5.5 Υλοποίηση του Nidhugg-BPOR

Όπως είδαμε και προηγούμενως οι βασικές τροποποιήσεις πρέπει να γίνουν στις συναρτήσεις see_events και add_branch. Ο ψευδοκώδικας για το see_events φαίνεται στον Αλγόριθμο 15.
Κατα τη διάρκει του see_events προσθέτουμε διακλαδώσεις και στην αρχή του block όπου αυτό είναι δυνατό. Για να το κάνουμε αυτό πρέπει να ελέγξουμε αν το thread που προσθέτουμε ήταν ενεργό στην αρχή του block.

Algorithm 15: see_events() for BPOR

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

Στην περίπτωση που η `add_branch()` εκτελείται απευθείας χρησιμοποιούμε την `add_conservative_branch()` που λειτουργεί ώς το "συντηρητικό" κομμάτι της

`see_events()`. Ο ψευδοκώδικας της συνάρτησης δίνεται στον Αλγόριθμο 16. Η `add_branch()` έχει ήδη περιγραφεί στον Αλγόριθμο 13.

Algorithm 16: try_to_add_conservative_branches()

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

Κατά την `add_branch()` προσθέτουμε διακλαδώσεις στα κατάλληλα σημεία χρησιμοποιώντας την λογική των persistent sets όπως είδαμε σε προηγούμεν ενότητα. Όπως αναφέραμε προστίθενται δύο ειδών διακλαδώσεις. Κατά την αναζήτηση μας στο sleep set θα πρέπει να αναζητούμε μόνο τα μη συντηρητικά threads. Στην περίπτωση που μια διακλάδωση του ίδιου thread προστίθεται και σαν συντηρητική και σαν μη συντηρητική τότε η συντηρητική υπερισχύει.

5.6 Υλοποίηση του Source-BPOR

Η υλοποίηση ξανά βασίζεται σε τροποποιησεις στις συναρτήσεις `add_branch()` καθώς και στην `see_events()`. Μπορούμε να παρατηρήσουμε ότι ο αλγόριθμος θα διαλέξει τον ίδιο υποψήφιο που θα διάλεγε ο Source-DPOR όταν θα προσέθετε μη συντηρητικές διακλαδώσεις και τον ίδιο υποψήφιο με τον Nidhugg-BPOR όταν θα διαλέξει συντηρητικές διακλαδώσεις. Ο ψευδοκόωδικάς για την `add_branch()` δίνεται στον Αλγόριθμο 17.

Algorithm 17: add_branch() routine for Source-BPOR

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

Κεφάλαιο 6

Αξιολόγηση των Υλοποιηθέντων Αλγορίθμων

Σε αυτό το κεφάλαιο η επίδοση κάθε αλγορίθμου που υλοποιήθηκε αξιολογείται. Αρχικά μελετάμε την επίδοσου του Nidhugg-DPOR προκειμένου να δείξουμε ότι και στην υλοποίηση μας πράγματι η επίδοσήτ του διαφέρει από αυτή του Source-DPOR. Η αξιολόγηση χωρίζεται σε δύο μέρη. Στο πρώοτ μέρος χρησιμοποιούνται μικ□ά συνθετικά προγράμματα ενώ στο δεύτερο χρησιμοποιείται πραγματικό λογισμικό. Τα συνθετικά τεστ μπορούν να βρεθούν στο παράρτημα.

6.1 Συνθετικά Προγράμματα

Τα συνθετικά τεστ έχουν προέλθει από διάφρορες πηγές και δεν είναι ιδιαίτερε πολύπλοκα καθώς θέλουμε απλώς να μας δείξουν τις διαφορές στην επίδοση μεταξύ Source-DPOR και Nidhugg-DPOR.

- writer-N-readers: Σε αυτό το τεστ N threads διαβάζουν (readers) την ίδια global μεταβλητή και ένα thread (writer) γράφει σε αυτή τη μεταβλητή. Είναι σηματνικό να σημειώσουμε ότι πριν την ανάγνωση της μεταβλητής κάποιο local operation του thread λαμβάνει χώρα. Επομένως αναμένουμε σημαντική διαφορά στην επίδοση μεταξύ των source sets και persistent sets.

- Account: Αυτό το τεστ αποτελεί μια προσομοίωση ενός τραπεζικού λογαρισαμού και χρησιμοποιεί locks για κάθε operation που συμβαίνει στον λογαριασμό. Υπάρχουν τρείς δυνατές λειτουργίες: Να αυξήσουμε το απόθεμα του λογαριασμού κατά ένα ποσό. Να κάνουμε ανάληψη μειώνοντας το απόθεμα με ένα συγκεριμένο ποσό. Να ελέγξουμε το αποτέλεσμα check_result (έλεγχος αποτελέσματος) όπου επιβεβαιώνουμε ότι final_balance == initial_balance + deposit − withdraw και μπορεί να συμβεί μόνο αν η κατάθεση και η ανάληψη έχουν ολοκληρωθεί.

- Micro: Σε αυτό το τεστ τρία threads δημιουργούνται και εκτελούν δύο φορές την εντολή x++ δύο φορές. Η συγκεκριμένη εντολή x++ αποτελείται από δύο επιμέρους λειτουργίες μία ανάγωσης της τιμής της μεταβλητής και μία το γράψιμο της μεταβλήτης.

- Last-zero test: Το πρόγραμμα αποτελείται από N+1 threads τα οποία εκτελούν λειτουργίες σε N+1 στοιχεία ενός πίνακα που αρχικά όλα είναι μηδενικά. Σε αυτό το πρόγραμμα το thread 0 αναζητά το μηδενικό στοιχείο του πίνακα με το μεγαλύτερο index ενώ τα άλλα N threads διαβάζουνε από τον πίνακα ένα στοιχείο και ενημερώνουν την τιμή του επόμενου. Η τελική κατάσταση του προγράμματος ορίζεται μοναδικά από τις που θα έχει ο πίνακα. Το Last-zero δεν παράγει περισσότερα tracce από τον DPOR για λόγους που εξηγούνται στη συνέχεια. Μια παραλλαγή του ενδιάμεσου κώδικα μπορεί να δείξει τη διαφορά.

- Indexer.c: Αυτό το benchmark χρησιμοποιεί την compare-and-swap(CAS) primitive instruction για να ελέγξει να ένα entry στον πίνακα είναι 0 και στη συνέχεια θέτει μια καινούρια τιμή σε αυτό.
- Indexermod.c: Σε αυτό το benchmark όλα τα threads διασχίζουν και προσπαθούν να γράψουν έναν πίνακα με αποτέλεσμα να προκύπτουν πολλές συγκρούσεις μεταξύ των threads.

