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

Παραλληλοποίηση των Dynamic Partial Order Reduction Αλγορίθμων στον Concuerror

Διπλωματική Εργασία

ΠΑΝΑΓΙΩΤΗΣ ΦΥΤΑΣ

Επιβλέπων : Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2019

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

Παραλληλοποίηση των Dynamic Partial Order Reduction Αλγορίθμων στον Concuerror

Διπλωματική Εργασία

ΠΑΝΑΓΙΩΤΗΣ ΦΥΤΑΣ

Επιβλέπων : Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 12η Ιουλίου 2019.

.....	.....	.....
Αριστείδης Παγουρτζής Αν. Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επ. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2019

.....

Παναγιώτης Φυτάς

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Παναγιώτης Φυτάς, 2019.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Ο έλεγχος και η επαλήθευση των ταυτόχρονων προγραμμάτων είναι μία πολύπλοκη εργασία. Λόγω του μη ντετερμινιστικού τρόπου με τον οποίο δουλεύει ο scheduler ενός συστήματος, μπορεί μόνο μερικά interleavings από διεργασίες να οδηγούν σε σφάλματα και συνεπώς, για να ελεγχθεί η ορθότητα ενός προγράμματος, πρέπει να εξεταστούν όλα τα πιθανά interleavings. Ο αριθμός, όμως, αυτών των interleavings είναι εκθετικός ως προς το μέγεθος του προγράμματος και τον αριθμό των νημάτων. Η πιο αποτελεσματική μέθοδος για να λυθεί αυτό το πρόβλημα είναι το stateless model checking με τη χρήση αναγωγής σε δυναμικές σχέσεις μερική διάταξης (Dynamic Partial Order Reduction ή DPOR). Καθώς τα παράλληλα συστήματα επεξεργασίας έχουν γίνει το κυρίαρχο πρότυπο των σύγχρονων υπολογιστικών συστημάτων, η παραλληλοποίηση των διαφόρων DPOR αλγορίθμων είναι αναγκαία για την κλιμάκωση αυτών των αλγορίθμων στους σύγχρονους υπολογιστές.

Αυτή η διπλωματική εργασία ασχολείται με τη παραλληλοποίηση του Concuerror, ενός stateless model checking εργαλείου που χρησιμοποιεί διάφορους DPOR αλγόριθμους για να ελέγξει ταυτόχρονα προγράμματα γραμμένα σε Erlang. Συγκεκριμένα, εστιάσαμε στο να σχεδιάσουμε παράλληλες εκδοχές για τα τους δύο βασικούς αλγορίθμους του Concuerror: τον source-DPOR και τον optimal-DPOR και στο να καταστήσουμε εφικτή την παράλληλη εξερεύνηση διαφορετικών interleavings απο τον Concuerror. Επίσης, αξιολογήσαμε την επιτάχυνση και την κλιμακωσιμότητα των υλοποιήσεων μας σε διάφορα benchmarks που χρησιμοποιούνται ευρέως για την αξιολόγηση DPOR αλγορίθμων. Συγκεκριμένα, η υλοποίηση μας επιτυγχάνει σημαντική επιτάχυνση και καταφέρνει να διατηρήσει κλιμακωσιμότητα για 32 παράλληλους schedulers.

Λέξεις κλειδιά

Stateless Model Checking, Συστηματικός Έλεγχος Ταυτοχρονισμού, Αναγωγή σε Δυναμικές Σχέσεις Μερικής Διάταξης, Παραλληλοποίηση, Ταυτοχρονισμός, Erlang

Abstract

Testing and verifying concurrent programs is quite a daunting task. Due to the non-determinism of the scheduler, errors can occur only on specific process interleaving sequences and therefore, all possible different schedulings should be examined. Currently, one of the most practical methods of dealing with the combinatorial state space explosion of this problem is a technique called Dynamic Partial Order Reduction (DPOR). As parallel processing has become the dominant paradigm in modern computer systems, developing parallel versions of DPOR algorithms is essential for scaling those algorithms to modern platforms.

This diploma thesis is concerned with the parallelization of Concuerror, a stateless model checking tool that uses various DPOR techniques for testing and verifying concurrent Erlang programs. Specifically, we have focused on developing parallel versions for the main DPOR algorithms implemented in Concuerror: source-DPOR and optimal-DPOR, and on modifying Concuerror, in order to be able to explore different interleavings in parallel. Also, we have evaluated the speedup and scalability of our implementation on certain benchmarks that are widely used for evaluating DPOR algorithms. Specifically, our implementation manages to achieve significant speedups, and, depending on the test case, scale up to 32 parallel workers.

Key words

Stateless Model Checking, Systematic Concurrency Testing, Dynamic Partial Order Reduction, Parallelization, Concurrency, Erlang

Ευχαριστίες

Αρχικά, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, τον Κωστή Σαγώνα, για τη βοήθεια του και την υποστήριξη του, κατά τη διάρκεια της προετοιμασίας της διπλωματικής μου. Θα ήθελα, ακόμα, να ευχαριστήσω τον βασικό developer του Concuerror, τον Σταύρο Αρώνη. Η βοήθεια του ήταν κομβική στο να αποκτήσω μια βαθιά κατανόηση, τόσο του Concuerror και των DPOR αλγορίθμων.

Επίσης, θέλω να ευχαριστήσω του φίλους μου που με στήριξαν κατά τη διάρκεια των σπουδών και περιστασιακά με συμβούλευαν για τη διπλωματική μου.

Κυρίως, θα ήθελα να ευχαριστήσω την οικογένεια μου που με υποστηρίζει όλα αυτά τα χρόνια. Χωρίς αυτούς δεν θα είχα καταφέρει να πετύχω τους στόχους μου.

Παναγιώτης Φυτάς,

Αθήνα, 12η Ιουλίου 2019

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-16, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2019.

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

Acknowledgements

First of all, I would like to thank my advisor, Kostis Sagonas, for his help and support during the preparation of this diploma thesis. I would also like to thank the main developer of Concuerror, Stavros Aronis. His help was instrumental in providing me with a deep understanding of both Concuerror and of DPOR algorithms.

I would, also, like to thank my friends, that both supported me during my studies and also occasionally provided advice on my thesis.

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

,

Athens, July 12, 2019

This thesis is also available as Technical Report CSD-SW-TR-1-16, National Technical University of Athens, School of Electrical and Computer Engineering, Department of Computer Science, Software Engineering Laboratory, July 2019.

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

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Acknowledgements 11

Περιεχόμενα 13

Κατάλογος πινάκων 15

Κατάλογος σχημάτων 17

Κώδικες 19

List of Algorithms 21

1. Εισαγωγή 23
- 1.1 Δοκιμή/Έλεγχος και Επαλήθευση Ταυτόχρονων Προγραμμάτων 23
- 1.2 Σκοπός της παρούσας εργασίας 23
- 1.3 Overview 24

2. Θεωρητικό Υπόβαθρο 25
- 2.1 Η γλώσσα Erlang 25
  - 2.1.1 Ταυτόχρονη Erlang 25
  - 2.1.2 Distributed Erlang 26
- 2.2 Stateless Model Checking και Partial Order Reduction 26
- 2.3 Notation 26
- 2.4 Event Dependencies 27
- 2.5 Ανεξαρτησία και Ανταγωνισμός 28
- 2.6 Επισκόπηση Concuerror 28
  - 2.6.1 Instrumenter 29
  - 2.6.2 Scheduler 29
  - 2.6.3 Logger 29

3. Δυναμικές Τεχνικές Μερικής Διάταξης 31
- 3.1 Βασικές έννοιες του DPOR 31
- 3.2 Ορισμός Source Sets 32
- 3.3 Source-DPOR Αλγόριθμος 32
- 3.4 Wakeup Δέντρα 33
  - 3.4.1 Λειτουργίες στα δέντρα αφύπνισης 34
- 3.5 Optimal-DPOR Αλγόριθμος 35

4. Παραλληλοποίηση source-DPOR Αλγορίθμου 37
4.1 Υπάρχουσα έρευνα 37
4.2 Παράλληλος source-DPOR 38
4.2.1 Βασική ιδέα 38
4.2.2 Αλγόριθμος 39
4.2.3 Εξισορρόπηση φορτίου 41
4.2.4 Ένα απλό παράδειγμα 42

5. Παραλληλοποίηση optimal-DPOR αλγορίθμου 47
5.1 Παράλληλος optimal-DPOR - Μια πρώτη προσπάθεια 47
5.1.1 Βασική ιδέα 47
5.1.2 Αλγόριθμος 48
5.1.3 Παράδειγμα 50
5.1.4 Αξιολόγηση απόδοσης 53
5.1.5 Ανάλυση απόδοσης 53
5.2 Αποδοτικός παράλληλος optimal-DPOR 55
5.2.1 Βασική ιδέα 55
5.2.2 Αλγόριθμος 56

6. Λεπτομέρειες Υλοποίησης 59
6.1 Επίλυση προβλημάτων με διεργασίες 60
6.2 Αλλαγή του μηχανισμού Replay 61

7. Αξιολόγηση Παραλληλοποίησης 63
7.1 Επισκόπηση των Benchmarks 63
7.2 Αποτελέσματα Implementation 63
7.3 Ανάλυση Απόδοσης 66
7.4 Τελικά Σχόλια 68

8. Συμπεράσματα 71

Βιβλιογραφία 73

Κατάλογος πινάκων

5.1 Parallel optimal-DPOR performance. 53

7.1 Sequential performance of source-DPOR and optimal-DPOR on four benchmarks. 64

Κατάλογος σχημάτων

4.1 Simple readers-writers example 42
4.2 Interleavings explored by the sequential source-DPOR. 43
4.3 Initial interleaving explored by the parallel algorithm. 43
4.4 Exploration of the assigned traces by each scheduler. 44
4.5 Execution Tree if Scheduler 1 returned first. 45

5.1 *Lastzero* 2 example 50
5.2 Interleavings explored by the sequential optimal-DPOR 51
5.3 Initial interleaving explored by the parallel optimal-DPOR 52
5.4 Planner Queue and Frontier sizes for the execution of lastzero. 54

7.1 Performance of readers 15 with Budget of 10000. 64
7.2 Performance of readers 15 with budget of 30000. 64
7.3 Performance of rush hour with Budget of 10000 for source and 30000 for optimal. 65
7.4 Performance of lastzero 11 with Budget of 10000 for source and 30000 for optimal. 65
7.5 Performance of indexer 17 with Budget of 10000 for source and 30000 for optimal. 66
7.6 Number of times schedulers stopped their execution with a Budget of 10000. 67
7.7 Number of times schedulers stopped their execution with a Budget of 30000. 68

Κώδικες

6.1 Environment variables 61

List of Algorithms

1 Πηγή-DPOR 33
2 Βέλτιστη-DPOR 35

3 Βρόχος ελεγκτή 39
4 Οριζόντια διαίρεση 40
5 Βρόχος εξερεύνησης χρονοπρογραμματιστή 40
6 Χειρισμός της απόκρισης χρονοδιαγράμματος 41

7 Ελεγκτής για βέλτιστο - DPOR - Πρώτη Προσπάθεια 48
8 Βέλτιστο διαχωρισμό ορίων 49
9 Διαχείριση χρονοδιαγράμματος και απάντηση προγραμματιστή 49
10 Βέλτιστος διαχωρισμός στα σύνορα - αλγόριθμος με δυνατότητα κλιμάκωσης 56
11 Scheduler Exploration Loop - Βελτιστοποιήσιμο βέλτιστο-DPOR 57

Κεφάλαιο 1

Εισαγωγή

1.1 Δοκιμή,΄Ελεγχος 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). 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 [Gode05]. Ακόμα, αυτή η τεχνική πάσχει από συνδυαστική έκρηξη. Ωστόσο, διαφορετικές παρεμβολές που μπορούν να ληφθούν μεταξύ τους από την εναλλαγή γειτονικών και ανεξάρτητων βημάτων εκτέλεσης μπορεί να θεωρηθεί ισοδύναμη. Οι αλγόριθμοι [Gode96, Clar99, Valm91] partial order reduction (POR) χρησιμοποιούν αυτή την παρατήρηση για να μειώσουν με επιτυχία το μέγεθος του διερευνηθέντος χώρου κατάστασης. Οι Dynamic Partial Order Reduction (DPOR) αλγόριθμοι [Flan05, Abdu14] καταφέρνουν να επιτύχουν μια ακόμα αυξημένη μείωση, με την ανίχνευση των εξαρτήσεων με μεγαλύτερη ακρίβεια. Οι τεχνικές DPOR έχουν εφαρμοστεί με επιτυχία σε εργαλεία όπως το Concuerror [Chri13, Goto11], Nidhugg [Abdu15], Inspect [Yang07] και Eta [Sims11].

Ο παραλληλισμός των αλγορίθμων DPOR είναι απαραίτητος προκειμένου να καταστούν κλιμακωτές στις σύγχρονες αρχιτεκτονικές υπολογιστών, αλλά και δυνητικά να επιτύγχουν σημαντικές επιταχύνσεις που θα ανακουφίσουν την επίδραση της εκθετικής έκρηξης χώρου κατάστασης. Η παραλληλισμός των αλγορίθμων DPOR που χρησιμοποιούν σταθερά σύνολα [Flan05, Lei06, Valm91] έχει ήδη γίνει, έχουν εξεταστεί και έχουν εκτελεστεί παράλληλες εκδόσεις για το Inspect [Yang08] και Eta [Sims12].

1.2 Σκοπός της παρούσας εργασίας

Σε αυτή τη διατριβή, θα επικεντρωθούμε στον παραλληλισμό του Concuerror, ενός stateless model checker που χρησιμοποιείται για τη δοκιμή ταυτόχρονων προγραμμάτων Erlang. Συγκεκριμένα, πρόκειται να:

- Αναπτύξουμε παράλληλες εκδόσεις για δύο αλγορίθμους DPOR: source-DPOR [Abdu14] και optimal-DPOR [Abdu14].
- Εφαρμόσουμε αυτούς τους παράλληλους αλγορίθμους στο Concuerror.

- Αξιολογήσουμε την απόδοση της εφαρμογής μας.

1.3 Overview

Στο Κεφάλαιο 2 παρέχουμε βασικές πληροφορίες για την Erlang, τον Concuerror και την αφαίρεση που χρησιμοποιούνται για το μοντέλο των ταυτόχρονων συστημάτων. Στο Κεφάλαιο 3 περιγράφουμε τους αλγόριθμους source-DPOR και optimal-DPOR. Στα Κεφάλαια 4 και 5 παρουσιάζουμε την παράλληλη έκδοση που έχουμε αναπτύξει για τον source-DPOR και optimal-DPOR, αντίστοιχα. Στο κεφάλαιο 6 περιγράφουμε τα βασικά προβλήματα που είχαμε με την εφαρμογή των αλγορίθμων στον Concuerror. Στο Κεφάλαιο 7 παρουσιάζουμε και αξιολογούμε την απόδοση που επιτεύχθηκε με την εφαρμογή μας. Τέλος, στο κεφάλαιο 8 συνοψίζουμε τα προηγούμενα κεφάλαια και εξετάζουμε πιθανές επεκτάσεις του έργου μας.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Η γλώσσα Erlang

Η Erlang είναι μια δηλωτική γλώσσα προγραμματισμού με ενσωματωμένη υποστήριξη για ταυτοχρονισμό, αντοχή σφάλματος, επαναφόρτωση κώδικα και αυτόματη διαχείριση μνήμης. Η Erlang αναπτύχθηκε αρχικά από την Ericsson το 1986 με σκοπό τον προγραμματισμό βιομηχανικά συστήματα τηλεπικοινωνιών. Ωστόσο, αργότερα συνειδητοποίησε ότι ήταν επίσης κατάλληλο για εφαρμογές πραγματικού χρόνου [Vird96]. Το 1998, η Erlang / OTP (Open Telecom Platform) απελευθερώθηκε ως open source και έκτοτε χρησιμοποιήθηκε εμπορικά από διάφορες εταιρείες, συμπεριλαμβανομένης της Ericsson, για μια μεγάλη ποικιλία εφαρμογών μεγάλης κλίμακας.

2.1.1 Ταυτόχρονη Erlang

Η κύρια δύναμη της Erlang απορρέει από την ενσωματωμένη υποστήριξη ταυτόχρονης λειτουργίας. Στον πυρήνα αυτού είναι οι διεργασίες Erlang, όπου η καθεμία έχει τον δικό του μετρητή προγραμμάτων, το λεξικό διαδικασίας και τη στοίβα κλήσεων. Επιπλέον, η Erlang υλοποιεί τις διαδικασίες της μέσω του συστήματος runtime του BEAM (VM της Erlang) και επομένως δεν έχουν αντιστοιχιστεί σε λειτουργικά συστήματα. Αυτές οι διαδικασίες χρησιμοποιούν ελάχιστη μνήμη, μπορούν να αλλάξουν την εργασία πολύ γρήγορα, μπορεί να τρέξει παράλληλα και χιλιάδες από αυτά μπορούν να υπάρχουν σε ένα μόνο μηχάνημα. Μια διαδικασία αναγνωρίζεται παγκοσμίως από το Pid (αναγνωριστικό διαδικασίας).

Η επικοινωνία Erlang βασίζεται κυρίως στη μετάδοση μηνυμάτων, καθώς η κατάσταση των διαδικασιών Erlang είναι (“ ως επί το πλείστον ”) δεν είναι κοινόχρηστο.Συνεπώς, ο χειριστής ‘‘!” μπορεί να χρησιμοποιηθεί για την ασύγχρονη αποστολή μηνυμάτων μεταξύ διαδικασιών, που μπορεί να είναι οποιουδήποτε τύπου δεδομένων. Το μήνυμα τοποθετείται στο “ mailbox ” (ουρά μηνυμάτων) της διαδικασίας λήψης, μέχρι να εξαχθεί από ένα *receive* έκφραση. Η έκφραση *receive* χρησιμοποιεί αντιστοίχιση προτύπων για σάρωση του γραμματοκιβωτίου με εντολή FIFO για ένα μήνυμα που ταιριάζει με αυτό το μοτίβο. Αν δεν βρεθεί τέτοιο μήνυμα, η διαδικασία λήψης αποκλείεται στο *receive*, περιμένοντας να σταλεί ένα νέο μήνυμα ή να υπάρξει ένα χρονικό όριο, σε περίπτωση που η έκφραση *receive* είχε ένα τμήμα *after*.

Η εκκίνηση μιας διαδικασίας μπορεί να γίνει αποτελεσματικά μέσω του BIF (ενσωματωμένη συνάρτηση) (*spawn/1*) (και των παραλλαγών της). Καλώντας αυτή τη συνάρτηση, δημιουργείται μια νέα ταυτόχρονη διαδικασία erlang προκειμένου να αξιολογηθεί η λειτουργία που καθορίζονται στα επιχειρήματα του *spawn*. Το PID αυτής της νέας διαδικασίας είναι η τιμή επιστροφής του *spawn* λειτουργία.