6.2 RCU

Το Read-Copy-Update (RCU) είναι ένα μηχανισμός συγχρονισμού που εφευρέθηκε από τους McKenney and Slingwine [McKe98], και βασίζεται στην αμοιβαία απόκληση πόρων. Προστέθηκε τον πυρήνα του Linux τον Οκτώβριο του 2002 και επέτρεψε σημαντίκη βελτίωση στην ταυτόχρονη ανάγνωση και ενημέρωση. Σε αντίθεση με τα συνήθη locking primitives που διασφαλίζουν αμοιβαίο αποκλεισμό μεταξύ threads ανεξάρτητα με το αν είναι αναγνώστες ή εγγραφείς ή με τη χρήση reader-writer locks που επιτρέπουν ταυτόχρονες αναγνώσεις ή μία εγγραφή, το RCU επιτρέπει ταυτόχρονες αναγνώσεις και μία εγγραφή.
Η τεχνική του DPOR χρησιμοποιήθηκε σαν μια προσέγγιση για ελεγχθεί συστηματικά ο κώδικα του RCU που χρησιμποιείται στο Linux kernel (Tree RCU) κάτω από το memory model του sequential consistency. Η μοντελοποίηση επιτρέπει στο Nidhugg να αναπαράγει σε μερικά δευτερόλεπτα σφάλαμτα που έχουν αναφερθεί κατα καιρούς σχετικά με το RCU [Koko17b].
Το RCU είναι ένα ιδανικό testcase για την αξιολόγησης διαφόρων υλοποιήσεων του DPOR και Bounded DPOR καθώς:

- Είναι ένα πρόγραμμα που χρησιμοποιείται πραγματικά και όχι ένα συνθετικό τεστ synthetic test.
- Ο αριθμός των διαφορετικών δρομολογήσεων είναι αρκετά μεγάλος για να δούμε διαφορές στην επίδοση.
- Προηγούμενη δουλειά [Koko17b] μας επιτρέπει να αξιολογήσουμε την ορθότητα των υλοποιήσεων μας.

6.3 Αξιολόγηση των Μη Φραγμένων Αλγορίθμων

Όπως έγινε σαφές στο κεφάλαιο 3 η υλοποίηση των persistent sets είναι πολύ σημαντική καθώς χρησιμοποιείται σε κάθε bounding technique. Σε αυτή την ενότητα θα δείξουμε τις διαφο□ες στην επίδοση μεταξύ των Source-DPOR και Nidhugg-DPOR τόσο στα συνθετικά tests όσο και στο RCU.

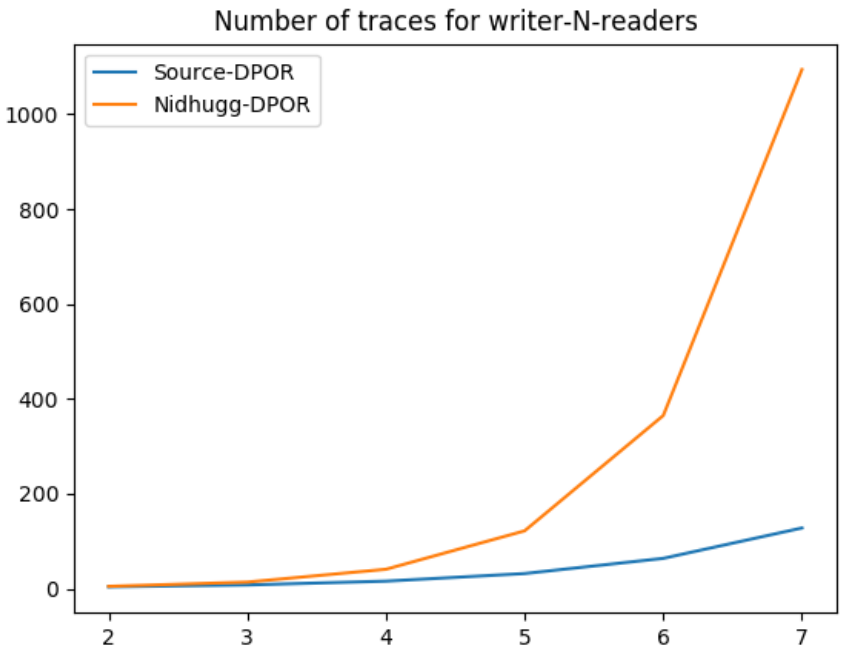
6.3.1 Αξιολόγηση των Persistent Sets στα Συνθετικά Προγράμματα

Τα αποτελέσματα παρουσιάζονται με δύο διαφορετικούς τρόπους. Τα αποτελέσματα για το writer-N-readers testcase δίνονται σε μορφή γραφήματος, στο Figure 6.1, προκειμένου να απεικονίσουμε την κλιμάκωση του χώρου καταστάσεων καθώς και την σημαντική επίδραση που έχει ο source-DPOR. Τα υπόλοιπα αποτελέσματα δίνονται στο Table 6.1 ώστε να μπορέσουμε εύκολα να τα συγκρίνουμε. Ηθελημένα δεν σημειώνουμε τη διάρκεια εκτέλεση καθώς στις περισσότερες περιπτώσεις ο αριθμός των traces είναι πολύ μικρός όπως και ο χρόνος. Όπως ήταν αναμενόνμενο τα συνθετικά testcase έδειξαν ότι ο Source-DPOR πράγματι έχει καλύτερη επίδοση από τον Nidhugg-DPOR. Όπως ήταν

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

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

αναμενόμενο ο Source-DPOR εξερευνά λιγότερα traces από τον Nidhugg-DPOR. Είναι σημαντικό να σημειώσουμ ότι η διαφορά οφείλεται στον αριθμός των sleep set blocked traces τα οποία προκύτπουν από τον DPOR αλγόριθμο και τα οποία δεν συναντάμε στον source DPOR. Αυτή η μείωση δεν είναι σταθερή σε όλες τις περιπτώσεις.



Σχήμα 6.1: writer-N-readers

6.3.2 Αξιολόγηση των Persistent Sets στο RCU

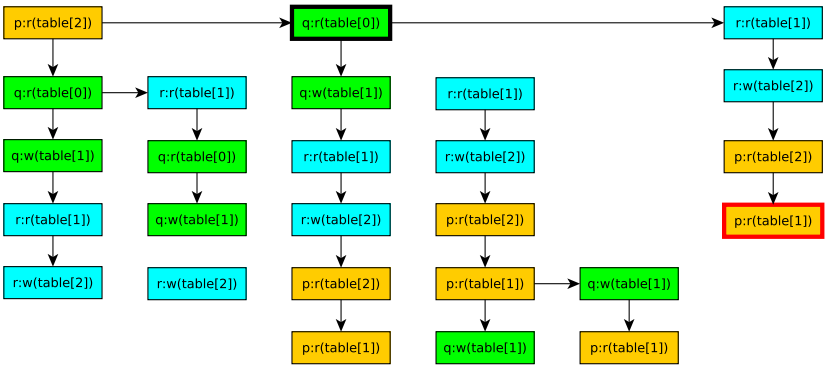
Παρατηρήσμα ότι δεν υπάρχει διαφορά μεταξύ Source sets και persistent sets και επομένως δεν παρουσιάζουμε αποτέλεσματα καθώς αυτά θα συμφωνούν με αυτα του [Koko17b]. Ο λόγος που τα αποτελέσατα του Nidhugg-DPOR και Source-DPOR είναι ίδια οφείτεαι στις λειτουργίες που εκτελούνται και που δεν επιτρέπουν την βελτιστοποίηση του Source-DPOR.

6.4 Σύγκριση με τα αποτελέσματα του Concuerror

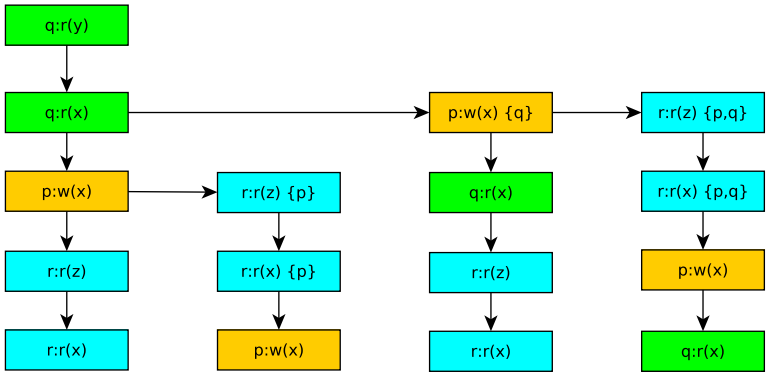
Ο Concuerror είναι ένα εργαλείο που αναπτύχθηκε από την ίδια ερευνητική ομάδα και έχει στόχο να εντοπίζει σφάλματα σε προγράμματα Erlang. Επομένως η σύγκριση της επίδοσής του μπορεί να μας δώσει μια ένδειξη για την ορθότητα των υλοποιήσεών μας. Υπάρχουν περιπτώσεις here Concuerror's Source-DPOR εξερευνά λιγότερα traces από τον Classic-DPOR ενώ στο Nidhugg αυτό δεν ισχύει. Επιπλέον, η υλοποίηση του Nidhugg-DPOR φαίνεται να εξερευνά λιγότερα traces από αυτά που εξερευνόνται από τον Classic-DPOR του Concuerror [Abdu14]. Προσπαθήσαμε να ερμηνεύσουμε αυτή τη συμπεριφορά και συμπεράναμε ότι οι λόγοι που οι τιμές διαφέρουν είναι οι εξής:

- Η υλοποίηση των persistent set: Στον Concuerror ο υπολογισμός των persistent sets είναι πιο χαλαρός από του Nidhugg με αποτέλεσμα το τελευταίο να υπολογίζει μικρότερα persistent sets. Στο Figure 6.2 δίνεται το παράδειγμα του Last-zero testcase του Concuerror ως παράδειγμα υπολογισμού μεγαλύτερου persistent set. Στην πραγματικότητα το q δε θα έπρεπε ποτέ να προστεθεί στο persistent set καθώς δεν έχει conflict με άλλες διεργασίες.

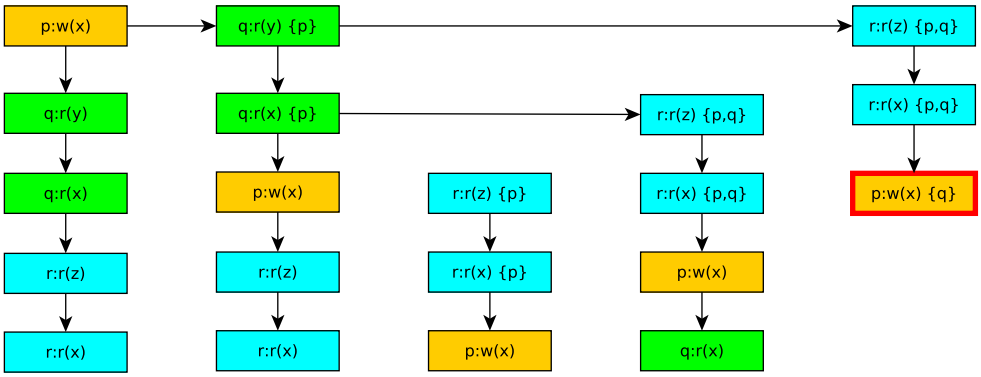
- Ο αριθμός των traces που εξερευνότναι σχετίζεται άμεσα με την δρομολόγηση των γεγονότων: Έστω ένα πρόγραμμα που αποτελείται από δύο διεργασίες που διαβάζουν μια μεταβλητή x και μία που γράφει σε αυτή τη μεταβλητή x. Στο Figure 6.3 παρουσιάζεται η εξερεύνηση όταν ο ένας αναγνώστης δρομολογείται πρώτος. Παρατηρούμε ότι εξερευνόνται ακριβώς 4 traces. Στην περίπτωση που δρομολογούντας πρώτα ο εγγραφέας 6.4 θα εξερευνότναν 5 traces με το ένα trace να γίνεται sleep set blocked.



Σχήμα 6.2: Lastzero Concuerror



Σχήμα 6.3: Scheduling Effect reader-writer-reader



Σχήμα 6.4: Scheduling Effect writer-reader-reader

6.5 Αξιολόγηση Τεχνικών Περιορισμού

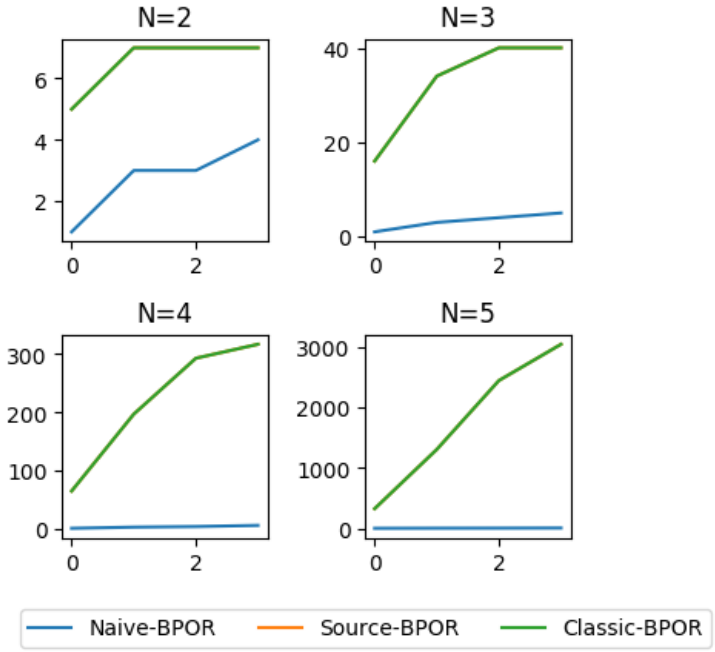
Κατά την αξιολόγηση των τεχνικών περιορισμού λάβαμε υπόψιν δύο παράγοντες. Τον αριθμό των traces που εξερευνόνται και το κατα πόσο καλύπτουμε όλο το state space. Η πρώτη μετρική σχετίζεται με τον χρόνο που απαιτείται για να βρεθεί ένα σφάλμα. Η δεύτερη είναι σημαντική καθώς αντικατοπτρίζει το tradeoff μεταξύ του χρόνου και της ακρίβειας των αποτελεσμάτων με την έννοια του ότι η μη εύρεση ενός σφάλματος δε θα συνεπάγεται την ύπαρξή του αν δεν καλύπτουμε ολόκληρο το χώρο καταστάσεων. Η σημασία του soundness έχει συζητηθεί στο Κεφάλαιο 4. Όπως είναι αναμενόμενο ταχύτεροι αλγόριθμοι περιορίζουν το soundness της αναζήτησης.