Συχνά υποστηρίζεται ότι την Erlang δεν έχει κοινή μνήμη μεταξύ διαφορετικών διαδικασιών [Arms07] και η επικοινωνία μεταξύ των διαδικασιών στηρίζεται αποκλειστικά μετάβαση μηνύματος. Ωστόσο, αυτό δεν είναι εξ ολοκλήρου αληθές, δεδομένου ότι είναι

δυνατόν για διαφορετικές διαδικασίες πρόσβαση στην ίδια μνήμη μέσω της ενότητας ETS (Erlang Term Storage).

2.1.2 Distributed Erlang

Ένας κόμβος Erlang είναι ένα runtime σύστημα Erlang που περιέχει μια πλήρη εικονική μηχανή που περιέχει το δικό του το δικό του χώρο διευθύνσεων και το σύνολο των διαδικασιών [Arms07]. Ένας κόμβος έχει αντιστοιχιστεί σε ένα όνομα η μορφή "name@host" .Οι κόμβοι Erlang μπορούν να συνδεθούν μεταξύ τους χρησιμοποιώντας cookies και μπορούν να επικοινωνήσουν μέσω του δικτύου. Τα Pids συνεχίζουν να είναι μοναδικά σε διαφορετικούς κόμβους (σε παγκόσμιο επίπεδο). Εντούτοις, μέσα δύο διαφορετικοί κόμβοι, δύο διαφορετικές διεργασίες μπορούν να έχουν το ίδιο τοπικό Pid.

Τα προγράμματα Distributed Erlang μπορούν να εκτελούνται σε διαφορετικούς κόμβους. Μια διαδικασία Erlang μπορεί να γεννηθεί σε οποιαδήποτε κόμβος, τοπικό ή απομακρυσμένο. Όλα τα πρωταρχικά (''!'', *receive*, κ.λπ.) λειτουργούν μέσω του δικτύου με τον ίδιο τρόπο που στον ίδιο κόμβο.

2.2 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.

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.

2.3 Notation

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.

2.4 Event Dependencies

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 μηνυμάτνων 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 2.1. (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 $E.w \simeq' E'.w$.

2.5 Ανεξαρτησία και Ανταγωνισμός

Μπορούμε πλέον να ορίσουμε την ανεξαρτησία μεταξύ υπολογισμών. Αν $E.p$ και $E.w$ είναι δύο ακολουθίες εκτέλεσης, τότε το $E \models p\Diamond w$ 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 $E \models p\Diamond w$ 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 $E \models p\Diamond q$ συμβολίζει ότι τα επόμενα βήματα των p και q είναι ανεξάρτητα μετά το E. Το $E' \models p\Diamond w$ συμβολίζει ότι το $E \not\models p\Diamond w$ δεν ισχύει.

Για μια ακολουθία w με $p \in w$, let $w \backslash p$ συμβολίζουμε την ακολουθία w με την πρώτη εμφάνιση 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΄Εστω 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.

2.6 Επισκόπηση Concuerror

Ο Concuerror [Chri13, Goto11] είναι ένα εργαλείο που χρησιμοποιεί διάφορα stateless model checking μοντέλα προκειμένου να συστηματικά δοκιμάστε ένα πρόγραμμα Erlang, με στόχο να ανιχνεύσετε και να αναφέρετε τα σφάλματα χρόνου εκτέλεσης που σχετίζονται με τη συναναστροφή. Συγκεκριμένα, ο Concuerror περιηγείται στο χώρο κατάστασης ενός προγράμματος, κάτω από μια δεδομένη δοκιμαστική σουίτα με καθορισμένη είσοδο, για να ελέγξετε αν εμφανίζονται ορισμένα σφάλματα σε συγκεκριμένες παρεμβολές ή για να επαληθεύσετε την απουσία τυχόν σφαλμάτων. Τέτοια σφάλματα περιλαμβάνουν αφύσικες διεργασίες εξόδου, ασυμβίβαστες εξαιρέσεις, παραβιάσεις ισχυρισμών και αδιέξοδα. Η λειτουργικότητα του Concuerror μπορεί να περιγραφεί κυρίως μέσω των κύριων στοιχείων του: του Instrumenter, του Scheduler και του Logger.

2.6.1 Instrumenter

Ο Concuerror χρησιμοποιεί τον κώδικα ενός προγράμματος χωρίς να χρειάζεται να κάνει τροποποίηση στο Erlang VM. Αντ 'αυτού, αυτό χρησιμοποιεί μια πηγή για μετάφραση πηγής που προσθέτει σημεία προτίμησης σε διάφορα σημεία του κώδικα ενός προγράμματος. Όταν η εκτέλεση ενός προγράμματος φτάσει σε ένα σημείο εξαγοράς, η διαδικασία θα αποδώσει την εκτέλεση του με αποκλεισμό σε μια εντολή λήψης, μέχρι να σταλεί ένα μήνυμα συνέχειας από το Scheduler.

Αυτό επιτρέπει τον έλεγχο του προγραμματισμού των διαδικασιών ενός προγράμματος και, ως εκ τούτου, την αναδημιουργία ενός συγκεκριμένου interleaving. Επιπλέον, αυτό επιτρέπει την τροποποίηση συγκεκριμένων BIF που αλληλεπιδρούν με την παγκόσμια κατάσταση ενός προγράμματος, εισάγοντας ένα σημείο προτίμησης πριν από αυτές τις κλήσεις λειτουργίας και τον έλεγχο της εκτέλεσης τους.

2.6.2 Scheduler

Προκειμένου να διερευνηθεί ο πλήρης χώρος ενός παράλληλου προγράμματος, είναι ζωτικής σημασίας να μπορέσουμε να " αναγκάσουμε " συγκεκριμένους προγραμματισμούς (interleavings) των διαδικασιών του. Ο Scheduler είναι υπεύθυνος για τον έλεγχο της εκτέλεση των διαδικασιών, για την παραγωγή των απαιτούμενων παρεμβολών και ταυτόχρονα για να ελέγξει και χειριστεί πιθανά σφάλματα που μπορεί να προκύψουν.

Ο Scheduler είναι επίσης υπεύθυνος για τον προσδιορισμό ποιων διεμπλοκών πρέπει να ελεγχθούν. Αυτό έγινε με την εφαρμογή διάφορων αλγορίθμων DPOR (persistent-DPOR, source-DPOR, optimal-DPOR). Ο προεπιλέγμενος αλγόριθμος που χρησιμοποιείται σήμερα από το Concuerror είναι optimal-DPOR. Ωστόσο, ο χρήστης μπορεί να καθορίσει την τεχνική που θα χρησιμοποιήσει το Concuerror για την αναζήτηση του χώρου κατάστασης.

Η λειτουργία του Scheduler μπορεί να χωριστεί σε δύο βασικά μέρη: *the exploration phase* και το *planning phase* (σύμφωνα με τους περισσότερους αλγορίθμους DPOR, όπως περιγράφεται στο Κεφάλαιο 3). Η φάση σχεδιασμού είναι υπεύθυνη για τον προσδιορισμό των αλληλεπιδράσεων που πρέπει να διερευνηθούν και η φάση εξερεύνησης είναι υπεύθυνη για την παραγωγή αυτών των παρεμβολών.

2.6.3 Logger

Τα προγράμματα δοκιμών είναι άχρηστα χωρίς να παρέχουν στον χρήστη πληροφορίες σχετικά με τον τρόπο με τον οποίο δημιουργήθηκε ένα σφάλμα. Αυτό είναι την ευθύνη του Logger. Κατά τη διάρκεια της εκτέλεσης, ο Scheduler καταγράφει πληροφορίες σχετικά με τις διερευνηθείσες παρεμβολές. Ο Logger είναι υπεύθυνος για τη σύνταξη αυτών των πληροφοριών προκειμένου να εκτυπώσει το ίχνος ενός προγράμματος που οδήγησε σε ένα πιθανό σφάλμα. Ταυτόχρονα, όταν χρησιμοποιείται σε λειτουργία debugging, ο Logger είναι απαραίτητος για τον εντοπισμό σφαλμάτων για τους προγραμματιστές του Concuerror.

Κεφάλαιο 3

Δυναμικές Τεχνικές Μερικής Διάταξης

3.1 Βασικές έννοιες του DPOR

Γενικά, οι αλγόριθμοι DPOR χρησιμοποιούν μια αναζήτηση βάθρου βάθους για πρώτη φορά για να εξερευνήσουν τον χώρο κατάστασης ενός ταυτόχρονου συστήματος. Αυτή η εξερεύνηση βασίζεται σε δύο βασικές έννοιες: *persistent sets* και *sleep sets*, οι οποίες βεβαιώνουν επαρκές τμήμα (τουλάχιστον μία παρεμβολή από διαφορετικά ίχνη Mazurkiewicz) του κρατικού χώρου, ενώ προσπαθεί να ελαχιστοποιήσει οποιαδήποτε περιττή εξερεύνηση.

Διαισθητικά, ένα επίμονο σύνολο σε κατάσταση s είναι (συγκεκριμένο) υποσύνολο του $enabled(s)$ του οποίου η εξερεύνηση εγγυάται ότι όλα θα διερευνηθούν μη ισοδύναμες παρεμβολές (από διαφορετικά ίχνη Mazurkiewicz). Αυτό είναι ζωτικής σημασίας για την απόδειξη της ορθότητας του Classic Αλγόριθμοι DPOR [Flan05], με την παραδοχή (που λαμβάνεται υπόψη από την αφαίρεσή μας) ότι ο κρατικός μας χώρος είναι ακυκλικός και πεπερασμένος. Ο τρόπος κατασκευής τέτοιων συνόλων διαφέρει από το χαρτί στο χαρτί [Flan05, Lei06, Valm91], και αυτές οι μεταβολές μπορούν να οδηγήσουν σε διαφορετικούς βαθμούς μείωσης του κρατικού χώρου.

Η τεχνική του ύπνου, η οποία είναι δωρεάν για τα επίμονα σύνολα (δεν συμβάλλει στην αξιοπιστία του αλγορίθμου), στοχεύει στην περαιτέρω μείωση του αριθμού των διερευνηθέντων διεμπλοκών. Ο ύπνος που έχει οριστεί σε μια αλληλουχία εκτέλεσης E περιέχει διαδικασίες, των οποίων η εξερεύνηση θα ήταν περιττή, εμποδίζοντας την διερεύνηση ισοδύναμων παρεμβολών.

Συγκεκριμένα, μετά την διερεύνηση του $E.p$, η διαδικασία p προστίθεται στο set sleep στο E. Από αυτό το σημείο, το p θα υπάρχει σε οποιοδήποτε σετ ύπνου μιας ακολουθίας εκτέλεσης της μορφής $E.w$, με την προϋπόθεση ότι το $E.w$ είναι επίσης μια ακολουθία εκτέλεσης και $E \models p\Diamond w$. Οι διαδικασίες στο σετ ύπνου δεν πρόκειται να εκτελεστούν από αυτό το σημείο, εκτός αν εντοπιστεί εξάρτηση. Για παράδειγμα, το p θα αφαιρεθεί από τον ύπνο που έχει οριστεί στο $E.w.q$ αν εντοπιστεί εξάρτηση μεταξύ του τα επόμενα βήματα των q και p.

Μπορούμε να αποδείξουμε [Gode96] ότι τα σύνολα ύπνου θα αποκλείσουν τελικά όλες τις πλεονάζουσες παρεμβολές και έτσι οι μόνες παρεμβολές που πρόκειται να εξερευνηθούν πλήρως θα ανήκουν σε διαφορετικά ίχνη Mazurkiewicz. Ωστόσο, αυτό δεν σημαίνει ότι τα σετ ύπνου αποφεύγουν όλες τις περιττές εξερευνήσεις. Για να επεξεργαστείτε, ύπνο σύνολα καθιστούν δυνατή την εξερεύνηση μιας αλληλεπίδρασης, η οποία ανήκει στο ίδιο ίχνος Mazurkiewicz ως ένα άλλο διεμπλοκή που έχει ήδη εξερευνηθεί, για να μπλοκάρει τελικά, έχοντας όλες τις ενεργοποιημένες διαδικασίες της εμφανίζονται στο σετ ύπνου. Αυτό ονομάζεται *sleep-set blocking* και σημαίνει ότι όλα τα πιθανά ίχνη, από το σημείο αυτό, είναι περιττές και επομένως δεν χρειάζεται να διερευνηθούν περαιτέρω. Τα σετ ύπνου δεν εγγυώνται τη βέλτιστη λειτουργία για έναν αλγόριθμο DPOR, δεδομένου ότι περιττώνονται ίχνη διερευνηθούν, αν και όχι εντελώς.

Η πηγή-DPOR [Abdu14] αντικαθιστά τα επίμονα σύνολα με *source sets* προκειμένου να επιτευχθεί σημαντικά καλύτερη μείωση της ποσότητας των διερευνηθέντων διεμπλοκών.

Ωστόσο, η πηγή-DPOR εξακολουθεί να πάσχει από ύπνο μπλοκάρισμα. Optimal-DPOR [Abdu14] συνδυάζει την έννοια των συνόλων πηγών με το *wakeup trees* για να αποφύγει πλήρως την παρεμπόδιση της δέσμης ύπνου και να οδηγήσει στην εξερεύνηση του βέλτιστου υποσυνόλου των παρεμβολών.

3.2 Ορισμός Source Sets

Προτού καθορίσουμε τυπικά τα πηγαία σύνολα, πρέπει να καθορίσουμε τις έννοιες των πιθανών αρχικών βημάτων σε μια αλληλουχία εκτέλεσης [Abdu14]:

Definition 3.1. (Initials after an execution sequence $E.w$, $I_{[E]}(w)$)
Για μια ακολουθία εκτέλεσης $E.w$, αφήστε $I_{[E]}(w)$ να υποδηλώσει το σύνολο του διαδικασίες που εκτελούν συμβάντα e στο $dom_{[E]}(w)$ που δεν έχουν "Συμβαίνει-πριν" προκατόχους στο $dom_{[E]}(w)$. Πιο τυπικά, $p \in I_{[E]}(w)$ αν $p \in w$ και δεν υπάρχει άλλο συμβάν $e \in dom_{[E]}(w)$ με $e \rightarrow_{E.w} next_{[E]}(p)$.

Με τη χαλάρωση αυτού του ορισμού, μπορούμε να πάρουμε τον ορισμό του Weak Initials, WI:

Definition 3.2. (Weak Initials after an execution sequence $E.w$, $WI_{[E]}(w)$)
Για μια ακολουθία εκτέλεσης $E.w$, αφήστε $WI_{[E]}(w)$ να δηλώσει την ένωση $I_{[E]}(w)$ και το σύνολο διαδικασίες που εκτελούν συμβάντα p έτσι ώστε $p \in enabled(s_{[E]})$.

Για να διευκρινιστούν αυτές οι σημειώσεις, για μια ακολουθία εκτέλεσης $E.w$:

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

Definition 3.3. (Source Sets)
Αφήστε το E να είναι μια ακολουθία εκτέλεσης, και αφήστε το W να είναι ένα σύνολο ακολουθιών, έτσι ώστε το $E.w$ να είναι μια εκτέλεση ακολουθία για κάθε $w \in W$. Ένα σύνολο T των διαδικασιών είναι μια πηγή που έχει οριστεί για W μετά από E εάν για κάθε $w \in W$ έχουμε $WI_{[E]}(w) \cap P = \emptyset$.

Μια άμεση συνέπεια αυτού του ορισμού είναι ότι κάθε σύνολο διαδικασιών που μπορούν να καλύψουν τον πλήρη χώρο κατάστασης μετά από μια ακολουθία εκτέλεσης E μπορεί να θεωρηθεί ένα σύνολο πηγών E.

3.3 Source-DPOR Αλγόριθμος

Εδώ παρουσιάζουμε τον αλγόριθμο source-DPOR [Abdu14].

Ένα βήμα εκτέλεσης $Explore(E, Sleep)$ είναι υπεύθυνο για τις εξερευνήσεις όλων των ιχνών Mazurkiewicz που αρχίζουν με το πρόθεμα E. Αυτές οι εξερευνήσεις αρχίζουν με την προετοιμασία του $backtrack(E)$ με μια αυθαίρετη ενεργοποιημένη διαδικασία που δεν είναι στο σετ ύπνου ($Sleep$). Από αυτό το σημείο προς τα εμπρός, για κάθε διαδικασία p που υπάρχει στο $backtrack(E)$ source-DPOR θα εκτελέσει δύο κύριες φάσεις.

Κατά την πρώτη φάση (ανίχνευση αγώνα), βρίσκουμε όλα τα συμβάντα e που συμβαίνουν στο E (δηλ., $e \in dom(E)$) τα οποία είναι αγωνιστικά με το επόμενο συμβάν του p, και αυτός ο αγώνας μπορεί να αντιστραφεί ($e \lesssim_{E.p} next_{[E]}(p)$). Για κάθε τέτοιο συμβάν e, εμείς προσπαθούμε να αντιστρέψουμε αυτόν τον αγώνα εξασφαλίζοντας ότι το επόμενο συμβάν του p γίνεται πριν από το e ή χρησιμοποιώντας τον συμβολισμό του αλγορίθμου, ότι μια ακολουθία ισοδύναμη με $E.notdep(e, E).p.proc(e).z$ (z είναι οποιαδήποτε συνέχιση της ακολουθίας εκτέλεσης) διερευνήθηκε. Ένα τέτοιο ίχνος θα μπορούσε

Algorithm 1: Πηγή-DPOR

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

να διερευνηθεί λαμβάνοντας το επόμενο διαθέσιμο βήμα από οποιαδήποτε διαδικασία στο $I_{[E']}(notdep(e, E).p)$ (όπου $E' = pre(E, e)$) στο E'. Επομένως, μια διαδικασία από $I_{[E']}(notdep(e, E).p)$ προστίθεται στο backtrack που έχει οριστεί στο E', υπό την προϋπόθεση ότι δεν υπάρχει ήδη εκεί.

Κατά την τελευταία φάση (εξερεύνηση), ανακαλύπτουμε αναδρομικά το $E.p$. Ο ύπνος που έχει οριστεί στο $E.p$ αρχικοποιείται κατάλληλα λαμβάνοντας τον ύπνο που έχει οριστεί στο E και την κατάργηση όλων των διαδικασιών των οποίων το επόμενο βήμα εξαρτάται από το επόμενο βήμα του p. Αυτό εξασφαλίζει ότι στο $E.p$ δεν θα εξετάσουμε τους αγώνες των διαδικασιών που έχουν ήδη εξεταστεί, εκτός αν αγωνιστούν με τη νέα προγραμματισμένη διαδικασία p. Μετά την ολοκλήρωση της εξερεύνησης του $E.p$, προστίθεται p στον ύπνο που έχει οριστεί στο E, επειδή θέλουμε να αποφύγουμε την εκτέλεση ισοδύναμου ίχνους.