6.5.1 Αξιολόγηση Τεχνικών Περιορισμού στα Συνθετικά Προγράμματα

Τα αποτελέσματα στα διάφορα testcases παρουσιάζονται σε ενότητα. Όπως και στις προηγούμενες ενότητες παρουσιάζονται και πάλι με δύο διαφορετικούς τρόπους.

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

Πίνακας 6.2: Traces for various bound limits



Σχήμα 6.5: writer-N-readers bounded

Όπως ήταν αναμενόμενο ο Naive-BPOR εξερευνά σημαντικά λιγότερα traces από τον Nidhugg-BPOR και τον Source-DPOR. Όμως, όπως ήδη έχει αναφερθεί δεν εξερευνάται όλο το state spacce. Ο αριθμός των tracecs που εξερευνόνται στις sound εκδοχές των αλγορίθμων είναι σημαντικά μεγαλύτερος καθώς η προσθήκη των συντηρητικών διακλαδώσεων αυξάνει πολύ το state space. Ένα μη αναμενόμενο αποτέλεεσμα είναι ότι δεν υπάρχει διαφορά μεταξύ του Nidhugg-BPOR και του Source-BPOR.

6.5.2 Αξιολόγηση των Τεχνικών Περιορισμού στο RCU

Τα αποτελέσματα των διαφόρωων υλοποιήσεων του BPOR δίνονται εδώ. Ας σημειώσουμε ότι δεδομένου ότι ο Source-DPOR δεν παρουσιάζει καλύτερη επίδοση από τον Nidhugg-DPOR δεν μπορούμε να αναμένουμε διαφορά στην επίδοση και μεταξύ των φραγμένων εκδοχών τους. Πράγματα οι δοκιμές που έγιναν δεν παρουσίασαν καμία διαφοράρ και γι αυτή το λόγο μόνο η μία εκδοχή παρουσιάζεται στα αποτελέσματα. Κάθε πίνακα παρουσιάζει τα αποτελέσματα για ένα συγκεκριμένο bound σε ό,τι αφορά τον αριθμό των traces, το χρόνο που διήρκησε η εξερεύνηση όσο και στο κατα πόσο εντοπίστηκε σφάλμα. Συμβολίζουμε με F τον εντοπισμό του σφάλματος και με NF τον μη εντοπισμό.

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

Πίνακας 6.3: RCU results without bound

ver:	3.0						3.19						4.9.6					
method:	Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	1	0.2	NF	2	0.18	NF	1	0.29	NF	2	0.31	NF	1	0.57	NF	2	0.59	NF
-DFORCE_FAILURE_1	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DFORCE_FAILURE_3	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.32	NF	1	0.56	NF	2	0.61	NF
-DFORCE_FAILURE_5	1	0.17	NF	2	0.18	NF	1	0.29	NF	2	0.3	NF	1	0.56	NF	2	0.58	NF
-DLIVENESS_CHECK_1	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DLIVENESS_CHECK_2	1	0.18	NF	2	0.18	NF	1	0.3	NF	2	0.31	NF	1	0.56	NF	2	0.59	NF
-DLIVENESS_CHECK_3	1	0.17	NF	2	0.18	NF	1	0.29	NF	2	0.3	NF	1	0.56	NF	2	0.6	NF

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

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

Πίνακας 6.5: RCU results for bound $b = 1$

ver:	3.0						3.19						4.9.6					
method:	Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	9	0.29	NF	353	5.16	NF	5	0.39	NF	153	3.8	NF	5	0.68	NF	153	6.51	NF
-DFORCE_FAILURE_1	9	0.3	NF	353	5.21	NF	5	0.37	NF	153	3.77	NF	5	0.68	NF	153	6.54	NF
-DFORCE_FAILURE_3	9	0.31	NF	188	2.69	NF	5	0.39	NF	201	7.03	F	5	0.72	NF	258	14.24	F
-DFORCE_FAILURE_5	9	0.29	NF	306	4.27	NF	5	0.36	NF	105	2.51	NF	5	0.67	NF	90	3.65	NF
-DLIVENESS_CHECK_1	9	0.31	NF	182	2.62	NF	5	0.37	NF	94	1.97	NF	5	0.69	NF	79	2.84	NF
-DLIVENESS_CHECK_2	10	0.34	NF	216	3.15	NF	5	0.37	NF	94	1.97	NF	5	0.68	NF	97	3.55	NF
-DLIVENESS_CHECK_3	9	0.3	NF	201	2.78	NF	5	0.36	NF	105	2.27	NF	5	0.68	NF	88	3.19	NF

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

ver:	3.0						3.19						4.9.6					
method:	Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR			Naive-BPOR			Classic-BPOR		
	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error	traces	time	error
-	17	0.5	NF	1627	24.18	NF	8	0.42	NF	603	15.76	NF	8	0.77	NF	659	29.29	NF
-DFORCE_FAILURE_1	17	0.51	NF	1627	24.36	NF	8	0.42	NF	603	15.84	NF	8	0.77	NF	659	29.25	NF
-DFORCE_FAILURE_3	17	0.5	NF	634	8.78	NF	8	0.57	NF	1091	36.71	F	8	1.0	NF	1481	79.17	F
-DFORCE_FAILURE_5	17	0.5	NF	1157	16.0	NF	8	0.4	NF	386	9.56	NF	8	0.73	NF	324	13.01	NF
-DLIVENESS_CHECK_1	17	0.51	NF	597	8.28	NF	8	0.42	NF	251	5.18	NF	8	0.76	NF	198	6.67	NF
-DLIVENESS_CHECK_2	20	0.64	NF	767	10.93	NF	8	0.41	NF	251	5.19	NF	8	0.76	NF	258	8.99	NF
-DLIVENESS_CHECK_3	17	0.5	NF	665	9.0	NF	8	0.42	NF	292	6.24	NF	8	0.76	NF	232	7.91	NF

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

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

Πίνακας 6.8: RCU results for bound $b = 4$

ver:	3.0						3.19						4.9.6					
method:	Source-DPOR			Classic-BPOR			Source-DPOR			Classic-BPOR			Source-DPOR			Classic-BPOR		
	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound	traces	time	bound
-	19398	316.78	NF	2	0.18	NF	24760	907.7	NF	2	0.31	NF	28996	1739.91	NF	2	0.59	NF
-DFORCE_FAILURE_1	146	2.41	F	275	4.2	4	41	1.6	F	182	5.51	4	33	2.42	F	300	15.42	4
-DFORCE_FAILURE_3	2372	33.9	NF	2	0.18	NF	13264	539.05	F	201	7.03	2	8114	492.84	F	258	14.24	2
-DFORCE_FAILURE_5	4888	71.29	NF	2	0.18	NF	9	0.92	F	60	2.34	4	9	1.52	F	60	3.92	4
-DLIVENESS_CHECK_1	2024	28.94	NF	2	0.18	NF	608	12.79	NF	2	0.31	NF	488	16.72	NF	2	0.59	NF
-DLIVENESS_CHECK_2	3888	56.71	NF	2	0.18	NF	608	12.76	NF	2	0.31	NF	516	18.34	NF	2	0.59	NF
-DLIVENESS_CHECK_3	2184	30.68	NF	2	0.18	NF	688	18.79	NF	2	0.3	NF	532	18.61	NF	2	0.6	NF

Πίνακας 6.9: Comparison between DPOR and BPOR

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

Πίνακας 6.10: Comparison between DPOR and BPOR with the bug

Παρατηρούμε ότι τελικά τα σφάλματα εντοπίζονται πολύ γρηγορότερα. Το πιο σημαντικό αποτέλεσμα είναι το `-DFORCE_FAILURE_3` το οποίο εντοπίζεται σε 6 seconds για bound $b = 3$ ενώ απαιτούνται 464.77 δευτερόλεπα στην αντίστοιχη unbounded εκδοχή. Επιπλέον παρατηρούμε ότι με $b = 4$ όλα τα σφάλματα εντοπίζονται. Έτσι φαίνεται να επιβεβαιώνεται η εμπειρική παρατήρηση ότι τα λάθη εντοπίζονται σε μικρό bound. Το γεγονός ότι αυτά τα λάθη είναι κατασκευασμένα δεν μπορούν να αποτελέσουν ισχυρή ένδειξη, δεν παύουν όμως να είναι ένα στοιχείο. Όπως ήταν αναμενόμενο για μεγαλύτερα bounds ($b = 4$) ο αριθμός των traces αυξάνει εκθετικά. Ένα άλλο πολύ ενδιαφέρον αποτέλεσμα είναι ότι όσο το bound μεγαλώνει το σφάλμα παίρνει περισσότερη ώρα να βρεθεί. Στο παράδειγμα `-DFORCE_FAILURE_3` παρατηρούμε ότι το σφάλμα δεν εντοπίζεται για $b = 4$ στο δοθέν χρονικό διάστημα ενώ εντοπίζεται πολύ γρήγορα για bound $b = 2$.

6.5.3 Ένα Γνωστό Σφάλμα

Η αλλαγή στη δρομολόγηση των threads αποκάλυψε και ένα σφάλμα στην υλοποίηση του Nidhugg. Συγκεκριμένα αυτή η αλλαγή οδηγεί σε αύξηση στον αριθμό των traces. Προκειμένου να παρούσιασουμε καλή σύγκριση των αποτελεσμάτων διαδόσαμε το σφάλμα και στην unbounded εκδοχή του αλγορίθμου. Τα αποτελέσματα είναι και πάλι εντυπωσιακά 6.10.

6.6 Ισοδυναμία μεταξύ Classic-BPOR και Source-BPOR (Ορθότητα του Source-BPOR)

Προς έκπληξη μας τα αποτελέσματα του Classic-BPOR και Source-BPOR πάντα ταυτίζονται. Περαιτέρω διερεύνηση έδειξε ότι αυτές οι τεχνικές είναι ισοδύναμες.
Μια διαισθητική εξήγηση της ισοδυναμίας των δύο τεχνικών βασίζεται στην εξής παρατήση:

- Έστω B_v μια συνάρτηση που υπολογίζει το bound count τότε $B_v(pre(E, e)) \leq B_v(E)$ για κάθε $e \in E$.
- Τα σημεία που αυξάνεται το preemption count είναι τα σημεία των διακλαδώσεων.

Επομένως ο Classic-BPOR αλγόριθμος θα προσθέσει και συντηρητικές διακλαδώσεις στα σημεία που το preemption count αυξάνεται. Μη συντηρητικές διακλαδώσεςι που θα προστίθεντο απορρίπτονται από τον Source-DPOR προστίθενται ως συντηρητικές καθώς οδηγούν σε εξερεύνηση ήδη εξερευνηθέντων traces με μικρότερο preemption count.
Έτσι δείξαμε την ισοδυναμία του Source-BPOR με τον Classic-BPOR.

Κεφάλαιο 7

Επιπλέον Συζήτηση για το Πρόβλημα του Περιορισμού

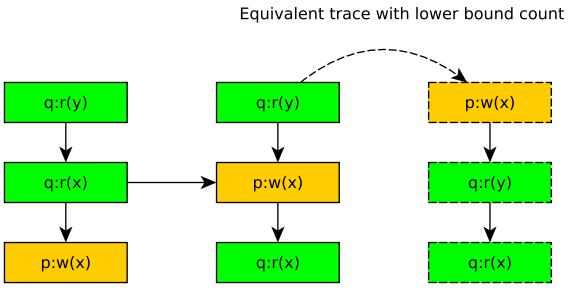
Σε αυτό το κεφάλαιο συζητόνται εναλλακτικές προσεγγίσεις του preemption bounding problem για τον DPOR. Προτείνεται μια καινούρια προσέγγιση η οποία δείχνουμε ότι είναι ισοδύναμη με την προσθήκη συντηρητικών διακλαδώσεων η οποία όμως δεν χρησιμοποιεί συντηρητικές διακλαδώσεις.

7.1 Τεχνικές χωρίς την Προσθήκη Συντηρητικών Διακλαδώσεων

Σε προηγούμενο κεφάλαιο συζητήσαμε τις προκλήσεις που προκύπτουν από τη σχεδίαση ενός Bounded DPOR αλγορίθμου. Είδαμε ότι δεν μπορούν να γίνουν σημαντικές βελτιώσεις όταν προσθέτουμε συντηρητικές διακλαδώσεις και ότι πολλές βελτιστοποιήσεις που δουλεύουν για τις μη φραγμένες εκδοχές των αλγορίθμων δεν δουλεύουν για τις φραγμένες εκδοχές τους.

7.1.1 Κίνητρο

Ο μόνος αλγόριθμος που δεν προσθέτει συντηρητικές διακλαδώσεις είναι ο Naive-BPOR. Για ένα επαρκές bound ένα trace που περιείχε σφάλμα θα εξετάζοταν από τον αλγόριθμο. Το ελλάτωμα αυτού του αλγορίθμου είναι ότι δεν είναι sound. Σε αυτό τον αλγόριθμο μια συνάρτηση υπολογίζει τον αριμό των preemptive switches. Όμως πολλές switches που υπολογίζονται μπορούν να αποφευχθούν.
Ένα παράδειγμα δίνεται στο Figure 7.1 που εξηγεί καλύτερα την ιδέα. Έστω ένα πρόγραμμα που αποτελείτα από τις διεργασίες p και q. Η διεργασία p γράφει μια μοιραζόμενη μεταβλητή x και η διεργασία q διαβάζει μια μεταβλητή y (η οποία δεν τροποιείται από κάποια άλλη διεργαία), και γράφει την μεταβλητή x. Υπάρχουν δύο δυνατά interleavings όπως φαίνεται και στην εικόνα. Αν υποθέσουμε ότι κάνουμε εξερεύνηση με bound 0 τότε και μια συντηρητική διακλάδωση πρέπει να προστεθεί.