Πρακτικά, στο Concuerror ο αλγόριθμος είναι δομημένος διαφορετικά. Η κύρια διαφορά είναι ότι ο αλγόριθμος μπαίνει στον αγώνα όταν η παρεμβολή έχει φτάσει στο τέλος της. Σε αυτό το σημείο όλες οι κούρσες που συνέβησαν κατά τη διεμπλοκή ανιχνεύονται και τα σημεία εκτροπής εισάγονται στα κατάλληλα προθέματα της πλήρους ακολουθίας εκτέλεσης. Στη συνέχεια, η εξερεύνηση συνεχίζει με την εξερεύνηση του πακέτου backtrack του μεγαλύτερου προθέματος πρώτα. Αυτό δεν επηρεάζει την αξιοπιστία του αλγορίθμου αφού το μόνο πράγμα που αλλάζει είναι η σειρά με την οποία διερευνώνται οι νέες παρεμβολές.

Θα πρέπει να σημειώσουμε εδώ ότι $Explore(E, Sleep)$ δεν χρειάζεται καμία πρόσθετη πληροφορία από τα διάφορα προθέματα του E που δεν έχει ήδη καθιερωθεί. Ωστόσο, το $Explore(E, Sleep)$ μπορεί να προσθέσει σημεία backtrack στα διάφορα προθέματα του E. Αυτό είναι ζωτικής σημασίας και πρέπει να λαμβάνεται υπόψη κατά την προσπάθεια παραλληλισμού της πηγής-DPOR.

3.4 Wakeup Δέντρα

Προκειμένου να επιτύχουμε τη βέλτιστη λειτουργία, πρέπει να αποφύγουμε εντελώς την παρεμπόδιση των παρεμβολών που έχουν υποστεί νάρκωση. Αυτό επιτυγχάνεται με συνδυάζοντας ένα μηχανισμό που ονομάζεται tree awakeup [Abdu14] με σύνολα πηγών.

Παρατηρήστε ότι στην πηγή-DPOR πρέπει να διερευνηθεί μια ακολουθία της μορφής $E'.notdep(e, E).p.proc(e).z$, αλλά μόνο μια ενιαία διαδικασία από το σύνολο Αρχικών $notdep(e, E).p$ προστίθεται δυνητικά στο σύνολο backtrack. Ως εκ τούτου, ένα κομμάτι

των πληροφοριών για το πώς να αντιστραφεί ο αγώνας χάνεται. Αυτό μπορεί να οδηγήσει σε δέσμευση σετ ύπνου, καθώς θα μπορούσε να διερευνηθεί μια εναλλακτική αλληλουχία αντι αυτου. Διαισθητικά, τα δέντρα αφύπνισης κρατούν με τη μορφή ενός δέντρου τα θραύσματα που πρέπει να εξερευνηθούν με τη σειρά για να εξερευνήσετε τις απαραίτητες παρεμβολές, αποφεύγοντας ταυτόχρονα την αποτροπή του ύπνου.

Προκειμένου να ορίσουμε δέντρα αφύπνισης, παρουσιάζουμε πρώτα τις γενικεύσεις του τις έννοιες των Αρχικών και των Αδύναμων Αρχικών, ώστε να μπορούν να περιέχουν ακολουθίες διαδικασιών αντί μόνο διαδικασιών:

- $v \sqsubseteq_{[E]} w$ denotes that exists a sequence v' such that $E.v.v'$ and $E.w$ are execution sequences with the relation $E.v.v' \simeq E.w$. What this means is that after E, v is a possible way to start an execution that is equivalent to w. To connect this to the concept of Initials we have $p \in I_{[E]}(w)$ iff $p \sqsubseteq_{[E]} w$.

- $v \sim_{[E]} w$ denotes that exist sequences v' and w' such that $E.v.v'$ and $E.w.w'$ are execution sequences with the relation $E.v.v' \simeq E.w.w'$. What this means is that after E, v is a possible way to start an execution that is equivalent to $E.w.w'$. To connect this to the concept of Weak Initials we have $p \in WI_{[E]}(w)$ iff $p \sim_{[E]} w$.

Definition 3.4. (Ordered Tree)
Ένα *ordered tree* είναι ένα ζεύγος $\langle B, \prec \rangle$, όπου το B (το σύνολο των κόμβων) είναι ένα πεπερασμένο πρόθεμα-κλειστό σύνολο ακολουθιών διαδικασιών με την κενή ακολουθία $\langle\rangle$ να είναι η ρίζα. Τα παιδιά ενός κόμβου w, του τύπου $w.p$ για κάποιο σύνολο διαδικασιών p, διατάσσονται από το $\prec$. Στο $\langle B, \prec \rangle$, μια τέτοια παραγγελία μεταξύ παιδιών έχει επεκταθεί στο σύνολο τάξη $\prec$ στο B αφήνοντας το $\prec$ να είναι η επαγόμενη σχέση μετά την παραγγελία μεταξύ των κόμβων στο B. Αυτό σημαίνει ότι αν τα παιδιά $w.p_1$ και $w.p_2$ παραγγέλλονται ως $w.p_1 \prec w.p_2$, τότε $w.p_1 \prec w.p_2 \prec w$ στην επαγόμενη μετά την παραγγελία.

Definition 3.5. (Wakeup Tree)
Αφήστε το E να είναι ακολουθία εκτέλεσης και P ένα σύνολο διαδικασιών. ένα *wakeup tree* μετά από $\langle E, P \rangle$ είναι ένα διατεταγμένο δέντρο $\langle B, \prec \rangle$, για το οποίο διατηρούνται οι ακόλουθες ιδιότητες: begin itemize item $WI_{[E]}(w) \cap P = \emptyset$ για κάθε φύλλο w του B. item Για κάθε κόμβο στο B της φόρμας $u.p$ και $u.w$ έτσι ώστε $u.p \prec u.w$ και $u.w$ να είναι ένα φύλλο η ιδιότητα $p \notin WI_{[E.u]}(w)$ πρέπει να ισχύει. end itemize

3.4.1 Λειτουργίες στα δέντρα αφύπνισης

Μια σημαντική λειτουργία που χρησιμοποιείται από τον βέλτιστο αλγόριθμο DPOR είναι η εισαγωγή νέων αρχικών θραυσμάτων παρεμβολών, που πρέπει να διερευνηθούν, στο δέντρο αφύπνισης.

Λαμβάνοντας υπόψη ένα δέντρο awakeup $\langle B, \prec \rangle$ μετά από $\langle E, P \rangle$ και κάποια ακολουθία w με Το $E.w$ είναι μια ακολουθία εκτέλεσης τέτοια ώστε $WI_{[E]}(w) \cap P = \emptyset$, χρησιμοποιούνται οι ακόλουθες ιδιότητες καθορίστε το $insert_{[E]}(w, \langle B, \prec \rangle)$:

- $insert_{[E]}(w, \langle B, \prec \rangle)$ is also a wakeup tree after $\langle E, P \rangle$.

- Any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$.

- $insert_{[E]}(w, \langle B, \prec \rangle)$ contains a leaf u with $u \sim_{[E]} w$.

Αφήστε το v να είναι το μικρότερο (σύμφωνα με τη σειρά $\prec$) στο B με $v \sim_{[E]} w$. Η επιχείρηση Το $insert_{[E]}(w, \langle B, \prec \rangle)$ μπορεί να ληφθεί ως $\langle B, \prec \rangle$, με την προϋπόθεση ότι v είναι ένα φύλλο ή προσθέτοντας $v.w'$ ως φύλλο και διατάζοντας το μετά από όλους τους

υπάρχοντες κόμβους στο B της μορφής $v.w''$, όπου w' είναι η συντομότερη ακολουθία με $w \sqsubseteq_{[E]} v.w'$.

Ας περιγράψουμε επίσης τη λειτουργία $subtree(\langle B, \prec \rangle, p)$. Για ένα δέντρο αφύπνισης $\langle B, \prec \rangle$ και μια διαδικασία $p \in B$, $subtree(\langle B, \prec \rangle), p)$ χρησιμοποιείται για να δηλώσει το subtree του $\langle B, \prec \rangle$ με ρίζες στο p. Πιο τυπικά,$subtree(\langle B, \prec \rangle), p) = \ langleB', \ prec' \ rangle$ where $B' = \ w \ midp.w \ \}$ and $\ prec'$ is the extension of $\ prec$ to B'.

3.5 Optimal-DPOR Αλγόριθμος

Εδώ παρουσιάζεται ο βέλτιστος αλγόριθμος DPOR όπως περιγράφεται στο αρχικό του έγγραφο [Abdu14].

Algorithm 2: Βέλτιστη-DPOR

```
1  Function Explore(E,Sleep,WuT)
2      if enabled(s_[E]) = ∅ then
3          foreach e, e' ∈ dom(E) such that (e ≲_E e') do
4              let E' = pre(E, e);
5              let v = notdep(e, E).proc(e');
6              if sleep(E') ∩ WI_[E'](v) = ∅ then
7                  insert_[E'](v, wut(E'));
8      else
9          if WuT ≠ ⟨{⟨⟩}, ∅⟩ then
10             wut(E) := WuT;
11         else
12             choose p ∈ enabled(s_[E]);
13             wut(E) := ⟨{p}, ∅⟩;
14         sleep(E) := Sleep;
15         while ∃p ∈ wut(E) do
16             let p = min_≺{p ∈ wut(E)};
17             let Sleep' := {q ∈ sleep(E) | E ⊨ p◇q};
18             let WuT' = subtree(wut(E), p);
19             Explore(E.p, Sleep', WuT');
20             add p to sleep(E);
21             remove all sequences of form p.w from wut(E);
```

Ομοίως, με τους άλλους αλγορίθμους, το βέλτιστο-DPOR έχει δύο διαφορετικές φάσεις: ανίχνευση αγώνα και εξερεύνηση κατάστασης. Ωστόσο, ο αλγόριθμος είναι δομημένος διαφορετικά. Με τον ίδιο τρόπο που η πηγή-DPOR είναι δομημένη στο Concuerror, το βέλτιστο-DPOR ανιχνεύει μόνο τους αγώνες όταν έχει επιτευχθεί μια μέγιστη ακολουθία εκτέλεσης (δηλαδή, δεν υπάρχει αριθ ενεργοποιημένες διαδικασίες). Αυτό είναι απαραίτητο επειδή η προϋπόθεση για την εισαγωγή νέων δέντρων αφύπνισης είναι μόνο ισχύει όταν το κομμάτι που πρόκειται να εισαχθεί περιέχει όλα τα συμβάντα στις πλήρεις εκτελέσεις αυτού δεν συμβαίνουν μετά από e και αυτές που συμβαίνουν μετά e.

Η φάση ανίχνευσης αγώνων λειτουργεί ως επί το πλείστον παρόμοια με την πηγή-DPOR. Οι κύριες διαφορές έχουν να κάνουν με το γεγονός ότι απαιτούμε τη γνώση του συνόλου ύπνου για κάθε πρόθεμα E' και ότι χρησιμοποιούνται οι έννοιες των Αδύναμων Αρχικών αντί για τα αρχικά για να καθορίσουν αν ένα θραύσμα πρόκειται να εισαχθεί στο δέντρο αφύπνισης, το οποίο έχει τις ρίζες του στο πρόθεμα E'.

Στη φάση εξερεύνησης μιας μη μεγίστης ακολουθίας εκτέλεσης, το δέντρο αφύπνισης αυτής της ακολουθίας αρχικοποιείται στο δεδομένου WuT. Εάν το WuT είναι άδειο, τότε επιλέγεται μια αυθαίρετα ενεργοποιημένη διαδικασία, με τον ίδιο τρόπο που θα εφαρμοζόταν για τους μη βέλτιστους αλγορίθμους. Στη συνέχεια, για κάθε διαδικασία που υπάρχει στο WuT η λειτουργία εξερεύνησης θα καλείται αναδρομικά, με την κατάλληλη subtree του $wut(E)$. Αυτό εγγυάται την διερεύνηση του πλήρους τμήματος. Μετά την ολοκλήρωση της αναδρομικής κλήσης, οι ακολουθίες που εξερευνήθηκαν αφαιρούνται από το δέντρο αφύπνισης. Τα σύνολα ύπνου αντιμετωπίζονται με παρόμοιο τρόπο με τα προηγούμενα αλγορίθμους.

Όπως υποδηλώνει το όνομα, ο optimal-DPOR είναι βέλτιστο υπό την έννοια ότι ποτέ δεν ερευνά δύο μέγιστες ακολουθίες εκτέλεσης που ανήκουν στο ίδιο ίχνος Mazurkiewicz, καθώς μπορεί να αποδειχθεί ότι καμία παρεμβολή δεν είναι sleep-set blocked [Abdu14].

Κεφάλαιο 4

Παραλληλοποίηση source-DPOR Αλγορίθμου

Ο Concuerror χρησιμοποιεί κυρίως αλγορίθμους DPOR για τη συστηματική δοκιμή ταυτόχρονων προγραμμάτων Erlang. Επομένως, η παραλληλισμός του Concuerror συνεπάγεται το σχεδιασμό παράλληλων εκδόσεων για τους αλγορίθμους DPOR. Δεδομένου ότι το Concuerror γράφεται στο Erlang, το οποίο είναι μια λειτουργική γλώσσα που βασίζεται στο μήνυμα που μεταδίδει στο κοινό τα δεδομένα μεταξύ των διαδικασιών, θα ακολουθήσουμε μια προσέγγιση που περνάει το μήνυμα, αναπτύσσοντας παράλληλα τους αλγορίθμους μας.

Σε αυτό το κεφάλαιο, θα παρουσιάσουμε την παράλληλη έκδοση του αλγορίθμου source-DPOR. Ας αρχίσουμε συζητήστε μερικές υπάρχουσες εργασίες για την παραλληλισμό των αλγορίθμων DPOR που βασίζονται επίμονα στο σύνολο.

4.1 Υπάρχουσα έρευνα

Όταν παραλληλίζουμε τους αλγορίθμους DPOR, ιδανικά θα θέλαμε να αναπτύξουμε παράλληλους αλγορίθμους που διερευνούν ακριβώς τον ίδιο αριθμό παρεμβολών με τις διαδοχικές εκδόσεις τους. Με άλλα λόγια, θέλουμε να διατηρήσουμε την αξιοπιστία των αλγορίθμων μας, ενώ παράλληλα δεν διερευνάμε περισσότερες παρεμβολές από ό, τι είναι απαραίτητο

Με μια πρώτη ματιά, ο παραλληλισμός των αλγορίθμων DPOR μπορεί να φαίνεται απλός. Δεδομένου ότι ο κρατικός χώρος ενός προγράμματος δεν περιέχει κύκλους, θα πρέπει απλώς να διανείμουμε αυτό το κρατικό χώρο σε πολλούς εργάτες-προγραμματιστές. Για παράδειγμα, στην περίπτωση του αλγορίθμου 1, θα εκχωρούσαμε ένα πρόθεμα της φόρμας $E.p$ σε έναν προγραμματιστή. Αυτός ο προγραμματιστής θα ήταν υπεύθυνος για την εκτέλεση του $explore(E.p, Sleep)$. Ωστόσο, αυτή η προσέγγιση οδηγεί σε δύο βασικά ζητήματα [Yang07].

Πρώτον, οι αλγόριθμοι DPOR εντοπίζουν τους αγώνες και ενημερώνουν τα σύνορα εξερεύνησης με μη τοπικό τρόπο. Ενώ κλήσεις στο $explore(E.p, Sleep)$ μπορεί να εγγυηθεί ότι για όλες τις μέγιστες εκτελέσεις ακολουθίες της μορφής $E.w$, ο αλγόριθμος έχει διερευνήσει κάποια ακολουθία εκτέλεσης E' η οποία είναι σε $[E.w]_{\simeq}$, σημεία backtrack μπορούν επίσης να εισαχθούν στα προθέματα του $E.p$. Για παράδειγμα, ας υποθέσουμε ότι αναθέτουμε σε έναν δρομολογητή την εξερεύνηση του $E.p$ και σε έναν άλλο προγραμματιστή την εξερεύνηση του $E.q$. Εφόσον χρησιμοποιούμε το πέρασμα μηνυμάτων ως μοντέλο προγραμματισμού μας, οι προγραμματιστές μας θα χρησιμοποιούν διαφορετικά αντίγραφα του προθέματος E. Και οι δύο αυτές διερευνήσεις μπορούν να οδηγήσουν στην προσθήκη της ίδιας διαδικασίας r στο backtrack στο E. Αυτό θα σήμαινε αυτό και οι δύο προγραμματιστές θα καταλήξουν να φωνάζουν $explore(E.r, Sleep)$. Επομένως, διαφορετικοί προγραμματιστές μπορεί να καταλήξουν να εξερευνούν πλήρως ταυτόσημες παρεμβολές. Προκειμένου να καταπολεμηθούν αυτές οι περιττές εξερευνήσεις, Yang et aï. Το [Yang07] προτείνει μια ευρετική που απλώς τροποποιεί την τεμπέλη προσθήκη καταχωρήσεων backtrack στο όριο εξερεύνησης [Flan05] για να γίνει πιο πρόθυμος. Η

προσθήκη εγγραφών backtrack πιο ανυπόμονα, δηλαδή, νωρίτερα κατά τη φάση εξερεύνησης, μειώνει τις πιθανότητες δύο διαφορετικών εργαζομένων διερευνώντας ταυτόσημες παρεμβολές. Στο παραπάνω σενάριο, αυτή η στρατηγική θα μπορούσε να έχει οδηγήσει στην προσθήκη r στο backtrack στο E πριν αναθέτοντας την εξερεύνηση των $E.p$ και $E.q$ σε διαφορετικούς προγραμματιστές. Επομένως, και οι δύο προγραμματιστές μας θα ξέρουν ότι υπάρχει r στο το backtrack του E και κανένας από αυτούς δεν θα το είχε προσθέσει, αποφεύγοντας την διπλή διερεύνηση του $E.r$. Ωστόσο, αυτό είναι απλά ένα ευρετικό, πράγμα που σημαίνει ότι ανάλογα με το δοκιμασμένο πρόγραμμα, σημαντική ποσότητα περιττών υπολογισμών μπορεί να συμβεί. Ιδιαίτερα, αυτό το ευρετικό αποτυγχάνει να αποτρέψει τις περιττές εξερευνήσεις, όταν κλαδιά στον κώδικα οδηγούν σε διαφορετικούς αγώνες.

Δεύτερον, το μέγεθος των διαφόρων κομματιών του κρατικού χώρου δεν μπορεί να είναι γνωστό a priori. Αυτό σημαίνει ότι κάποια μορφή εξισορρόπησης φορτίου είναι απαραίτητο για την επίτευξη γραμμικής επιτάχυνσης. Yang et αϊ. Το [Yang07] προτείνει τη χρήση κεντρικού αντισταθμιστή φορτίου για την εκφόρτωση εργασίας από έναν προγραμματιστή. Συγκεκριμένα, ένας χρονοπρογραμματιστής καλεί τον εξισορροπητή φορτίου όταν το σύνολο αριθμός καταχωρήσεων backtrack στην ακολουθία εκτέλεσης του χρονοπρογραμματισμού υπερβαίνουν ένα όριο. Ωστόσο, για διαφορετικά προγράμματα και διαφορετικό αριθμό εργαζομένων, διαφορετικό όριο τιμές θα πρέπει να χρησιμοποιούνται [Sims12]. Ακόμα, οι Yang κ.ά. δεν παρέχουν πληροφορίες για το πρόβλημα της επιλογής ενός κατάλληλου ορίου.