Σχήμα 7.1: An example of avoidable preemption-switch

Στο Figure 7.1 το preemptive switch που λαμβάνει χώρα θα μπορούσε εύκολα να αποφευχθεί αντιστρέφοντας απλώς την πρώτη εντολή του q με την πρώτη την p. Αλλά τί επιτρέπει αυτή την αλλαγή;
Η απάντηση βρίσκεται στα γεγονότα που αποτελείται το block. Στην περίπτωσή μας το block αποτελείται από ένα μόνο βήμα. Το πρώτο block διαβάζει μια μεταβλητή που δε χρησιμοποιείτα από άλλο block. Επομένως τα δύο block δεν έχουν happens-before σχέση. Η παρατήρηση αυτή οδηγεί στο επόμεον ερώτημα: Ποία preemption switches είναι υποχρεωτικά; Ή ισοδύναμα ποια traces δεν μπορούν να παραχθούν χωρίς preemptive switch; Επιπλέον, είναι δυνατό για ένα trace να υπολογίσουμε τον ελάχιστο αριθμό από preemptive switches που θα απαιτούσε ένα ισοδύναμο trace;

7.1.2 Ένας Αλγόριθμος χωρίς Συντηρητκές Διακλαδώσεις

Ένας αλγόριθμος που θα έκανε bounded search θα ήταν διαφορετικός από τον Naive-BPOR μόνο σε ότι αφορά τη συνάρτηση που υπολογίζει το preemption count της δρομολόγησης. Αυτή η συνάρτηση f θα ήταν αύξουσα δηλαδή θα ίσχυε για ένα πρόθεμα E, $f(E) \leq f(E.E')$ για κάθε E'.
Η γενική μορφή του αλγορίθμου δίνεται στον Αλγόριθμο 18.

Algorithm 18: General form of the BPOR without branch addition

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

Συγκρίνοντας τον Αλγόρθμο 18 με τον 1 παρατηρούε ότι αντί να υπολογίζουμε το $B_v(S.t)$ δηλαδή το preemption count ενός trace υπολογίζουμε το $min(B_v[S.t])$ δηλαδή το ελάχιστο B_v για όλα τα ισοδύναμα traces.

7.1.3 Υπολογισμός του Ελάχιστου Preemption Count

Το μόνο που μας μένει είναι η κατασκεύ της συνάρτησης f. Για ένα trace E που αποτελείται από blocks πολλές σχέσεις happens-before ισχύουν. Κάθε ισοδύναμο trace πρέπει να συμφωνεί με αυτές τις σχέσεις. Επίσης σχέσεις happens-before ισχύουν για τις εντολές εντός ενός block. Γι αυτή την ενότητα μόνο θα μας απασχολήσουν οι σχέσεις μεταξύ blocks. Οι λόγοι που έγινε αυτή η επιλογή είναι οι εξής:

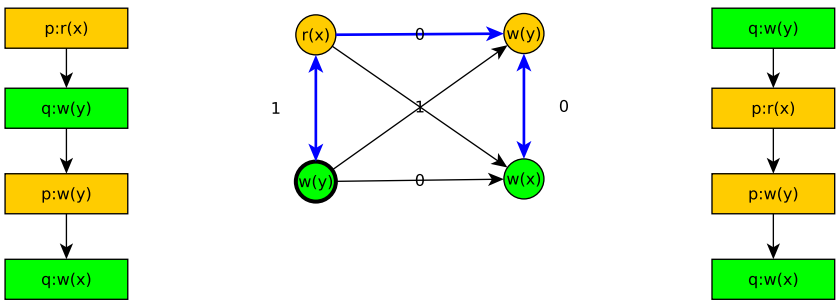
- Οι αλγόριθμοι που παρουσιάζονται στη συνέχεια είναι πολύ πιο απλοί.
- Δεν μας ενδιαφέρει να σπάσουμε περαιτέρω κάθε block και επομένως μπορούμε να δούμε toblock σαν μια ενότητα.

Αυτές οι σχέσεις happens-before σχηματίζουνε ένα γράφο. Αυτός ο γράφος αποτελείται από κόμβους που αντιστοιχούν σε blocks και ακμές που αντιστοιχούν στις σχέσεις μεταξύ τους. Προφανώς block που ανήκουν στο ίδιο thread έχουν happens-before σχέση. Επίσης μπορούμε να κινηθούμε από το ένα block στο άλλο από τη στιγμή που αυτά είναι ταυτόχρονα. Προσθέτουμε βάρη σε κάθε ακμή. Ακμές που συνδέουν blocks του ίδιου thread έχουν βάρος 0. Οι ακμές που ξεκινάν από block που είναι μπλοκαρισμένα έχουν επίσης βάρος 0. Οι υπόλοιπες ακμές έχουν βάρος ένα.

Προκειμένου να βρούμε το ελάχιστο preemption count διασχίζουμε όλα τα block του γράφου χωρίς να παραβιάζουμε τις happens-before σχέσεις. Επομένως το minimum bound count αντιστοιχεί στο ελάχιστο hamiltonian μονοπάτι το οποίο δεν παραβίαζει τις happens-before σχέσεις.
Επειδή ο υπολογισμός του ελάχιστου hamiltonian μονοπατιού είναι απαιτητικός κατασκευάζουμε ένα γράφο που περιορίζει όσο αυτό είναι δυνατό πιθανές διασχίσεις που παραβιάζουν τις σχέσεις happens-before.
Ένας αλγόριθμος που προσθέτει τα blocks στο γράφο είναι δίνεται στον Αλγόριθμο 19. Ο αλγόριθμος λειτουργεί επαγωγικά. Αρχικά ο γράφος αποτελείται από το πρώτο block. Όταν block του trace ολοκληρώνεται το προσθέτουμε στο γράφο. Προσθέτουμε κάθε block που συμβαίνει ταυτόχρονα με ένα άλλο block με διπλή ακμή Επιπλέον συνδέουμ το πιο πρόσφατο block κάθε thread που συμβαίνει πριν από το καινούριο block με ακμες που καταλήγουν στο καινούριο block.

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

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



Σχήμα 7.2: Graph example

Στο Figure 7.2 ένα απλό παράδειγμα ενός τέτοιου γράφου παρουσιάζεται. Για το trace παρατηρούμε ότι το $w(y)$ του thread q είναι ταυτόχρονο με το $r(x)$ ενώ συμβαίνει πριν το

$w(y)$ του p thread. Κάθε μετάβαση κοστίζει 1 preemption switch και γι αυτό το λόγο έχει βάρος 1. Επιπλέον, μεταβάσεις μεταξύ των ίδιων thread κοστίζουν 0. Είναι σημαντικό να σημειώσουμε ότι αν παραβιάσουμε τις happens-before σχέσεις δεν υπάρχει διάσχιση που να μπορεί να διασχίσει όλους τους κόμβους. Για παράδειγμα ξεκινώντας από το $r(x)$ και μεταβένοντας στο $w(x)$ δεν υπάρχει τρόπος να διασχίσουμε όλους του κόμβους. Μπορούμε να δούμε ότι το hamiltonian path με βάρος 1 για ένα trace. Αυτό είναι και το ελάχιστο hamiltonian μονοπάτι.
Έτσι έχουμε καταφέρει να ανάξουμε το πρόβλημα του minimum preemption coount στο πρόβλημα του minimum hamiltonian path. Το πρόβλημα αυτό είναι γνωστό NP-hard. Έτσι δεν μπρούμε να αναμένουμε ο αλγόριθμος που υπολογίζει το βάρος να είναι θεαματικά πιο γρήγορος από μια εξερεύνηση DFS.
Αυτό είναι πολύ ενδιαφέρον αποτελέσμα καθώς απεικονίζε τη δυσκολία του DPOR bounding problem καθώς η προσθήκη των συντηρητικών διακλαδώσεων υπονοεί αυτή την DFS αναζήτηση.
Τώρα που έχουμε διαπιστώσει τη δυσκολία του προβλήματος ένα καινυριο ερώτημα δημιουργείται. Μπορούμε να προσεγγίσουμε το βάρος του ελάχιστου μονοπατιού; Ένας τέτοιας αλγόριθμος δεν θα ήταν sound αλλά θα ήταν μια βελτίωση στον Naive-BPOR χωρίς να έχουμε την έκρηξη του χώρου καταστάσεων.

7.1.4 Προσεγγίζοντας το Preemption Count

Δύο μέθοδη δοκιμάστηκαν για να προσεγγίσουμε την τιμή του preemption count. Η ιδέα και για τους δύο αλγορίθμους βασίζεται στην εξής παρατήρηση: Ένα preemption switch είναι υποχρεωτικό αν δύο block της ίδιας διεργασίας A διακόπτονται από ένα block της διεργασίας B. Δηλαδή αν ισχύει $e_1(A) \rightarrow e(B) \rightarrow e_2(A)$. Στην περίπτωση που ίσχυε $e_1(A) \not\rightarrow e(B)$ ή $e(B) \not\rightarrow e_2(A)$ θα μπορούσαμε να αντιστρέψουμε τα blocks χωρίς να παραβιάζουμε τις happens-before σχέσεις.
Ο Αλγόριθμος παρουσιάζεται εδώ:

Algorithm 20: First Approximation Algorithm

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

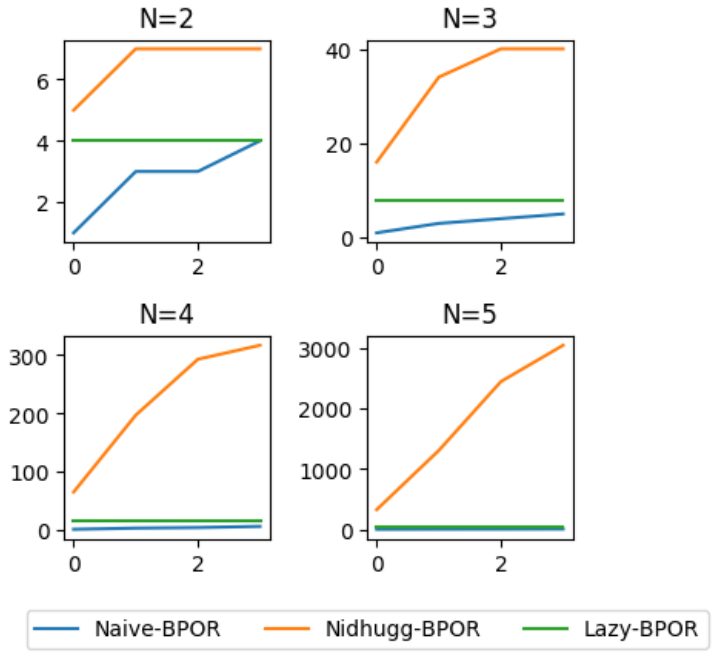
Στον Αλγόριθμο 20 εντοπίζουμε το πιο πρόσφατο block με το ίδιο pid με το τελευτάιο block. Στη συνέχεια προσπαθούμε να βρούμε αν υπάρχει κάποιο γεγονός που συμβαίνει μετά το πρώτο γεγονός και πριν το τελευταίο. Αν αυτό υπάρχει τότε αυξάνουμε τον μετρητή. Προκειμένου να διαπιστώσουμε τις happens-before σχέσεις τα vector clocks μπορούν να χρησιμοποιηθούν.
Ο δεύτερος αλγόριθμος εξερευνεί μεγαλύτερο κομμάτι του state space.

Algorithm 21: Second Approximation Algorithm

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

7.1.5 Αξιολόγηση των Αλγορίθμων Προσέγγισης

Οι προηγούμενες προσεγγίσεις που συζητήθηκαν παρουσάζουν κάποια ενδιαφέροντα αποτελέσματα. Και οι δύο αλγόριθμοι φαίνεται να είναι πιο "sound" και από τον BPOR και εξερευνούν traces που ξεπερνούν το bound. Αυτό προκύπτει από το γεγονός ότι τείνουν να υποεκτιμούν το preemption count καθώς πιο πολύπλοκες σχέσεις δεν αποκαλύπτονται. Παρατηρούμε ότι στο writer-N-readers ο αριθμός των traces είναι σταθερός για κάθε bound. Αυτό οφείλεται στο όιτ κάθε thread αποτελείται από μόνο μια εντολή.



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

7.1.6 Υλοποίηση του Lazy-BPOR

Τα προηγούμενα testcases δείξαν ότι μπορούμε να αποφύγουμε το state space explosion καθώς δεν προσθέτουμε συντηρητικές διακλαδώσεις. Το επόμενο βήμα είναι η υλοποίηση του Lazy-BPOR, ενός αλγορίθμου που υπολογίζει με ακρίβεια τα compulsory switches. Η διαφορά του από τον Naive-BPOR είναι οτι διατηρεί ένα γράφο καθ' όλη τη διάρκεια

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

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

εκτέλεσης του DPOR Η προσθήκη των κόμβων γίνεται όπως έχει ήδη περιγραφεί. Το preemption count γίνεται με τον υπολογισμό του minimum hamiltonian path.

Algorithm 22: Lazy-BPOR

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

7.1.7 Αξιολόγηση του Lazy-BPOR στο RCU

Τα αποτελέσματα παρουσιάζονται στη συνέχεια. Συγκρίνουμ τον Lazy-BPOR με την buggy εκδοχή του BPOR.
Στα Figures 7.2 και 7.4 παρουσιάζουμε τα αποτελέσματα για τα διάφορα testcases του RCU.