Simsa et αϊ. [Sims12] παρέχουν έναν καταλληλότερο τρόπο επίλυσης αυτών των ζητημάτων. Χρησιμοποιώντας ένα κεντρικό *Controller*, το οποίο παρακολουθεί το ρεύμα *execution tree* (ένα δέντρο του οποίου τα υποκαταστήματα αντιστοιχούν στις τρέχουσες ακολουθίες εκτέλεσης E των προγραμματιστών), βεβαιώνουν ότι δεν υπάρχουν περιττές εξερευνήσεις. Προτείνουν επίσης τη χρήση του *time slicing* για την επίτευξη εξισορρόπησης φορτίου.

4.2 Παράλληλος source-DPOR

Εδώ πρόκειται να παρουσιάσουμε πώς να παραλληλίσουμε αποτελεσματικά τον αλγόριθμο source-DPOR, τροποποιώντας τον παράλληλο αλγόριθμο που παρουσιάστηκε από τους Simsa et al. [Sims12].

4.2.1 Βασική ιδέα

Κανονικά, οι αλγόριθμοι DPOR εκτελούν μια πρώτη αναζήτηση στο βάθος του χώρου κατάστασης για να ελέγξουν για λανθασμένες παρεμβολές. Αντ 'αυτού, εμείς πρόκειται να χρησιμοποιήσουν πολλαπλές αναζητήσεις βάθους-πρώτων (διαμένοντας τα σύνορα της αναζήτησης) για να διερευνήσουν τον κρατικό μας χώρο.

Θα χρησιμοποιήσουμε ένα συγκεντρωτικό *Controller* που θα είναι υπεύθυνη για τη διανομή της εξερεύνησης σύνορα με διαφορετικούς εργάτες-προγραμματιστές. Ο ελεγκτής είναι επίσης θα επιβλέπει την παράλληλη εξερεύνηση, οπότε αποφεύγουμε να διερευνήσουμε περισσότερες παρεμβολές από τη διαδοχική έκδοση. Για να γίνει αυτό, ο ελεγκτής θα παρακολουθήσει τα σύνορα που διερευνάται με τη μορφή ενός *execution tree*. Εν ολίγοις, ο δέντρο εκτέλεσης αντιπροσωπεύει τον κρατικό χώρο του προγράμματος μας. Οι κόμβοι του δένδρου εκτέλεσης αντιπροσωπεύουν μη ντετερμινιστικά σημεία επιλογής και άκρα αντιπροσωπεύουν τις μεταβολές της κατάστασης του προγράμματος. Ένα μονοπάτι από τη ρίζα του δέντρου σε ένα φύλλο στη συνέχεια κωδικοποιεί μοναδικά ένα πρόγραμμα μια ακολουθία εκτέλεσης. Θα χρησιμοποιήσουμε επίσης τον όρο κλάδο για να αναφερθούμε σε ακολουθίες εκτέλεσης μέσα στο δέντρο εκτέλεσης.

Για να αποφύγουμε τις περιττές εξερευνήσεις, θα χρησιμοποιήσουμε το έννοια του *ownership* [Sims12] ενός κόμβου ενός χώρου κατάστασης. Ένας προγραμματιστής διαθέτει αποκλειστικά έναν κόμβο του κρατικού χώρου αν είναι είτε περιελάμβανε είσοδο backtrack εντός του τμήματος των συνόρων που ανατέθηκε στον συγκεκριμένο προγραμματιστή ή αν είναι απόγονος ενός κόμβου που έχει ο χρονοπρογραμματιστής. Όλοι οι άλλοι κόμβοι θεωρούνται ότι έχουν α *disputed* ιδιοκτησία.

Ο προγραμματιστής, κατά τη διεξαγωγή της πρώτης αναζήτησης βάθους, πρόκειται να είναι επιτρέπεται να διερευνήσει μόνο τους κόμβους που κατέχει. Όταν αντιμετωπίζετε αμφισβητούμενες κόμβους, ο χρονοπρογραμματιστής θα αναφέρει στον ελεγκτή, που θα το κάνει να παρακολουθείτε τα πλήρη ενεργά σύνορα. Εάν αυτός ο αμφισβητούμενος κόμβος δεν υπάρχει μέσα στα πλήρη σύνορα, τότε δεν υπάρχει άλλη ο προγραμματιστής έχει διερευνήσει αυτόν τον κόμβο και επομένως τον προγραμματιστή μπορεί να *claim ownership* στον αμφισβητούμενο κόμβο και να συνεχίσει με διερευνώντας το. Διαφορετικά, η κυριότητα του αμφισβητούμενου κόμβου έχει ζητηθεί από κάποιον άλλο προγραμματιστή και επομένως από αυτόν τον κόμβο μπορεί να απορριφθεί από τα σύνορα του χρονοπρογραμματιστή μας.

Για παράδειγμα, εάν ένας υπεύθυνος χρονοπρογραμματισμού είναι υπεύθυνος για την εξερεύνηση μιας αλληλουχίας εκτέλεσης E που έχει ένα μόνο backtrack είσοδος p στο E, τότε αυτός ο χρονοπρογραμματιστής κατέχει κάθε κόμβο αυτός είναι ένας απόγονος του $E.p$ δηλαδή, κατέχει κάθε ακολουθία εκτέλεσης που αρχίζει με το πρόθεμα $E.p$ (ή αλλιώς, το πλήρες δευτερεύον τμήμα που έχει ρίζες στο $E.p$). Τώρα ας υποθέσουμε ότι κατά τη διάρκεια της εξερεύνησης του υποτάγματος που έχει ρίζες στο $E.p$, έχει προστεθεί r στο το backtrack στο E. Δεδομένου ότι αμφισβητείται αυτός ο κόμβος, ο προγραμματιστής δεν θα διερευνήσει αυτή την είσοδο backtrack. Αντ 'αυτού, θα αναφέρει πίσω στον ελεγκτή, για να καθορίσετε αν κάποια άλλη είσοδο έχει ήδη διεκδικήσει την κυριότητα πάνω από $E.r$.

4.2.2 Αλγόριθμος

Algorithm 3: Βρόχος ελεγκτή

```
1  Function controller_loop(N, Budget, Schedulers)
2      E_0 ← an arbitrary initial execution sequence;
3      Frontier ← [E_0];
4      T ← an execution tree rooted at E_0;
5      while Frontier ≠ ∅ do
6          Frontier ← partition(Frontier, N);
7          while exists an idle scheduler S and an unassigned execution sequence E in
             Frontier do
8              E_c ← a copy of E;
9              mark E as assigned in Frontier;
10             spawn(S, explore_loop(E_c, Budget));
11         Frontier, T ← wait_scheduler_response(Frontier, T);
```

Η λογική του ελεγκτή εμφανίζεται στον αλγόριθμο 3. Ο ελεγκτής διατηρεί ένα $Frontier$, το οποίο είναι ένα σύνολο ακολουθιών εκτέλεσης E, και ένα δέντρο εκτέλεσης T, το οποίο περιέχει ως κλάδους τις ακολουθίες εκτέλεσης του $Frontier$. Για όσο χρονικό διάστημα υπάρχει ακολουθία εκτέλεσης στο $Frontier$ ($Frontier \neq \emptyset$), ο ελεγκτής θα κατατμήσει τις ακολουθίες $Frontier$ στις περισσότερες από τις ακολουθίες εκτέλεσης N. Στη συνέχεια, τον ελεγκτή θα προσπαθήσει να εκχωρήσει όλες τις μη ανατεθείσες ακολουθίες εκτέλεσης

σε οποιονδήποτε χρονοπρογραμματιστή, με την αναπαραγωγή $explore_loop(E_c, Budget)$ λειτουργίες. Τέλος, θα μπλοκάρει μέχρι να λάβει μια απάντηση από έναν προγραμματιστή.

Algorithm 4: Οριζόντια διαίρεση

```
1  Function partition(Frontier, N)
2      for all E ∈ Frontier do
3          while total_backtrack_entries(E) > 1 and size(Frontier) < N do
4              E' ← the smallest prefix of E that has a backtrack entry ;
5              p ← a process ∈ backtrack(E');
6              E'_c ← a copy of E';
7              remove p from backtrack(E');
8              add p to sleep(E');
9              add backtrack(E') to sleep(E'_c);
10             add E'_c to Frontier;
11     return Frontier;
```

Κατά τη διάρκεια της φάσης *partitioning* (Αλγόριθμος 4), επιθεωρούμε το τρέχον $Frontier$ για να καθορίσουμε θα πρέπει να δημιουργήσουμε πρόσθετες ακολουθίες εκτέλεσης. Κάθε ακολουθία εκτέλεσης που περιέχει περισσότερες από μία καταχωρήσεις backtrack διαιρείται σε πολλαπλές αλληλουχίες μέχρις ότου είτε το $Frontier$ περιέχει αλληλουχίες N είτε όλες τις ακολουθίες έχουν ακριβώς μια είσοδο backtrack. Είναι ζωτικής σημασίας να τροποποιήσουμε κατάλληλα τα σετ ύπνου, διότι, αν ήμασταν απλά να καταργήσουμε καταχωρήσεις backtrack, ο αλγόριθμός μας θα έχει αυξημένο αριθμό παρεμβαλλόμενων παρεμβολών που έχουν αποκλειστεί από τον ύπνο. Επιπλέον, θα ήμασταν επίσης καταλήγουν δυνητικά να προσθέσουν εκ νέου τις ίδιες καταχωρήσεις backtrack, οι οποίες θα οδηγήσουν για να εξερευνήσετε τις διπλές παρεμβολές.

Algorithm 5: Βρόχος εξερεύνησης χρονοπρογραμματιστή

```
1  Function explore_loop(E_0, Budget)
2      StartTime ← get_time();
3      E ← E_0;
4      repeat
5          E' ← explore(E);
6          E' ← plan_more_interleavings(E');
7          E ← get_next_execution_sequence(E');
8          CurrentTime ← get_time();
9      until CurrentTime − StartTime > Budget or size(E) ≤ size(E_0);
10     send E to Controller ;
```

Ο αλγόριθμος 5 περιγράφει λεπτομερώς πώς οι προγραμματιστές διερευνούν τον καθορισμένο χώρο κατάστασης. Μια κλήση στο $explore_loop(E_0, Budget)$ εγγυάται ότι για όλες τις μέγιστες εκτελέσεις ακολουθίες της μορφής $E_0.w$, ο αλγόριθμος έχει διερευνήσει κάποια ακολουθία εκτέλεσης E'_0 η οποία είναι σε $[E_0.w]_{\simeq}$ Χρησιμοποιούμε $explore(E)$ και $plan_more_interleavings(E')$ ως τρόπος υψηλού επιπέδου για να περιγράψει τις κύριες φάσεις (εξερεύνηση κατάστασης και ανίχνευση αγώνα) της διαδοχικής πηγής-DPOR. Η συνάρτηση $plan_more_interleavings(E')$ θα μπορούσε να προσθέσει σημεία backtrack στα προθέματα του E_0. Αυτό θα μπορούσε να οδηγήσει σε διαφορετικούς προγραμματιστές να εξερευνήσουν ταυτόσημες παρεμβολές. Αποφεύγουμε αυτό έχοντας

η επιστροφή συνάρτησης $get_next_execution_sequence(E')$ το μεγαλύτερο πρόθεμα του E' που έχει ένα μη κενό σύνολο backtrack. Αυτό οδηγεί σε μια πρώτη εξερεύνηση βάθους των εκχωρημένων κρατήστε το διάστημα πριν εξετάσετε τις παρεμβολές εκτός του χώρου κατάστασης. Η εξερεύνηση συνεχίζει μέχρι να συναντήσουμε ένα πρόθεμα του E_0 ($size(E'') \leq size(E_0)$). Αυτό είναι απαραίτητο δηλώνουν ότι ο συγκεκριμένος χρονοπρογραμματιστής δεν θα διερευνήσει τις αλληλεπιδράσεις εκτός του χώρου κατάστασής του. Όταν η εξερεύνηση τερματίζει, τα σημεία backtrack που προστίθενται στα προθέματα του E_0 θα αναφερθούν στον ελεγκτή.

Algorithm 6: Χειρισμός της απόκρισης χρονοδιαγράμματος

```
1 Function wait_scheduler_response(Frontier, T)
2     receive E from a scheduler;
3     remove E from Frontier;
4     E', T ← update_execution_tree(E, T);
5     if E' has at least one backtrack entry then
6         add E' to Frontier;
7     return Frontier, T;
```

Όταν ο ελεγκτής λαμβάνει μια απάντηση (ακολουθία εκτέλεσης E) από έναν προγραμματιστή (αλγόριθμος 6), θα δοκιμάστε και αναφέρετε οποιαδήποτε νέα καταχωρήσεις backtrack στο E στο δέντρο εκτέλεσης T. Αυτό έγινε μέσω της συνάρτησης $update_execution_tree(E, T)$. Αυτή η λειτουργία επαναλαμβάνει ταυτόχρονα την ακολουθία εκτέλεσης και το δέντρο εκτέλεσης και εκείνες τις εγγραφές backtrack του E που δεν βρέθηκαν στο δέντρο εκτέλεσης, προστίθενται σε αυτό και δεν αφαιρούνται από την ακολουθία εκτέλεσης. Αυτό σημαίνει ότι αυτή η ακολουθία εκτέλεσης είναι η πρώτη προς *claim ownership* πάνω από αυτές τις καταχωρήσεις και ο χώρος κατάστασης που υπάρχει κάτω από αυτές. Όλες οι καταχωρήσεις backtrack που υπάρχουν ήδη στο T αφαιρούνται από το ακολουθία εκτέλεσης E (και προστίθεται στα σύνολα ύπνου στα κατάλληλα προθέματα του E), επειδή κάποια άλλη ακολουθία εκτέλεσης έχει ήδη ζητήσει την ιδιοκτησία τους. Αυτή η ενημερωμένη ακολουθία εκτέλεσης είναι στη συνέχεια που προστίθεται στο $Frontier$ του ελεγκτή. Με αυτόν τον τρόπο διασφαλίζουμε ότι δύο προγραμματιστές δεν μπορούν να εξερευνήσουν το ίδιο παρεμβολές.

Κατά την ενημέρωση του δένδρου εκτέλεσης, χρησιμοποιούμε επίσης την αρχική ακολουθία εκτέλεσης που είχε εκχωρηθεί σε έναν προγραμματιστή (εκείνη που δηλώνεται ως E_0 στον Αλγόριθμο 5) για να καταλάβουμε ποια τμήματα του δέντρου εκτέλεσης έχουν έχουν ήδη εξερευνηθεί. Τα τμήματα αυτά διαγράφονται από το δέντρο εκτέλεσης. Αυτό είναι υποχρεωτικό για να διατηρηθεί το μέγεθος του δένδρου εκτέλεσης αναλογικό προς το μέγεθος του τρέχοντος $Frontier$.

4.2.3 Εξισορρόπηση φορτίου

Προκειμένου να επιτύχουμε αξιοπρεπείς επιταχύνσεις και επεκτασιμότητα, είναι απαραίτητο να έχουμε εξισορρόπηση φορτίου [Sims12]. Αυτό έγινε μέσα από το χρόνο που κόπηκε την εξερεύνηση των ακολουθιών εκτέλεσης. αυτος ΕΙΝΑΙ Ο ΛΟΓΟΣ πίσω από τη χρήση του $Budget$ στον Αλγόριθμο 5. Με την επιστροφή των προγραμματιστών μετά από μια συγκεκριμένη χρονική περίοδο, μπορούμε να διασφαλίσουμε ότι ακόμα κι αν ο χώρος που τους έχει δοθεί είναι μεγαλύτερος σε σύγκριση με αυτόν των άλλων προγραμματιστών, θα το κάνουν τελικά βγαίνουν και έχουν την ακολουθία εκτέλεσης και στη συνέχεια, ο χώρος κατάστασής τους, χωρισμένοι. Πόσο αποτελεσματικό είναι αυτή η μέθοδος καθορίζεται από δύο μεταβλητές, το ανώτερο όριο N με τον αριθμό των ακο-

```
p:
write(x)
```

```
q:
read(x)
write(x)
```

```
r:
write(x)
```

Σχήμα 4.1: Simple readers-writers example

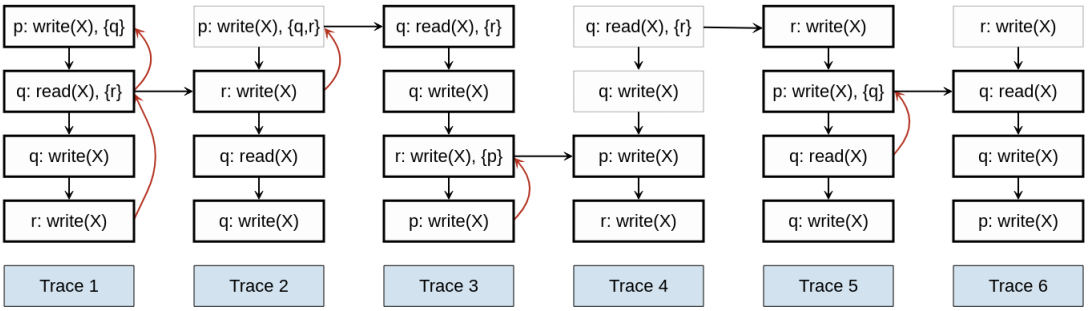
λουθιών εκτέλεσης στο μας $Frontier$ και $Budget$ ενός χρονοπρογραμματιστή.

Το ανώτερο ανώτατο όριο N σημαίνει ότι υπάρχει μεγαλύτερη εργατική δεξαμενή στους εργαζόμενους και δεν πρέπει πάντα να περιμένουν για να τερματίσει η συνάρτηση $update_execution_tree(E, T)$. Ως εκ τούτου, η χρήση των schedulers αυξάνεται. Ωστόσο, ένα μεγαλύτερο N σημαίνει αυξημένες απαιτήσεις μνήμης, καθώς περισσότερες αλληλουχίες εκτέλεσης είναι ενεργές ανά πάσα στιγμή. Κυρίως, σημαίνει επίσης ότι ο κρατικός χώρος χωρίζεται σε μικρότερα θραύσματα. Αυτό αυξάνει πόσο συχνά ένας προγραμματιστής ανακαλύπτει τους αμφισβητούμενους κόμβους, οι οποίοι οδηγεί σε αύξηση της επικοινωνίας μεταξύ των προγραμματιστών και του ελεγκτή. Ο καθορισμός αυτού του ορίου για το διπλασιασμό του ποσού των χρονοπρογραμματιστών, παράγει αξιοπρεπή αποτελέσματα για τις περισσότερες περιπτώσεις δοκιμής[Sims12].