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

Πίνακας 7.2: Comparison between DPOR and Lazy-BPOR

ver:	3.0						3.19						4.3						4.7						4.9.6					
method:	DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR			DPOR			LBPOR		
	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound	traces	time	error	traces	time	bound
-DASSERT_0	246	3.83	F	104	2.79	2	512	17.67	F	73	4.06	2	858	37.31	F	85	8.57	2	338	15.94	F	75	6.28	2	858	40.42	F	85	9.44	2
-DFORCE_FAILURE_1	247	3.55	F	141	3.45	3	515	18.21	F	121	8.68	3	861	37.8	F	163	21.73	3	341	15.9	F	123	11.28	3	861	40.52	F	163	23.54	3
-DFORCE_FAILURE_2	4	0.34	F	4	0.35	1	3	0.55	F	3	0.52	0	3	0.7	F	3	0.71	0	3	0.86	F	3	0.87	0	3	0.88	F	3	0.9	0
-DFORCE_FAILURE_3	2372	32.1	NF	38	1.87	NF	17094	636.25	F	200	54.62	1	15349	736.84	F	233	103.89	1	15349	714.01	F	233	107.1	1	15349	793.75	F	233	111.37	1
-DFORCE_FAILURE_4	78	1.43	F	51	1.38	2	61	2.74	F	24	2.1	1	16	1.67	F	14	1.79	1	27	2.48	F	17	2.27	1	27	2.6	F	17	2.34	1
-DFORCE_FAILURE_5	12426	185.57	NF	38	3.96	NF	118	4.1	F	52	3.58	3	112	5.12	F	52	5.26	3	112	5.51	F	52	5.66	3	112	5.8	F	52	5.92	3
-DFORCE_FAILURE_6	1	0.98	F	1	0.94	0	2	2.93	F	2	2.77	0	2	4.21	F	2	4.33	0	2	8.13	F	2	8.45	0	2	8.62	F	2	8.56	0

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

Συγκρίνοντας τα αποτελέσματα παρατηρούμε ότι ο Lazy-BPOR εξετάζει λιγότερα traces αλλά απαιτεί περισσότερο χρόνο καθώς ο υπολογισμός του preemption count απαιτεί αρκετό χρόνο.

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

Πίνακας 7.4: Comparison between BPOR and Lazy-BPOR

7.2 Συμπεράσματα

Παρόλο που ο Lazy-BPOR δεν είναι πιο αποδοτικός από τους υπόλοιπους αλγορίθμους που παρουσιάστηκαν σε αυτή τη διπλωματική παρουσιάζει μερικά ενδιαφέρονται αποτελέσματα.

- Μπορούμε να εξερευνήσουμε το preemption-bounded state space χωρίς την προσθήκη conservative branches.
- Καθώς δεν προσθέτει συντηρητικές διακλαδώσεις δίνει ένα άνω φράγμα στον αριθμό των traces πολύ μικρότερο από αυτό του BPOR (τον αριθμό των traces της unbounded εκδοχής)
- Το πιο ενδιαφέρον είναι ότι ανάγει την preemption-bounded search σε ένα γνωστό γραφοθεωρητικό πρόβλημα το οποίο μπορεί να επιδεχθεί ευρηστικές για την επιτάχυνση του υπολογισμού του hamiltonian path.

Κεφάλαιο 8

Επίλογος

Σε αυτή τη διπλωματική υλοποιήσηαμε DPOR αλγορίθμους που βασίζονται σε persistent sets και υλοποιήσαμε BPOR. Συνδιάσαμε τον source-DPOR με τον BPOR και δείξαμε ότι αυτές οι προσεγγίσεις είναι ισοδύναμες. Χρησιμοποιήσαμε αυτή την προσέγγιση στο RCU και υπολογίσαμε τον ελάχιστο αριθμό από preemption που απαιτούνται για να εντοπιστούν τα σφάλματα. Επίσης τα σφάλματα εντοπίστηκαν σε πολύ μικρότερο χρονικό διάστημα. Επιπλέον διερευνήθηκαν άλλες προσεγγίσεις που δεν απαιτούν την προσθήκη συντηρητικών διακλαδώσεων.
Παρόλ' αυτά η έρευνα μας δεν έχει τελείωσει ακόμα. Επόμενες ενέργειες που πρέπει να γίνουν είναι:

- Η μελέτη άλλων bounding τεχνικών και η αξιολόγησή τους σε σύγκριση μετ την premption bounded dynamic partial order reduction.
- Η υλοποίηση του BPOR πάνω στον optimal DPOR για το Nidhugg.
- Η χρήση της μεθόδου των Observers για τη μείωση του state space.
- Η παραλληλοποίηση του Nidhugg και η επίδραση της στην επίδοση τόσο unbounded όσο και στην bounded αναζήτηση.

Βιβλιογραφία

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

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

[Abdu17a] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Comparing Source Sets and Persistent Sets for Partial Order Reduction", in Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay and Radu Mardare, editors, *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pp. 516–536, Cham, 2017, Springer International Publishing.

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

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

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

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

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

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

[Chri13] Maria Christakis, Alkis Gotovos and Konstantinos Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *Sixth IEEE*

International Conference on Software Testing, Verification and Validation (ICST 2013), pp. 154–163, Los Angeles, CA, USA, 2013, IEEE Computer Society.

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

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

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

[Coon13] Katherine E. Coons, Madan Musuvathi and Kathryn S. McKinley, "Bounded Partial-Order Reduction", in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA 2013, pp. 833–848, New York, NY, USA, October 2013, ACM.

[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.

[Emmi11] Michael Emmi, Shaz Qadeer and Zvonimir Rakamarić, "Delay-bounded Scheduling", in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pp. 411–422, New York, NY, USA, 2011, ACM.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Lamp78] Leslie Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Παράρτημα A

A.1 Modifications in test suite

For any implementation to be verified the test suite already available with Nidhugg was used. However, in the test suite 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 this appendix, we report the changes to the test driver. 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 suite will report an error. The report of the test suite when bounded DPOR is executed is shown below.
Below are listed some testcases examined throughout.

```
// 1writer_2readers.c
#include <pthread.h>
#include <assert.h>

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

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

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

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

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

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

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

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

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

  pthread_mutex_init(&m, 0);

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

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

  return 0;
}

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

#define SIZE 128
#define MAX  4

atomic_int table[SIZE];

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

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

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

      if (h < 0)
        assert(0);

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

  int idx[N];

  int main()
  {
    pthread_t t[N];

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

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

#define SIZE 128
#define MAX  1

atomic_int table[SIZE];

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

int idx[N];

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

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

        return 0;
}

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

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

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

        return NULL;
}

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

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

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

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

        return 0;
}

//lazy.c

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

pthread_mutex_t  mutex;
int data = 0;

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

```
}


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


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


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

  pthread_t t1, t2, t3;

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

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

  return 0;
}

//micro.c

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

int x=0;

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

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

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

```
  assert(0<x);
}

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

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

  return 0;
}
```