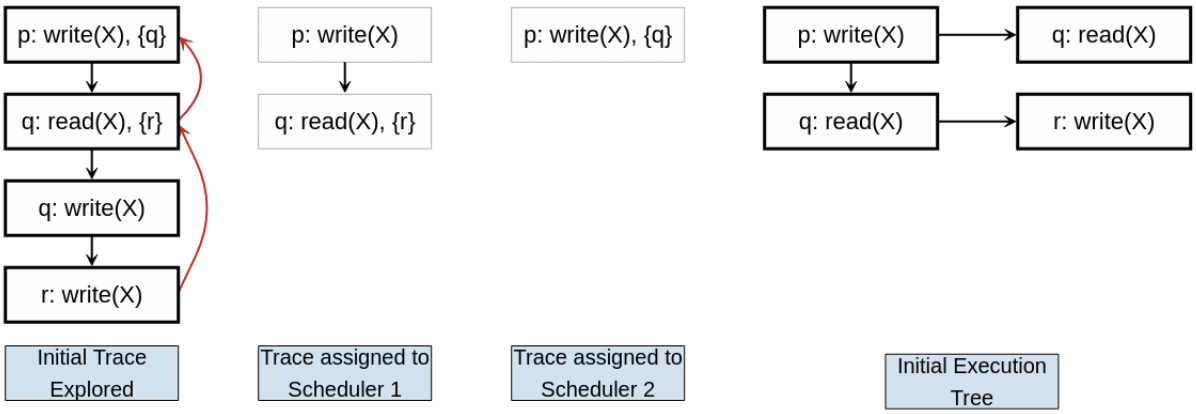
Μικρότερες τιμές $Budget$ οδηγούν σε πιο ισορροπημένο φόρτο εργασίας, καθώς η εργασία διανέμεται συχνότερα. Ωστόσο, εξαιρετικά χαμηλές τιμές μπορεί να οδηγήσουν σε αυξημένη γενική επικοινωνία μεταξύ του ελεγκτή και των προγραμματιστών. Αυτό μπορεί επίσης να αναγκάσει τον ελεγκτή να καταστεί σημαντικό εμπόδιο. Ο καλύτερος τρόπος να αντιμετωπιστεί αυτό, είναι να επιλέξετε μια αρχική τιμή $Budget$ περίπου 10 δευτερολέπτων. Όταν ένας χρονοπρογραμματιστής ξεκινά μια νέα εξερεύνηση, η αξία του προϋπολογισμού του θα εκχωρηθεί δυναμικά από τον ελεγκτή ανάλογα με το ποσό του αδρανείς χρονοπρογραμματιστές. Για παράδειγμα, η πρώτη εκτέλεση θα πρέπει να έχει προϋπολογισμό $\frac{Budget}{n}$ ήταν n είναι η συνολική ποσότητα προγραμματιστές, που είναι όλα αδρανείς. Όταν οι μισοί χρονοπρογραμματιστές είναι αδρανείς, αυτή η τιμή θα πρέπει να είναι $\frac{Budget}{2}$, κλπ. Αυτό καθιστά δυνατή την ύπαρξη μειωμένη επικοινωνία (υψηλότερος προϋπολογισμός) σε περιόδους με πολλούς πολυάριθμους προγραμματιστές και καλύτερη εξισορρόπηση (χαμηλότερος προϋπολογισμός) σε περιόδους με πολλούς χρονοπρογραμματιστές.

4.2.4 Ένα απλό παράδειγμα

Ας εξετάσουμε το παράδειγμα στο σχήμα 4.1. Σε αυτή την περίπτωση έχουμε 3 διαδικασίες που γράφουν και διαβάζουν μια κοινόχρηστη μεταβλητή x. Το σχήμα 4.2 αντιπροσωπεύει τα ίχνη που διερευνήθηκαν κατά τη διάρκεια της διαδοχικής πηγής-DPOR. Χρησιμοποιούμε ένα τολμηρό ορθογώνιο για να αντιπροσωπεύσουμε ένα νέο συμβάν και ένα ελαφρύ ορθογώνιο για να υποδηλώσουμε ένα επαναλαμβανόμενο γεγονός. Τα κόκκινα άκρα αντιπροσωπεύουν οι αγώνες που εντοπίζονται και σχεδιάζονται. Η πηγή που έχει οριστεί σε μια κατάσταση αναπαριστάται μέσα στις παρενθέσεις.

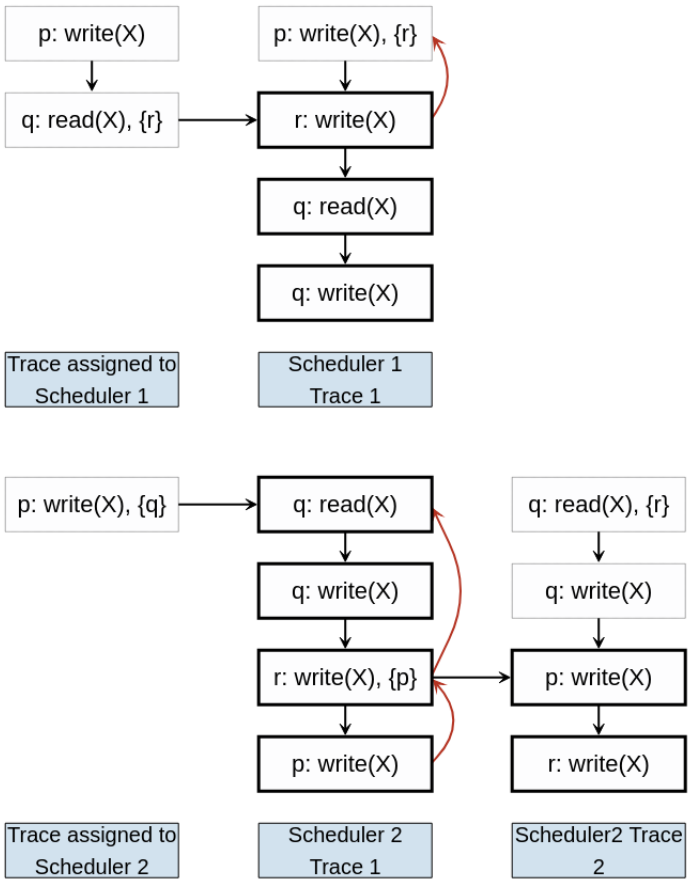


Σχήμα 4.2: Interleavings explored by the sequential source-DPOR.



Σχήμα 4.3: Initial interleaving explored by the parallel algorithm.

Το σχήμα 4.3 απεικονίζει το αρχικό βήμα της παράλληλης πηγής-DPOR. Αυτή η αρχική αλληλουχία εκτέλεσης, μαζί με τις ανιχνευόμενες φυλές, χωρίζεται σε θραύσματα τα οποία έχουν εκχωρηθεί σε διαφορετικούς προγραμματιστές. Αυτή η εικόνα περιέχει επίσης το αρχικό δέντρο εκτέλεσης που αντιπροσωπεύει τον κρατικό χώρο που υπάρχει στα σύνορα αναζήτησης σε αυτό το σημείο.



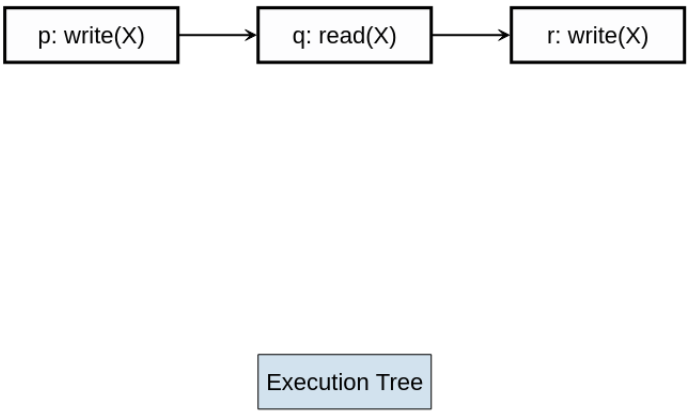
Σχήμα 4.4: Exploration of the assigned traces by each scheduler.

Στο σχήμα 4.4 απεικονίζουμε τον τρόπο με τον οποίο κάθε δρομολογητής εξερευνά την εκχωρημένη του ακολουθία εκτέλεσης. Ο πρώτος προγραμματιστής διερευνά το πρώτο του ίχνος: $p.r.q.q$ (ισοδύναμο με το δεύτερο ίχνος στο Σχήμα 4.2). Μετά την πραγματοποίηση της ανίχνευσης αγώνα, δεν υπάρχουν καταχωρήσεις backtrack κάτω από το καθορισμένο ίχνος. Η μόνη προγραμματισμένη ακολουθία εκτέλεσης είναι η ακολουθία r. Ωστόσο, αυτή η ακολουθία δεν διερευνάται, αφού δεν ανήκει στον κρατικό χώρο του χρονοπρογραμματισμού. Ο ελεγκτής έχει εκχωρηθεί στο δεύτερος χρονοπρογραμματιστής την ακολουθία εκτέλεσης q. Μετά τη διερεύνηση του πρώτου ίχνους, εντοπίζονται δύο ακόμη φυλές. Είναι σημαντικό να παρατηρήσετε εδώ ότι το πρώτο ίχνος του δεύτερου χρονοπρογραμματιστή είναι ισοδύναμο με το τρίτο ίχνος του διαδοχικού αλγορίθμου. Ωστόσο, ο διαδοχικός αλγόριθμος ανιχνεύει μόνο μία κούρσα. Αυτό συμβαίνει επειδή με τον διαδοχικό αλγόριθμο ο αγώνας μεταξύ $q : read(x)$ και Το $r : write(x)$ είχε ήδη ανιχνευθεί κατά τη δεύτερη διεμπλοκή και έτσι ο προγραμματισμός του παραβλέπεται. Αντίθετα, στον παράλληλο αλγόριθμο αυτή η παρεμβολή διερευνάται από έναν άλλο προγραμματιστή και επομένως δεν υπάρχει ήδη γνώση αυτής της φυλής. Αυτό οδηγεί σε και οι δύο προγραμματιστές μας έχουν εντοπίσει τον ίδιο αγώνα. Παρ 'όλα αυτά, και στους δύο χρονοπρογραμματιστές αυτή η νέα είσοδος backtrack είναι έξω από τους κρατικό χώρο.

Αυτό σημαίνει ότι θα πρέπει να αναφέρουν τα αποτελέσματά τους στον ελεγκτή. Οι προγραμματιστές που αναφέρουν πρώτα τα αποτελέσματά τους, θα είναι αυτός που θα ενημερώσει το δέντρο εκτέλεσης εισάγοντας τη νέα καταχώρηση που βρέθηκε. Αυτό ο χρονοπρογραμματιστής θα προσθέσει την ακολουθία εκτέλεσης στα σύνορα, η οποία θα χωριστεί και πάλι (δεν χρειάζεται εδώ ένα διαμέρισμα αφού τα ανεξερεύνητα σύ-

νορα θα έχουν μόνο μία φυλή). Τότε αυτή η ακολουθία εκτέλεσης θα είναι ανατεθεί σε έναν αναπάντητο προγραμματιστή. Ο προγραμματιστής που αναφέρει δευτερόλεπτα στον ελεγκτή, δεν θα μπορέσει να εισαγάγει το δικό του είσοδος backtrack στο δέντρο εκτέλεσης, επειδή αυτή η καταχώρηση θα είναι ήδη εκεί και η καταχώρηση backtrack αφαιρείται από την ακολουθία εκτέλεσης. Αυτή η ακολουθία εκτέλεσης θα αφεθεί χωρίς περισσότερες καταχωρήσεις backtrack και ως τέτοια δεν θα είναι εισάγονται στα σύνορα. Αυτό εγγυάται ότι δεν θα διερευνήσουμε πανομοιότυπες παρεμβολές περισσότερες από μία φορές.

Ας υποθέσουμε ότι ο Scheduler 1 ήταν εκείνος που κατάφερε να αναφέρει πρώτα τον ελεγκτή. Στη συνέχεια, το δέντρο εκτέλεσης θα είναι το ένα που απεικονίζεται στο σχήμα 4.5. Παρατηρήστε εδώ ότι τα κράτη που διερευνήθηκαν από τον Scheduler 1 διαγράφηκαν από το δέντρο εκτέλεσης. Αυτό διατηρεί το μέγεθος του δέντρου εκτέλεσης ανάλογο προς το μέγεθος του τα σημερινά σύνορα εξερεύνησης. Σε αυτό το σημείο ο προγραμματιστής 1 θα προσθέσει την ακολουθία εκτέλεσης του στα σύνορα. Μετά το Ο ελεγκτής θα εκχωρήσει αυτή την ακολουθία στο Scheduler 1 (αφού ο Scheduler 2 δεν έχει ακόμη επιστρέψει) και ο Scheduler 1 θα διερευνήσει τα τελευταία 2 ίχνη (ίχνη 5 και 6 από το διαδοχικό παράδειγμα). Αφού και οι δύο προγραμματιστές 1 και 2 επέστρεψαν την εκτέλεση θα έχουν τελειώσει, αφού δεν θα υπάρχουν πλέον ίχνη για εξερεύνηση.



Σχήμα 4.5: Execution Tree if Scheduler 1 returned first.

Κεφάλαιο 5

Παραλληλοποίηση optimal-DPOR αλγορίθμου

Σε αυτό το κεφάλαιο, θα παρουσιάσουμε μια πρώτη προσπάθεια παραλληλισμού του βέλτιστου-DPOR αλγόριθμο, να παράσχει μια ανάλυση για το γιατί αυτή η προσπάθεια δεν επιτυγχάνει οποιαδήποτε επιτάχυνση και στη συνέχεια να παρουσιάσει έναν βελτιωμένο παράλληλο βέλτιστο αλγόριθμο DPOR.

5.1 Παράλληλος optimal-DPOR - Μια πρώτη προσπάθεια

5.1.1 Βασική ιδέα

Η παραλληλισμός του αλγόριθμου βέλτιστου-DPOR είναι σημαντικά πιο πολύπλοκη από την πηγή-DPOR. Γνωρίζουμε ότι κάθε φορά που καλείτε $Explore(E, Sleep, WuT)$ επιστρέφει κατά τη διάρκεια του αλγόριθμου 2, τότε για όλες τις μέγιστες ακολουθίες εκτέλεσης της μορφής $E.w$, ο αλγόριθμος έχει διερευνήσει κάποια ακολουθία εκτέλεσης E' η οποία βρίσκεται στο $[E.w]_{\simeq}$ [Abdu14]. Με άλλα λόγια, οι κλήσεις προς $Explore(E, Sleep, WuT)$ εγγυώνται ότι η πλήρης υποδία έχει ρίζα στο E θα εξερευνηθούν. Ωστόσο, το πλήρες WuT σε κάποια ακολουθία εκτέλεσης E δεν μπορεί να είναι γνωστό μέχρι να ολοκληρώσουμε διερευνώντας όλες τις ακολουθίες εκτέλεσης που παραγγέλλονται πριν από το E, σύμφωνα με τη συνολική σειρά του κρατικού μας χώρου (Ορισμός 3.4). Αυτό συμβαίνει επειδή η λειτουργία $insert_{[E']}(v, wut(E'))$ μπορεί να προσθέσει καταχωρήσεις σε κάθε δέντρο αφύπνισης μιας ακολουθίας εκτέλεσης που παραγγέλλεται μετά την τρέχουσα ακολουθία εκτέλεσης.

Επομένως, όταν εκχωρείτε ένα ελλιπές δέντρο αφύπνισης σε έναν προγραμματιστή, δεν υπάρχει να εγγυηθεί ότι ο προγραμματιστής θα διερευνήσει το πλήρες καθορισμένο χώρο κατάστασης. Αυτό σημαίνει ότι εάν α ο χρονοπρογραμματιστής εισάγει ένα κομμάτι σε ένα δέντρο αφύπνισης που ανήκει σε διαφορετικό χρονοδιάγραμμα, δεν μπορούμε να το μάθουμε θραύσμα (ή διαφορετικό αλλά ισοδύναμο θραύσμα). Ως αποτέλεσμα, η έννοια της ιδιοκτησίας ενός η είσοδος backtrack, όπως ορίζεται στο Κεφάλαιο 4, δεν μπορεί να παραμείνει η ίδια για τον βέλτιστο αλγόριθμο DPOR.

Ένα άλλο βασικό ζήτημα με το βέλτιστο DPOR είναι το γεγονός ότι η συνάρτηση $insert_{[E]}(v, wut(E))$ μπορεί να καταλήξει και εισαγωγή σε $wut(E)$ και ακολουθία εκτέλεσης που είναι διαφορετική από v (αλλά θα οδηγήσει στην εξερεύνηση ισοδύναμου υποδέντρου). Αυτό σημαίνει ότι δύο ακολουθίες εκτέλεσης, ενώ διαφορετικές, μπορεί να είναι ισοδύναμη. Αυτό μπορεί να οδηγήσει σε διαφορετικούς χρονοπρογραμματιστές που εισάγουν στο δικό τους οι σειρές εκτέλεσης δέντρων αφύπνισης που ενώ είναι διαφορετικές, μπορεί να παράγουν ισοδύναμες παρεμβολές και ως εκ τούτου τη βέλτιστη λειτουργία του αλγορίθμου θα μπορούσε να χαθεί.

Ωστόσο, γνωρίζουμε ότι οποιοδήποτε φύλλο $\langle B, \prec \rangle$ παραμένει ένα φύλλο $insert_{[E]}(w, \langle B, \prec \rangle)$ [Abdu14]. This means that, during the sequential algorithm, any fragment that εισάγεται σε ένα δέντρο αφύπνισης είναι ένα κομμάτι που πρέπει να διερευνηθεί, εκτός και αν αφαιρεθεί κατά τη διάρκεια τη φάση εξερεύνησης του αλγορίθμου. Επομένως, όταν εισάγουμε ένα κομμάτι σε ένα δέντρο αφύπνισης, μπορούμε να το εξερευνήσουμε

εκτός λειτουργίας, αρκεί να είμαστε προσεκτικοί για να αφαιρέσουμε αυτές τις ακολουθίες εκτέλεσης θα είχε αφαιρεθεί στον αλγόριθμο διαδοχικών μεταβλητών. Μπορούμε να επωφεληθούμε από αυτό για να δημιουργήσουμε έναν αλγόριθμο που μπορεί να εξερευνήσει πολλές αλληλεπιδράσεις παράλληλα, αλλά η κούρσα ανιχνεύει κάθε διερευνημένη αλληλεπίδραση διαδοχικά.

5.1.2 Αλγόριθμος

Λόγω του γεγονότος ότι πρέπει να έχουμε παράλληλη εξερεύνηση των παρεμβολών, αλλά διαδοχικό σχεδιασμό, πρέπει να αποσυνδέσουμε τον κανονικό βρόχο εξερεύνησης ενός προγραμματιστή σε δύο διαφορετικά μέρη: *state exploration* και *race detection − planning*. Οι εργαζόμενοι (οι προγραμματιστές) θα είναι υπεύθυνοι για το πρώτο μέρος. Για το δεύτερο μέρος, πρόκειται να χρησιμοποιήσουμε μια κεντρική Σχεδιαστής. Ωστόσο, για να είναι δυνατή η καλύτερη διανομή της διαθέσιμης εργασίας στους προγραμματιστές όταν ο προγραμματιστής είναι απασχολημένος, πρόκειται επίσης να χρησιμοποιήσουμε έναν ελεγκτή.

Algorithm 7: Ελεγκτής για βέλτιστο - DPOR - Πρώτη Προσπάθεια

```
1  Function controller_loop(Schedulers)
2      E_0 ← an arbitrary initial execution sequence;
3      Frontier ← [E_0];
4      T ← an execution tree rooted at E_0;
5      PlannerQueue ← empty;
6      while size(Frontier) > 0 and size(PlannerQueue) > 0 do
7          Frontier ← partition(Frontier);
8          while exists an idle scheduler S and an unassigned execution sequence E in
             Frontier do
9              E_c ← a copy of E;
10             spawn(S, explore(E_c));
11         while the Planner is idle and PlannerQueue ≠ empty do
12             E ← PlannerQueue.pop();
13             update_trace(E, T);
14             spawn(Planner, plan(E));
15         wait_response(Frontier, T, PlannerQueue);
```

Ο αλγόριθμος 7 περιγράφει τη λειτουργικότητα του ελεγκτή. Ομοίως με την παράλληλη έκδοση πηγής-DPOR, ο ελεγκτής είναι υπεύθυνος για τη διατήρηση του τρέχοντος Frontier (καθώς και του διαχωρισμού του) και του τρέχοντος δένδρου εκτέλεσης και για την ανάθεση ακολουθιών εκτέλεσης στους προγραμματιστές, για όσο χρονικό διάστημα έχουμε προγραμματιστές σε αναμονή και διαθέσιμο έργο.

Εκτός από αυτό, ο ελεγκτής διατηρεί επίσης μια ουρά ολοκληρωμένων ακολουθιών εκτέλεσης που πρέπει να ανιχνευθούν. Όταν ο προγραμματιστής είναι αδρανής και η ουρά δεν είναι κενή, η ακολουθία εκτέλεσης ενημερώνεται (μέσω του $update_trace(E, T)$) και στη συνέχεια αποστέλλεται στον προγραμματιστή, ώστε οι αγώνες του να μπορούν να εντοπιστούν. Κατά την ενημέρωση της ακολουθίας εκτέλεσης από το δέντρο εκτέλεσης, οι υποτάξεις του δέντρου εκτέλεσης που διατάσσονται μετά την εκτέλεση της ακολουθίας (σύμφωνα με την παραγγελία της κατάστασής μας ο χώρος Ορισμός 3.4) εισάγονται στο δέντρο εκτέλεσης ως δέντρα αφύπνισης *not_owned*. Αυτό εγγυάται ότι δεν πρόκειται να εισαχθούν πλεονάζοντα κομμάτια για μελλοντικές εξερευνήσεις και επομένως ο αλγόριθμος παραμένει βέλτιστος.

Ο αγώνας λειτουργίας $plan(E)$ ανιχνεύει την πλήρως διερευνηθείσα ακολουθία εκτέλεσης E σύμφωνα με τη λογική του βέλτιστου-DPOR (Αλγόριθμος 2). Όταν ολοκληρωθεί ο προγραμματισμός της ακολουθίας, τα αποτελέσματα αναφέρονται στον ελεγκτή. Η συνάρτηση $explore(E)$ διερευνά την ακολουθία εκτέλεσης E μέχρι τη μέγιστη ακολουθία εκτέλεσης έχει επιτευχθεί και αναφέρει την ακολουθία εκτέλεσης στον ελεγκτή.

Algorithm 8: Βέλτιστο διαχωρισμό ορίων

```
1  Function partition(Frontier)
2      for all E ∈ Frontier do
3          while wakeup_tree_leaves(E) > 1 do
4              E' ← a prefix of E with wut(E') ≠ ∅ ;
5              v ← a leaf ∈ wut(E');
6              E'_c ← a copy of E';
7              mark v as not_owned at wut(E');
8              {Prefix, v, Suffix} ← split_wut_at(v, wut(E'_c));
9              mark Prefix and Suffix as not_owned at wut(E'_c);
10             add E'_c to Frontier;
11     return Frontier;
```

Το διαχωρισμό των συνόρων εξερεύνησης (Αλγόριθμος 8) έχει δύο κύριες διαφορές, σε σύγκριση με την παράλληλη πηγή-DPOR. Πρώτον, το σύνορο γίνεται εντελώς χωρισμένο, ώστε να μεγιστοποιήσουμε την παραλληλισμό της φάσης εξερεύνησης. Δεύτερον, οι καταχωρίσεις που διανέμονται από μία ακολουθία εκτέλεσης, δεν αφαιρούνται απλά από το backtrack και προστίθενται σε το σετ ύπνου. Είναι ζωτικής σημασίας εδώ να διατηρηθεί η σωστή διάταξη μεταξύ των αλληλεπιδράσεων (Definition 3.4), επειδή κατά τη διάρκεια της φάσης εξερεύνησης του αλγόριθμου βέλτιστου-DPOR, μπορούν να αφαιρεθούν ακολουθίες από το δέντρο αφύπνισης. Η διατήρηση της παραγγελίας θα διατηρήσει αυτή τη συμπεριφορά ανέπαφη στην παράλληλη έκδοση. Επομένως, η δεδομένη καταχώρηση χαρακτηρίζεται ως not_owned στην κατανεμημένη ακολουθία. Η συνάρτηση $split_wut_at(v, wut(E'_c))$ διαιρεί το αντίγραφο του δέντρου αφύπνισης σε τρία μέρη: το $Prefix$ (οι εντολές δέντρου αφύπνισης που παραγγέλθηκαν πριν από την ακολουθία v), το φύλλο v και το $Suffix$ (οι καταχωρήσεις αφύπνισης διέταξαν μετά από v). Οι πρώτες διεργασίες επεξεργασίας των καταχωρήσεων του το $Prefix$ προστίθεται στον ύπνο που έχει οριστεί στη νέα ακολουθία εκτέλεσης E'_c (π.χ. εάν $p.q.r$ είναι ένα φύλλο στο $Prefix$, τότε p προστίθεται στο $sleep(E'_c)$). Οι εγγραφές $Suffix$ σημειώνονται ως not_owned στο E'_c.

Algorithm 9: Διαχείριση χρονοδιαγράμματος και απάντηση προγραμματιστή

```
1  Function wait_response(Frontier, T, PlannerQueue)
2      receive a message M;
3      if M is sent from a Scheduler then
4          E ← M;
5          PlannerQueue.push(E);
6      else if M is sent from the Planner then
7          E ← M;
8          update_execution_tree(E, T);
9          add E to Frontier;
```

Μετά την εκχώρηση της διαθέσιμης εργασίας στους διαθέσιμους προγραμματιστές

```
p:
i = N
while x[i] != 0 and i > 0:
    i = i − 1
```

```
q:
R1 = x[0]
R2 = x[0]
assert(R1 == R2)
x[1] = R2 + 1
```

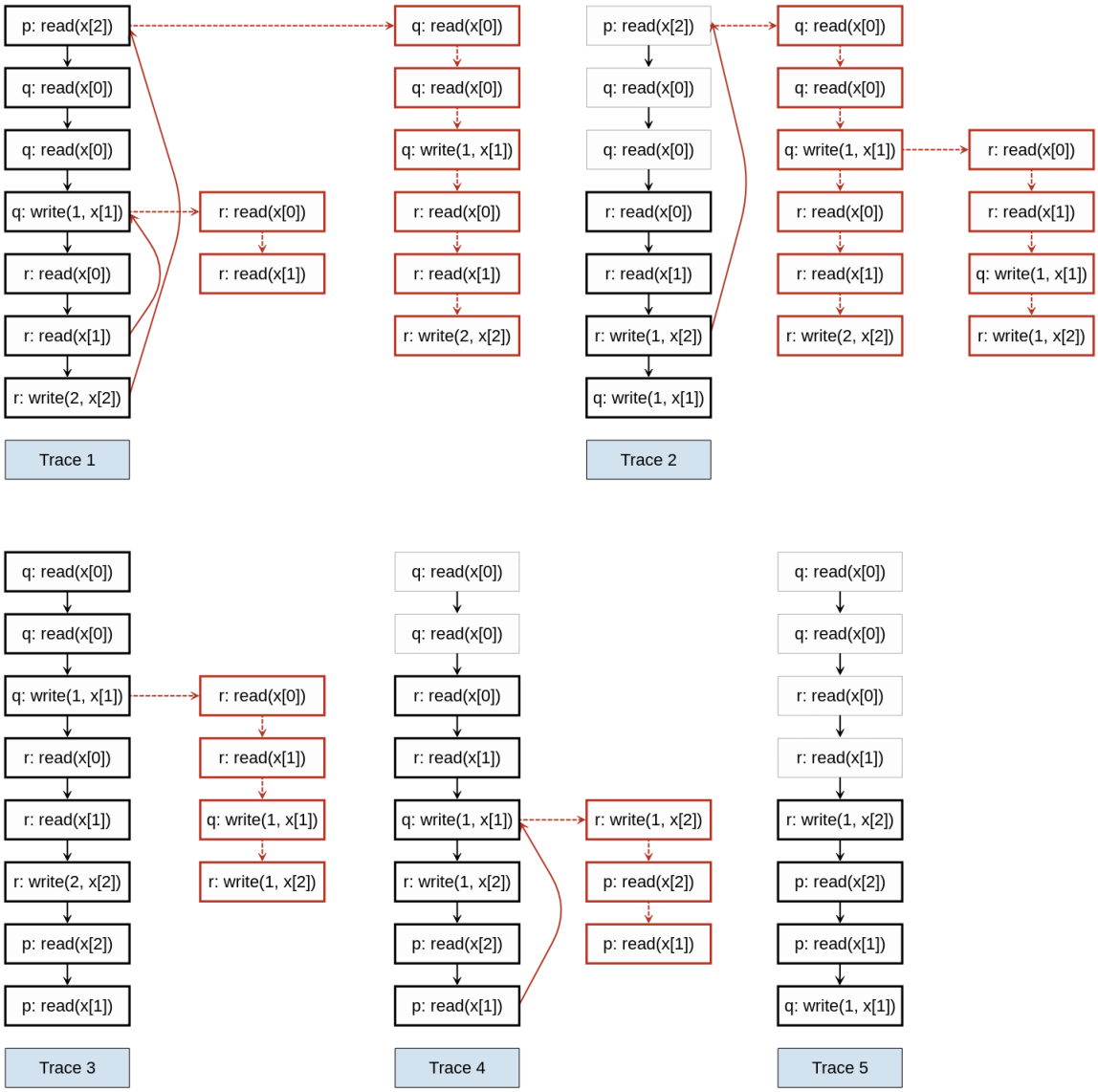
```
r:
R3 = x[0]
R4 = x[1]
assert(R3 == R4)
x[2] = R4 + 1
```

Σχήμα 5.1: *Lastzero* 2 example

και στο Planner, ο ελεγκτής θα περιμένει μια απάντηση είτε από έναν προγραμματιστή είτε από τον προγραμματιστή (αλγόριθμος 9). Όταν λαμβάνεται μια απάντηση από έναν προγραμματιστή, η ολοκληρωμένη ακολουθία εκτελέσεων που θα εξερευνηθεί θα προστεθεί στην ουρά του Planner και ο ελεγκτής θα συνεχίσει με τον βρόχο (Αλγόριθμος 7). Εάν λάβει απάντηση από το Planner, ο ελεγκτής θα ενημερωθεί το δέντρο εκτέλεσης T προσθέτοντας τα νέα δέντρα αφύπνισης που εισήχθησαν από το Planner και διαγράφοντας το επίθημα του την ακολουθία εκτέλεσης που μόλις ερευνήθηκε και δεν έχει ξυλουργικά δέντρα. Διαγράψαμε αυτό το μέρος για να έχουμε το το δέντρο εκτέλεσης περιέχει μόνο το τμήμα του κρατικού χώρου που είτε αυτή τη στιγμή διερευνάται είτε σχεδιάζεται να διερευνηθεί. Εάν δεν πρόκειται να διαγράψουμε αυτά τα επιθήματα, το μέγεθος του δένδρου εκτέλεσης τελικά θα είναι το μέγεθος του πλήρους χώρου κατάστασής μας.

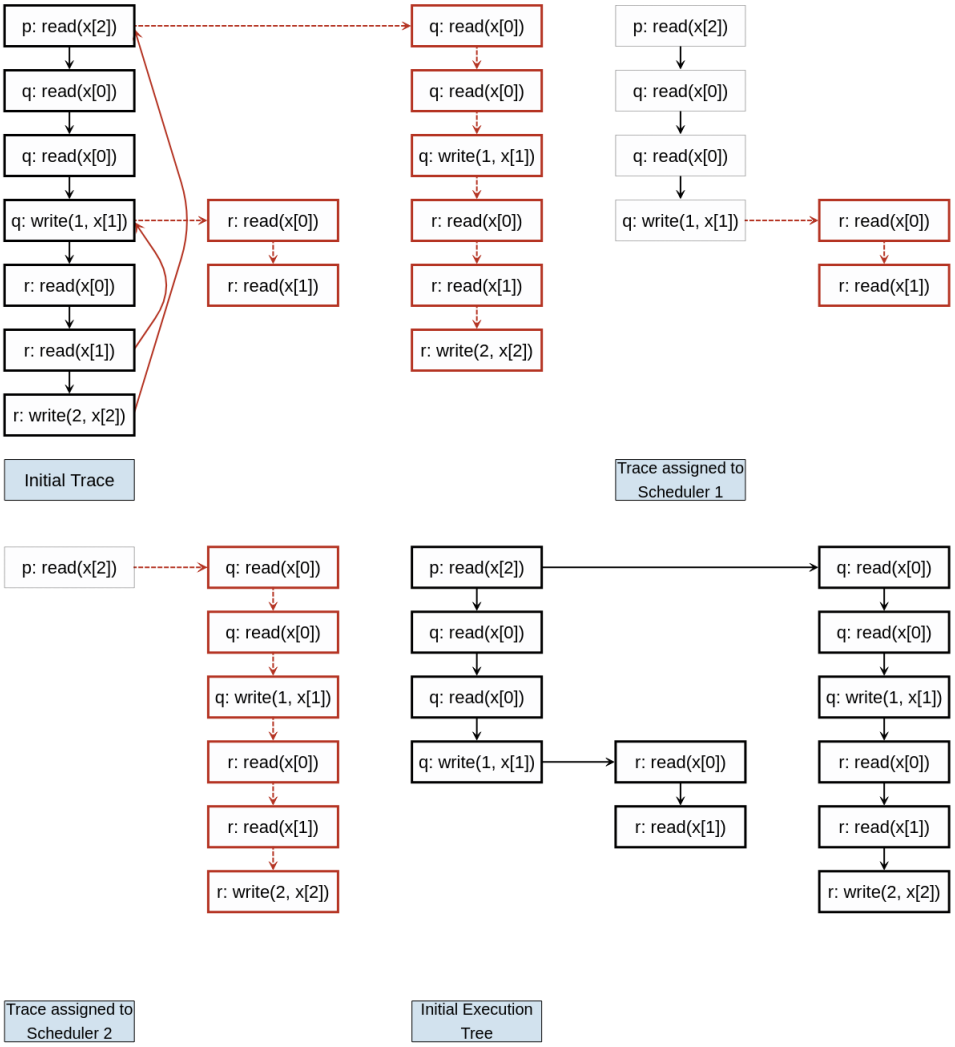
5.1.3 Παράδειγμα

Στο σχήμα 5.1 μπορούμε να δούμε τον ψευδοκώδικα του *lastzero* 2, εάν είχαμε μια σειρά 3 στοιχείων (αρχικά όλα τα στοιχεία έχουν μηδενική τιμή) και τρεις διαδικασίες. Η πρώτη διαδικασία (p) αναζητά τη συστοιχία για το μηδενικό στοιχείο με το υψηλότερο δείκτη. Οι άλλες δύο διαδικασίες αυξάνουν το καθορισμένο στοιχείο τους κατά μία τιμή 1.



Σχήμα 5.2: Interleavings explored by the sequential optimal-DPOR

Το σχήμα 5.2 αντιπροσωπεύει τα ίχνη που εξερευνήθηκαν κατά τη διάρκεια του διαδοχικού βέλτιστου-DPOR. Χρησιμοποιούμε ένα μαύρο τολμηρό ορθογώνιο για να αντιπροσωπεύσουμε ένα νέο συμβάν και ένα ελαφρύ ορθογώνιο για να υποδηλώσουμε ένα επαναλαμβανόμενο γεγονός. Οι συνεχείς κόκκινες άκρες αντιπροσωπεύουν τους αγώνες που εντοπίζονται και προγραμματίζονται. Οι κόκκινοι κόμβοι αντιπροσωπεύουν τις καταχωρήσεις δέντρου αφύπνισης σε κάθε ίχνος.



Σχήμα 5.3: Initial interleaving explored by the parallel optimal-DPOR

Το σχήμα 5.3 απεικονίζει το αρχικό βήμα του παράλληλου βέλτιστου-DPOR. Αρχικά διερευνάται μια αυθαίρετη αλληλουχία εκτέλεσης και στη συνέχεια οι φυλές της ανιχνεύονται και προγραμματίζονται με τη μορφή δέντρων αφύπνισης. Τα δέντρα αφύπνισης κατανέμονται σε διαφορετικά θραύσματα και όλα τα μη εκχωρημένα θραύσματα αντιστοιχίζονται στους αδρανείς προγραμματιστές. Επίσης, το δέντρο εκτέλεσης αρχικοποιείται με το τρέχον όριο εξερεύνησης.

Σε αυτό το παράδειγμα, ας υποθέσουμε ότι ο Scheduler 2 ολοκληρώνει πρώτα την εξερεύνηση του καθορισμένου ίχνους. Ο ελεγκτής θα τότε θα λάβετε το νέο διερευνημένο ίχνος και θα προσθέσετε αυτό το ίχνος στην ουρά του Planner. Δεδομένου ότι ο προγραμματιστής είναι αδρανής, αυτό το ίχνος θα σταλεί στο Planner για να ανιχνευθεί η κούρσα. Ενώ ο αγώνας ανιχνεύει αυτό το ίχνος, δεν θα υπάρξουν περισσότερες παρεμβολές σχεδιασμένος. Αυτό το ίχνος είναι ισοδύναμο με το 3ο ίχνος της διαδοχικής εκτέλεσης (Σχήμα 5.2). Παρατηρήστε εδώ ότι στον διαδοχικό αλγόριθμο αυτό το ίχνος είχε ένα επιπλέον δέντρο αφύπνισης. Αυτό το δέντρο αφύπνισης σχεδιάστηκε από το 2ο ίχνος του διαδοχικού αλγορίθμου, που δεν έχει ακόμη εντοπιστεί στο παράδειγμά μας. Επομένως, τα ίχνη 4 και 5 του διαδοχικού αλγορίθμου δεν μπορούν να είναι προγραμματισμένη από το 3ο ίχνος αλλά μόνο από το 2ο. Αυτό καθιστά εμφανές το κύριο ζήτημα της παραλληλισμού του βέλτιστου-DPOR: το πλήρες δέντρο αφύπνισης σε κάποια ακολουθία εκτέλεσης E δεν μπορεί να είναι γνωστό μέχρι να ολοκληρωθεί διερευνώντας όλες

Benchmark	Planning Time (%)	Exploration Time(%)	Sequential Time	Time for 4 Schedulers	Time for 24 Schedulers
readers 15	71.7%	28.3%	52m43.585s	98m28.251s	97m13.762s
lastzero 15	80.5%	19.5%	13m32.843s	24m98,312s	24m21,219s
readers 10	59.1%	40.9%	43.267s	59.699s	54.592s

Πίνακας 5.1: Parallel optimal-DPOR performance.

τις ακολουθίες εκτέλεσης που παραγγέλλονται πριν από το E, σύμφωνα με τη συνολική σειρά του κρατικού μας χώρου (Ορισμός 3.4)

5.1.4 Αξιολόγηση απόδοσης

Θα αξιοποιήσουμε τα ακόλουθα δύο συνθετικά σημεία αναφοράς την πρώτη προσπάθεια παραλληλισμού του βέλτιστου αλγορίθμου DPOR:

- *readers* N: This benchmark uses a writer process that writes a variable and N reader processes that read that variable.
- *lastzero* N: In this test we have $N+1$ processes that read and write on an array of $N+1$ size, which has all its values initialized with zero. The first process reads the array in order to find the zero element with the highest index. The other N processes read an array element and update the next one.

Το παράλληλο βέλτιστο DPOR δεν επιτυγχάνει κανένα είδος επιτάχυνσης. Πίνακας 5.1 κρατά πληροφορίες σχετικά με τον τρόπο εκτέλεσης της εφαρμογής μας σε διάφορες περιπτώσεις δοκιμών. Η εξερεύνηση και ο σχεδιασμός οι χρονικές σειρές δείχνουν ποιο ποσοστό του χρόνου του διαδοχικού αλγόριθμου δαπανώνται για την εξερεύνηση και τον προγραμματισμό αλληλεπιδράσεων αντίστοιχα. Αυτές οι μετρήσεις θα μας βοηθήσουν να εξηγήσουμε το λόγο αυτής της έλλειψης επιτάχυνσης.

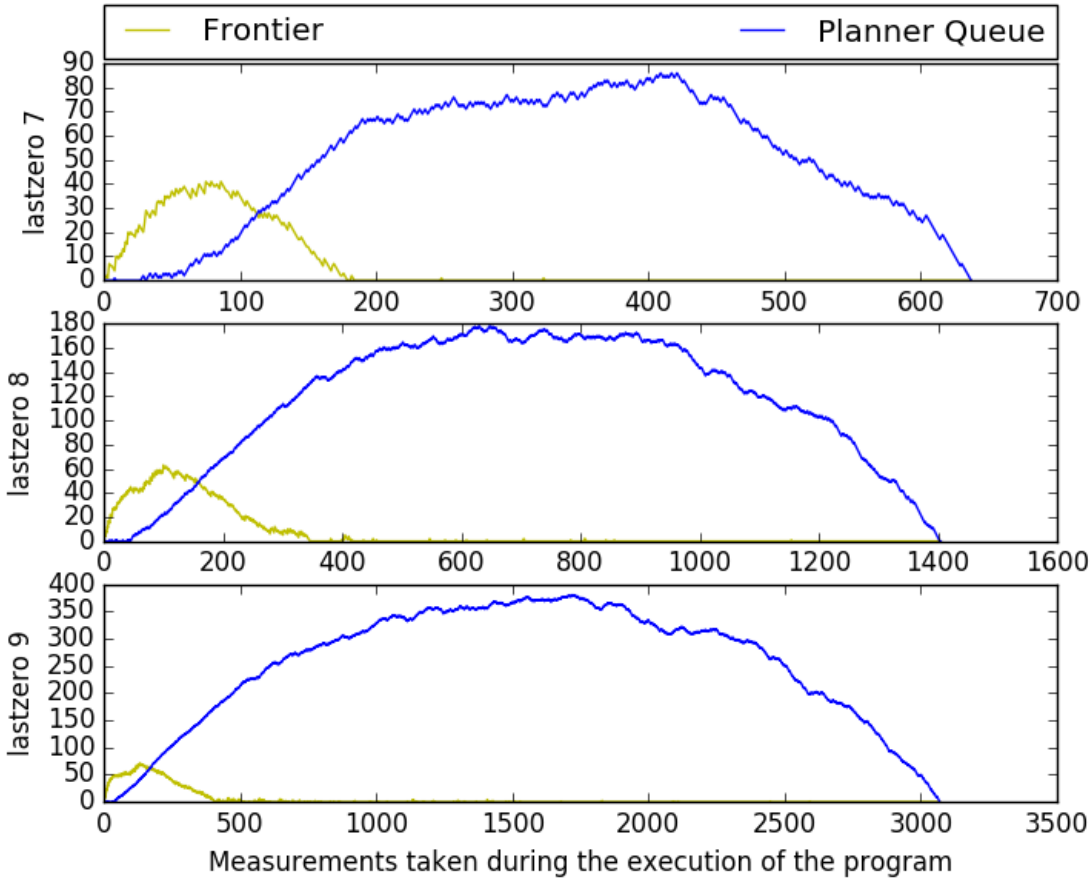
5.1.5 Ανάλυση απόδοσης

Αυτή η έλλειψη επιδόσεων δικαιολογείται από τους ακόλουθους λόγους.

Πρώτον, ας δούμε τα ποσοστά εξερεύνησης και προγραμματισμού του διαδοχικού αλγορίθμου, που παρουσιάζεται στον πίνακα 5.1. Ενώ το ποσοστό του χρόνου εξερεύνησης στην περίπτωση μιας μικρής δοκιμής η περίπτωση των αναγνωστών10 είναι γύρω στα 40%, παρατηρήσαμε ότι σε μεγαλύτερες περιπτώσεις δοκιμής αυτή η διαφορά κυμαίνεται μεταξύ 10%-30%. Αυτό σημαίνει ότι ακόμη και σε μια ιδανική εγκατάσταση, με μηδενική επιβάρυνση και άπειρους χρονοπρογραμματιστές, η επιτάχυνση μας δεν θα μπορούσε ποτέ να ξεπεράσει έναν παράγοντα των 1.429 (σε δοκιμαστική περίπτωση με χρόνο προγραμματισμού 70%, αν θεωρήσουμε ότι η φάση εξερεύνησης συμβαίνει αμέσως η επιτάχυνση θα ήταν $1.429 = \frac{100}{70}$, δεδομένου ότι η φάση σχεδιασμού θα έπρεπε ακόμα να πραγματοποιηθεί διαδοχικά). Αυτός ο λόγος, από μόνη της, καθιστά αδύνατο για τον αλγόριθμό μας να επιτύχει καλές επιδόσεις και κλιμάκωση.

Επιπλέον, ο ρυθμός με τον οποίο προγραμματίζονται νέες διεμπλοκές για εξερεύνηση οδηγεί σε ανεπαρκή αξιοποίηση των προγραμματιστών καθ 'όλη τη διάρκεια εκτέλεσης του αλγορίθμου. Συγκεκριμένα, παρατηρήσαμε ότι η εκτέλεση του αλγορίθμου DPOR μπορεί να χωριστεί σε τρεις διαδοχικές φάσεις. Στην πρώτη φάση, ο αγώνας ανίχνευσης μιας μοναδικής αλληλεπίδρασης παράγει ένα μεγάλο αριθμό παρεμβολών που πρέπει να εξερευνηθούν και ως εκ τούτου η εξερεύνηση Frontier είναι σχετικά μεγάλη. Αυτό

σημαίνει ότι οι προγραμματιστές έχουν αρκετή δουλειά και χρησιμοποιούνται. Κατά τη διάρκεια της δεύτερης φάσης, ο αγώνας που ανιχνεύει μια διεμπλοκή δημιουργεί ένα σχετικά μικρό αριθμό νέων διεμπλοκών που πρέπει να διερευνηθούν. Γενικά, σε όλη αυτή τη φάση, δεν δημιουργείται αρκετή δουλειά και οι προγραμματιστές τείνουν να παραμένουν αδρανείς. Στην τρίτη φάση, μόλις προγραμματιστούν νέες παρεμβολές, μέχρι να εξερευνηθεί ο πλήρης χώρος κατάστασης, οπότε ο αλγόριθμος τερματίζεται.



Σχήμα 5.4: Planner Queue and Frontier sizes for the execution of lastzero.

Αυτό το σχέδιο γίνεται ορατό από το σχήμα 5.4. Έχουμε εκτελέσει τον αλγόριθμό μας με τέσσερις χρονοπρογραμματιστές για τους τελευταίους 7, 8 και 9 δείκτες αναφοράς. Για να δημιουργήσετε αυτό το γράφημα, έχουμε λάβει μετρήσεις σχετικά με το μέγεθος της ουράς του Planner και του Frontier exploration των χρονοπρογραμματιστών. Οι μετρήσεις λαμβάνονται μετά την ολοκλήρωση της εξερεύνησης ή τον προγραμματισμό μιας διεμπλοκής. Σε αυτό το γράφημα, οι τρεις προαναφερθείσες φάσεις μπορούν να παρατηρηθούν σαφώς. Μετά τις πρώτες άκρα φάση, παρατηρούμε ότι η σχετική μεγάλο Frontier εξερεύνηση που έχει δημιουργηθεί, παίρνει " καταναλώνεται»σε ένα εξαιρετικά γρήγορο ρυθμό, οφείλεται στο γεγονός ότι έχουμε τέσσερις χρονοδιάγραμμα κατανάλωση από αυτήν, ενώ μόνο ένας Planner παράγει σε αυτό . επίσης, όπως αναφέραμε πριν, η εξερεύνηση ενός μόνο ίχνος είναι πολύ πιο γρήγορα από ό, τι ανίχνευσης αγώνα της. Συνεπώς, κατά τη δεύτερη φάση υπάρχουν πολύ λίγα ίχνη στο πλαίσιο της έρευνας αιχμής και, ως εκ τούτου, οι προγραμματιστές μείνετε υποχρησιμοποιούνται. μπορούμε επίσης να παρατηρήσετε από το γράφημα ότι αυτό το φαινόμενο εντείνεται, ενώ κράτους χώρο αυξάνει μας. ο κύριος λόγος πίσω από αυτό είναι το γεγονός ότι η κατάσταση χώρο ενός προγράμματος αυξάνεται, ο σχεδιασμός της παρεμβολής γίνεται ακόμα πιο αργά,

σε σύγκριση με την εξερεύνηση της παρεμβολής.

Για να καταστήσουμε τα πράγματα χειρότερα, ο αλγόριθμος μας έχει επίσης σημαντική επιβάρυνση. Όπως αναφέρθηκε προηγουμένως, επικοινωνία μεταξύ ο ελεγκτής και οι εργαζόμενοι μπορεί να είναι μια σημαντική συμφόρηση. Η παράλληλη πηγή-DPOR ασχολείται με την ανάθεση ένα κρατικό χώρο στους εργαζόμενους, ελαχιστοποιώντας την ανάγκη επικοινωνίας. Ωστόσο, σε αυτήν την προσπάθεια η χρήση του κεντρικού προγραμματιστή οδηγεί στους χρονοπρογραμματιστές να αναφέρουν ένα ίχνος πίσω στον ελεγκτή κάθε ένα μια εξερεύνηση φτάνει στο τέλος της.

5.2 Αποδοτικός παράλληλος optimal-DPOR

5.2.1 Βασική ιδέα

Όπως αναφέραμε στην προηγούμενη ενότητα, προσπαθώντας να αναπτύξουμε έναν παράλληλο αλγόριθμο βέλτιστου-DPOR με βάση την παράλληλη πηγή-DPOR έχει δύο βασικά ζητήματα.

Πρώτον, στον αλγόριθμο source-DPOR, το backtrack είναι ένα σύνολο διαδικασιών (Definition 3.3), που σημαίνει ότι περιέχει μεμονωμένα βήματα που ο αλγόριθμος πρόκειται να πάρει στο μέλλον.Στο παράλληλο αλγόριθμο, όπως αναφέρθηκε στο Κεφάλαιο 4, όταν μια καταχώρηση backtrack p σε μια ακολουθία εκτέλεσης E έχει αντιστοιχιστεί σε έναν προγραμματιστή, αυτός ο χρονοπρογραμματιστής θα κατέχει κάθε ίχνος που ξεκινά με το $E.p$ δηλ. Ο προγραμματιστής μπορεί να διερευνήσει την υποδίαση που έχει ρίζες στο E, χωρίς αναφορά στον ελεγκτή εκτός εάν δεν ξεπεραστεί ο προϋπολογισμός του). Ωστόσο, στη βέλτιστη-DPOR, εγγραφές backtrack είναι δέντρα που περιέχουν, ως κόμβους, ακολουθίες βημάτων που θα εξερευνηθούν. Ένας προγραμματιστής δεν μπορούσε να κατέχει το δέντρο λόγω του γεγονότος ότι στον Αλγόριθμο 2, νέα κομμάτια συνεχίζουν να εισάγονται στα δέντρα αφύπνισης και ως εκ τούτου, τα δέντρα δεν είναι πλήρη μέχρι να εξελιχθούν. Ένα δέντρο αφύπνισης είναι πλήρες, όταν έχουν εξερευνηθεί όλοι οι μικρότεροι κόμβοι με την παραγγελία του δέντρου ($p = min_{\prec}\{p \in wut(E)\}$). Σε κάθε σημείο καθ 'όλη την εκτέλεση του αλγορίθμου, υπάρχει μόνο ένα δέντρο αφύπνισης με αυτήν την ιδιότητα. Παρ 'όλα αυτά, κάθε φύλλο $\langle B, \prec \rangle$ παραμένει ένα φύλλο $insert_{[E]}(w, \langle B, \prec \rangle)$ [Abdu14]. This means that, a scheduler can own a subtree rooted at a leaf sequence of the wakeup tree. We are going to modify the concept of ownership as follows:

- Μια ακολουθία φύλλων μπορεί να επισημανθεί ως $owned$ στο δέντρο αφύπνισης ενός προγραμματιστή. Αυτό σημαίνει ότι αυτός ο χρονοπρογραμματιστής είναι επίσης ο ιδιοκτήτης κάθε κόμβου που βρίσκεται στο δευτερεύον φύλλο που έχει ρίζες σε αυτό το φύλλο. Για παράδειγμα, εάν το v είναι μια ακολουθία φύλλων ιδιοκτησίας στο $wut(E)$ κάποιου χρονοπρογραμματιστή, τότε αυτός ο χρονοπρογραμματιστής διαθέτει κάθε ακολουθία εκτέλεσης που έχει πρόθεμα $E.v$.

- Μια ακολουθία φύλλων χαρακτηρίζεται ως not_owned όταν κάποιος άλλος προγραμματιστής είναι υπεύθυνος για την αντίστοιχη δευτερεύουσα διάταξη.

- Όλοι οι άλλοι κόμβοι θεωρούνται $disputed$. Όταν μια αλληλουχία φύλλου εισάγεται κάτω από έναν αμφισβητούμενο κόμβο, αυτή η σειρά φύλλων θεωρείται αμφισβητούμενη.

Το δεύτερο ζήτημα έχει να κάνει με το γεγονός ότι η συνάρτηση $insert_{[E]}(v, wut(E))$ μπορεί να καταλήξει εισάγοντας μια διαφορετική ακολουθία (αλλά μια ισοδύναμη) από την v. Αυτό σημαίνει ότι δύο καταχωρήσεις φύλλων, αν και διαφορετικές, μπορεί να είναι

ισοδύναμη. Επομένως, η ιδιοκτησία των αμφισβητούμενων κόμβων δεν μπορεί να επιλυθεί ελέγχοντας απλώς εάν αυτοί οι κόμβοι υπάρχουν στο δέντρο εκτέλεσης. Ωστόσο, μπορούμε να τροποποιήσουμε το $insert_{[E]}(v, wut(E))$ για να εισαγάγουμε την ακολουθία εκτέλεσης μέσα στα δευτερεύοντα όρια του δένδρου εκτέλεσης, αντί των δέντρων αφύπνισης. Εάν μια αμφισβητούμενη ακολουθία μπορεί να εισαχθεί στο δέντρο εκτέλεσης, τότε αυτό σημαίνει ότι κανένας άλλος προγραμματιστής δεν έχει ανακαλύψει μια ισοδύναμη ακολουθία εκτέλεσης και ως εκ τούτου, η ιδιοκτησία αξιώνεται σε αυτή την ακολουθία εκτέλεσης. Διαφορετικά, αυτή η ακολουθία σημειώνεται ως not_owned.

5.2.2 Αλγόριθμος

Ο ελεγκτής έχει την ίδια λογική με αυτή από την παράλληλη πηγή-DPOR (αλγόριθμος 3). Ομοίως με τον αλγόριθμο πηγής-DPOR, ο ελεγκτής διατηρεί ένα $Frontier$, το οποίο είναι ένα σύνολο ακολουθιών εκτέλεσης E, και ένα δέντρο εκτέλεσης T, το οποίο περιέχει ως κλάδους τις ακολουθίες εκτέλεσης του $Frontier$. Για όσο χρονικό διάστημα υπάρχει ακολουθία εκτέλεσης στο $Frontier$ ($Frontier \neq \emptyset$), ο ελεγκτής θα κατατμήσει τις ακολουθίες $Frontier$ στις περισσότερες από τις ακολουθίες εκτέλεσης N. Στη συνέχεια, τον ελεγκτή θα προσπαθήσει να εκχωρήσει όλες τις μη ανατεθείσες ακολουθίες εκτέλεσης σε οποιονδήποτε χρονοπρογραμματιστή, με την αναπαραγωγή $explore_loop(E_c, Budget)$ λειτουργίες. Τέλος, θα μπλοκάρει μέχρι να λάβει μια απάντηση από έναν προγραμματιστή.

Algorithm 10: Βέλτιστος διαχωρισμός στα σύνορα - αλγόριθμος με δυνατότητα κλιμάκωσης

```
1  Function partition(Frontier, N)
2      for all E ∈ Frontier do
3          while total_owned_leaf_sequences(E) > 1 and size(Frontier) < N do
4              E' ← the smallest prefix of E that has a backtrack entry ;
5              v ← the smallest (w.r.t. ≺) owned leaf sequence ∈ wut(E');
6              E'_c ← a copy of E';
7              mark v as not_owned at wut(E');
8              {Prefix, v, Suffix} ← split_wut_at(v, wut(E'_c));
9              mark Prefix and Suffix as not_owned at wut(E'_c);
10             add E'_c to Frontier;
11     return Frontier;
```

Η φάση $partitioning$ (Αλγόριθμος 10) λειτουργεί αρκετά παρόμοια με τον αλγόριθμο 4. Η κύρια διαφορά εδώ είναι ότι όταν διανέμουμε αλληλουχίες από το δέντρο αφύπνισης, δεν τις προσθέτουμε απλώς στο σύνολο ύπνου της αρχικής αλληλουχίας εκτέλεσης, αλλά τις χαρακτηρίζουμε ως μη ιδιοκτησία. Ο λόγος πίσω από αυτό, μπορεί να δει στις γραμμές 20-21 του Αλγορίθμου 2. Αφού ο βέλτιστος-DPOR έχει ολοκληρώσει την εξερεύνηση ενός δέντρου αφύπνισης σε κάποια ακολουθία εκτέλεσης E, η διαδικασία έναρξης αυτού του δέντρου αφύπνισης προστίθεται στον ύπνο που έχει οριστεί στο E και οι ακολουθίες που ξεκινούν με αυτή τη διαδικασία απομακρύνονται από τα υπόλοιπα δέντρα αφύπνισης στο E. Προκειμένου να διατηρηθεί αυτή η συμπεριφορά ανέπαφη, τροποποιήσαμε τη φάση εξερεύνησης βέλτιστου DPOR, έτσι ώστε όταν συναντούμε κόμβους, των οποίων κάθε παιδί είναι not_owned, οι κόμβοι αυτοί προστίθενται στο σετ ύπνου και οι κατάλληλες ακολουθίες αφαιρούνται από το δέντρο αφύπνισης.

Ο βρόχος εξερεύνησης προγραμματιστή (Αλγόριθμος 11) παραμένει ως επί το πλείστον ο ίδιος. Έχουμε απλά αλλάξει τον τερματισμό ελέγξτε για να αντικατοπτρίσετε την

Algorithm 11: Scheduler Exploration Loop - Βελτιστοποιήσιμο βέλτιστο-DPOR

```
1  Function explore_loop(E_0, Budget)
2      StartTime ← get_time();
3      E ← E_0;
4      repeat
5          E' ← explore(E);
6          E' ← plan_more_interleavings(E');
7          E ← get_next_execution_sequence(E');
8          CurrentTime ← get_time();
9      until CurrentTime − StartTime > Budget or (ownership(E) ≠
         owned and WuT(E) has no owned sequences);
10     send E to Controller ;
```

αλλαγή στην έννοια της ιδιοκτησίας. Δεν επιτρέπεται εδώ, ότι όταν το $get_next_execution_sequence(E')$ βρίσκει κόμβους που έχουν not_owned ιδιοκτησία, η αρχική διαδικασία p αυτών των κόμβων προστίθεται στο κατάλληλο σετ ύπνου και οι ακολουθίες της φόρμας αρχίζουν $p.w$ αφαιρούνται από το κατάλληλο δέντρο αφύπνισης. Με άλλα λόγια, αυτό είναι υπεύθυνο για τις γραμμές 20-21 του Αλγορίθμου 2.

Ο ψευδοκώδικας του χειρισμού της απόκρισης του προγραμματιστή παραμένει ο ίδιος με τον αλγόριθμο 6. Ωστόσο, η συνάρτηση $update_execution_tree(E, T)$ αλλάζει. Αυτή η λειτουργία επαναλαμβάνει ταυτόχρονα την ακολουθία εκτέλεσης και το δέντρο εκτέλεσης, και για κάθε αμφισβητούμενο φύλλο δέντρου αφύπνισης που βρίσκεται στην ακολουθία εκτέλεσης, καλείται μια τροποποιημένη έκδοση του $insert_{[E]}(v, wut(E))$. Αυτή η έκδοση προσπαθεί να εισάγει τα φύλλα από τα δέντρα αφύπνισης της ακολουθίας εκτέλεσης στο δέντρο εκτέλεσης. Εάν πετύχει, αξιώνει την κυριότητα πάνω σε αυτό το φύλλο και πάνω σε κάθε ισοδύναμο φύλλο. Αν δεν γίνει εισαγωγή, αυτό σημαίνει ότι κάποιος άλλος προγραμματιστής έχει διεκδικήσει την κυριότητα πάνω σε ένα ισοδύναμο φύλλο και ως εκ τούτου αυτό το φύλλο έχει επισημανθεί ως not_owned στην ακολουθία εκτέλεσης E.

Όταν ενημερώνουμε το δέντρο εκτέλεσης, διαγράφουμε επίσης τμήματα από το δέντρο εκτέλεσης που έχουν ήδη ερευνηθεί. Τέλος, ο μηχανισμός εξισορρόπησης φορτίου παραμένει ο ίδιος με τον αλγόριθμο παράλληλης πηγής-DPOR.

Κεφάλαιο 6

Λεπτομέρειες Υλοποίησης

Στο Κεφάλαιο 4 περιγράψαμε πώς οι αλγόριθμοι Source-DPOR και Optimal-DPOR μπορούν να παραλληλιστούν χρησιμοποιώντας πολλαπλούς χρονοπρογραμματιστές να διερευνήσουν ταυτόχρονα διαφορετικές παρεμβολές. Προκειμένου να επιτευχθεί αυτό, πρέπει να γίνουν ορισμένες τροποποιήσεις. Εδώ συνοψίζουμε τις απαραίτητες τροποποιήσεις.

Θα πρέπει να περιγράψουμε συνοπτικά τον τρόπο με τον οποίο ο χρονοπρογραμματιστής λειτουργεί στη διαδοχική υλοποίηση. Ο χρονοπρογραμματιστής ξεκινά διερευνώντας πλήρως μια αυθαίρετη παρεμβολή (με μια μέγιστη αλληλουχία E), μέσω της συνάρτησης *explore/1*. Συνεχίζει καλώντας το *plan_more_interleaving/1*, για να ανιχνεύσει τους αγώνες της διερευνηθείσας αλληλεπίδρασης και να σχεδιάσει την εξερεύνηση μελλοντικές παρεμβολές σύμφωνα με τη λογική του χρησιμοποιούμενου αλγορίθμου. Ας υποθέσουμε ότι πρέπει να εξερευνήσουμε μια ακολουθία $E'.p$, όπου E' είναι ένα πρόθεμα των E και p μια διαδικασία. Η επόμενη κλήση του *explore/1* θα επαναφέρει όλες τις διαδικασίες του ηθοποιού, για να τους επαναφέρει στην αρχική τους κατάσταση, και τότε θα επαναλάβει το πρόθεμα E' για να αναδημιουργήσει την παγκόσμια κατάσταση στο E', χωρίς να αναδημιουργήσει εντελώς τα συμβάντα στο πρόθεμα. Αφού ολοκληρωθεί η επανάληψη, οι διαδικασίες στο backtrack (ή στο δέντρο αφύπνισης) θα προγραμματιστούν και τελικά τα υπόλοιπα γεγονότα θα προγραμματιστούν αυθαίρετα έτσι ώστε να μην υπάρχουν ενεργοποιούνται περισσότερες διαδικασίες. Ο προγραμματιστής στη συνέχεια θα προσπαθήσει να προγραμματίσει περισσότερες παρεμβολές. Τελειώσαμε όταν δεν υπάρχουν άλλες παρεμβολές που πρέπει να εξερευνήσουμε (ή μέχρι να βρεθεί σφάλμα αν η επιλογή *keep_going* είναι ρυθμισμένη στο *false*).

Ας περιγράψουμε, επίσης, τους κύριους τύπους δεδομένων που χρησιμοποιούνται στον προγραμματιστή:

- $event()$: αντιστοιχεί στο συμβάν e μιας διαδικασίας p, σύμφωνα με τη σημείωσή μας. Περιέχει το Erlang Pid (process identifier) of the actor process p, as well as information about the code (e.g. the BIF) of this specific event.

- $\#event_tree\{\}$: αναφέρεται είτε στο backtrack είτε στο δέντρο αφύπνισης σε ένα συγκεκριμένο σημείο.

- $\#trace_state\{\}$: περιέχει πληροφορίες σχετικά με ένα βήμα εκτέλεσης μιας ακολουθίας εκτέλεσης E, such as the backtrack (or the wakeup tree) and the sleep set at this point.

- $\#scheduler_state\{\}$ ένα αρχείο που περιέχει πληροφορίες σχετικά με την κατάσταση του χρονοπρογραμματιστή such as the algorithm used and, most importantly, the current trace, which is a list of $\#trace_state\{\}$ records. This list roughly corresponds to the execution sequence E as defined in our framework.

6.1 Επίλυση προβλημάτων με διεργασίες

Στο διαδοχικό Concuerror, για κάθε διαδικασία ενός δοκιμασμένου προγράμματος, ο χρονοπρογραμματιστής χρειάζεται να δημιουργήσει μόνο μία διαδικασία. Στη συνέχεια ο χρονοπρογραμματιστής θα ελέγχει την εκτέλεση των διαδικασιών ενός προγράμματος για την παραγωγή διαφορετικών παρεμβολών. Προκειμένου να διερευνηθούν ταυτόχρονα διαφορετικές διεμπλοκές, για κάθε διαδικασία στο δοκιμασμένο πρόγραμμα, κάθε παράλληλος προγραμματιστής πρέπει να δημιουργήσει τη δική του διαδικασία. Αυτό τεχνικά σημαίνει ότι πρέπει να έχουμε διαφορετικές διαδικασίες Erlang που αντιστοιχούν στην ίδια διαδικασία του δοκιμασμένου προγράμματος. Οι διαδικασίες Erlang χαρακτηρίζονται από το Pid, το οποίο είναι παγκοσμίως μοναδικό. Το Pid μιας διαδικασίας χρησιμοποιείται επίσης στο Concuerror για να προσδιορίσει μια διαδικασία και, συνεπώς, να χαρακτηρίσει ένα ίχνος. Κατά τη μεταφορά των ιχνών μεταξύ των προγραμματιστών οποιοδήποτε Pid που βρέθηκε οπουδήποτε στο ίχνος θα πρέπει να αλλάξει ώστε να αντικατοπτρίζει τα Pids των διαφορετικών χρονοπρογραμματιστών.

Αυτό σημαίνει ότι πρέπει να δημιουργηθεί μια αντιστοίχιση μεταξύ των Pids των διαφορετικών χρονοπρογραμματιστών. Αυτή η χαρτογράφηση μπορεί να καθοριστεί μέσω το *symbolic names* που ο Concuerror εκχωρεί στη δοκιμασμένη διαδικασία με την ακόλουθη λογική:

$$Symbol(p) = \begin{cases} \text{``}P\text{''} & \text{if } p \text{ is the initial process} \\ Symbol(q).i & \text{if } p \text{ is the } i_{th} \text{ child of } q \end{cases}$$

Ωστόσο, η δημιουργία τέτοιων αντιστοιχιών δεν αρκεί για να εγγυηθεί ότι ένα ίχνος μπορεί να μεταφερθεί μεταξύ διαφορετικών χρονοπρογραμματιστών. Είναι σημαντικό ότι η ίδια ακολουθία εκτέλεσης οδηγεί στην ίδια παγκόσμια κατάσταση ανεξάρτητα από τον προγραμματιστή που την εξερευνά. Συγκεκριμένα, Erlang δίνει τη δυνατότητα να συγκρίνετε τα Pids. Για παράδειγμα, η παραγγελία δύο Pids θα μπορούσε να αλλάξει το αποτέλεσμα ενός κλάδου σε ένα πρόγραμμα. Αυτό θα μπορούσε οδηγούν στο ίδιο ίχνος που οδηγεί σε διαφορετικές παγκόσμιες πολιτείες σε διαφορετικούς χρονοπρογραμματιστές. Επιπλέον, μέσω της χρήσης του BIF *pid_to_list/1*, ένα Pid θα μπορούσε να υπάρχει με τη μορφή μιας συμβολοσειράς σε κάποιο ίχνος και ως αποτέλεσμα θα έπρεπε να προσπαθήσουμε και να αναλύσουμε κάθε συμβολοσειρά σε ένα ίχνος για να ελέγξετε αν αναφέρεται το Pid.

Λύπουμε αυτά τα ζητήματα, έχοντας κάθε δρομολογητή να τρέξει στον δικό του κόμβο Erlang. Είναι δυνατό για δύο διαδικασίες, που βρίσκονται σε διαφορετικούς κόμβους, να έχει το ίδιο τοπικό Pid. Ενώ προσπαθούμε να εφαρμόσουμε έναν τέτοιο μηχανισμό, αντιμετωπίσαμε δύο βασικά ζητήματα:

- Erlang does not give the option to request specific Pids. Nevertheless, the Erlang VM of a node assigns Pids in a sequential ordering. For example, after spawning a process with $< 0.110.0 >$ as a local Pid, then next process spawned in that node would have a Pid of $< 0.111.0 >$. We can use this to preemptively spawn processes on different nodes with the same Pid, by creating a *process_spawner*. We require that nothing besides our schedulers runs on our nodes and therefore, there will be no interference with sequence of the spawned Pids on the node. Firstly, we must reach a consensus between the different schedulers as to the initial local Pid. This consensus can be achieved by having each scheduler send to the *process_spawner* the first available local Pid in their node (we can get this by spawning a dummy process). The *process_spawner* chooses the maximum local Pid and sends it to all the schedulers. The schedulers can then spawn a process with this maximum Pid by spawning and killing dummy processes until they reach the requested Pid. Then they can spawn a specified (by the user) amount or processes preemptively. The i_{th} processes spawned this way on

different nodes, will all have the same local Pid. Thanks to that, we can have different processes on different nodes with the same local Pid that corresponds to the same symbolic process.

- Simply sending a trace between schedulers on different nodes will result in the Erlang VM changing every Pid on the trace to their global values. Those values, however, are unique. We can avoid this by transforming every Pid on that trace to a string (by using the *pid_to_list/1* BIF) before sending the trace. When we send the transformed trace the VM will not interfere with the transformed Pids. The local Pids can then be recovered from the receiving scheduler by using the *list_to_pid/1* BIF. These local Pids will refer to processes with the same symbolic name on different nodes.

6.2 Αλλαγή του μηχανισμού Replay

Ακόμη και μετά την επίλυση του προβλήματος Pid, η επανάληψη των ιχνών σε διαφορετικούς χρονοπρογραμματιστές δεν πρόκειται να λειτουργήσει. Υπάρχουν δύο βασικοί λόγοι πίσω από αυτό.

Πρώτον, κατά τη διάρκεια εκτέλεσης ορισμένων συμβάντων (όπως γεγονότα που σχετίζονται με πίνακες ETS ή spawning), ο Concuerror χρησιμοποιεί διάφορα ETS πίνακες για την παρακολούθηση συγκεκριμένων πληροφοριών. Όταν ο Concuerror εξερευνά ένα ίχνος στη λειτουργία επανάληψης, βεβαιώνεται ότι είναι τέτοιο υπάρχουν πληροφορίες και οδηγεί τις δοκιμασμένες διαδικασίες στην κατάλληλη κατάσταση. Επομένως, όταν ο Concuerror διερευνά ένα εντοπίζει δημιουργεί κάποιες παρενέργειες στην παγκόσμια κατάστασή του. Αυτές οι παρενέργειες δεν θα υπάρχουν έναν διαφορετικό κόμβο Erlang, αφού οι πίνακες ETS δεν μοιράζονται μεταξύ των κόμβων.

```
1    Pid = spawn(fun() ->
2                    receive
3                        exit ->
4                            ok
5                    end
6                end),
7    Lambda =
8        fun() ->
9            Pid ! exit
10       end.
```

Listing 6.1: Environment variables

Δεύτερον, οι λειτουργίες lambda που ορίζονται από το χρήστη μπορούν να έχουν μερικές μεταβλητές περιβάλλοντος. Η αξία αυτών των μεταβλητών είναι αμετάβλητη αφού οριστεί η λειτουργία λάμδα. Στην καταχώριση 6.1, εάν ένα ίχνος περιέχει ένα συμβάν που εφαρμόζει αυτή τη λειτουργία, αυτό το συμβάν δεν θα μπορεί να επαναληφθεί σωστά από χρονοπρογραμματιστή διαφορετικό από αυτόν που έχει το δημιούργησε, καθώς η μεταβλητή περιβάλλοντος Pid δεν μπορεί να αλλάξει. Ο μόνος λογικός τρόπος για να επιλυθεί αυτό είναι να αλλάξτε τον τρόπο λειτουργίας της αναπαραγωγής.

Συγκεκριμένα, πρέπει να έχουμε δύο διαφορετικές λειτουργίες επανάληψης: *pseudo* και *actual*. Η αναπαραγωγή ψευδολογίου χρησιμοποιείται όταν αναπαραγωγή ιχνών που δημιουργήθηκαν από τον ίδιο χρονοπρογραμματιστή και λειτουργεί ακριβώς όπως η αναπαραγωγή του διαδοχικού Κατασκοπεία. Η πραγματική επανάληψη επαναδημιουργεί τα συμβάντα και τις παρενέργειες ενός ίχνους και χρησιμοποιείται για επαναλαμβάνοντας τις παρεμβολές που έχουν ληφθεί από άλλους προγραμματιστές. Στις ενσωματωμένες εκδηλώσεις, το επιτυγχάνουμε αυτό, θέτοντας ένα *actual_replay* flag to *true* and by making changes on how the *concuerror_callback* module handles those replays. Σε άλλα συμβάντα,

ορίσαμε την τιμή *event_info* σε *undefined*, αναγκάζοντας τα συμβάντα αυτά να αναδημιουργηθούν.

Κεφάλαιο 7

Αξιολόγηση Παραλληλοποίησης

Σε αυτό το κεφάλαιο, θα παρουσιάσουμε τα αποτελέσματα απόδοσης των παράλληλων πηγών μας - DPOR και βέλτιστων αλγορίθμων DPOR, υλοποιήθηκε στο Concuerror. Πρόκειται να αξιολογήσουμε τα αποτελέσματά μας σε ορισμένα «συνήθη» και συνθετικά σημεία αναφοράς που είναι κανονικά χρησιμοποιείται για να δοκιμάσει τους αλγορίθμους DPOR, καθώς και ένα πραγματικό πρόγραμμα erlang. Τέλος, θα προσπαθήσουμε να εξηγήσουμε τη συμπεριφορά του παράλληλου προγράμματος όπως αντικατοπτρίζεται σε αυτά χάρτες, σχεδιάζοντας συναφή συμπεράσματα όποτε είναι δυνατόν.

7.1 Επισκόπηση των Benchmarks

Πρώτα θα δώσουμε μια σύντομη επισκόπηση των δοκιμασμένων προγραμμάτων:

- $indexer\ N$: This test uses a Compare and Swap (CAS) primitive instruction to check if a specific element of a matrix is set to 0 and if so, set it to a new value. This is implemented in Erlang by using ETS tables and specifically the $insert_new/2$ function. This function returns false if the key of the inserted tuple exists (the entry is set to 0) or it inserts the tuple if the key is not found. N refers to the number of threads that are performing this function.

- $readers\ N$: This benchmark uses a writer process that writes a variable and N reader processes that read that variable.

- $lastzero\ N$: In this test we have $N + 1$ processes that read and write on an array of $N + 1$ size, which has all its values initialized with zero. The first process reads the array in order to find the zero element with the highest index. The other N processes read an array element and update the next one.

- $rush\ hour$: a program that uses processes and ETS tables to solve the Rush Hour puzzle in parallel, using A*search. Rush hour is a complex but self-contained (917 lines of code) program.

7.2 Αποτελέσματα Implementation

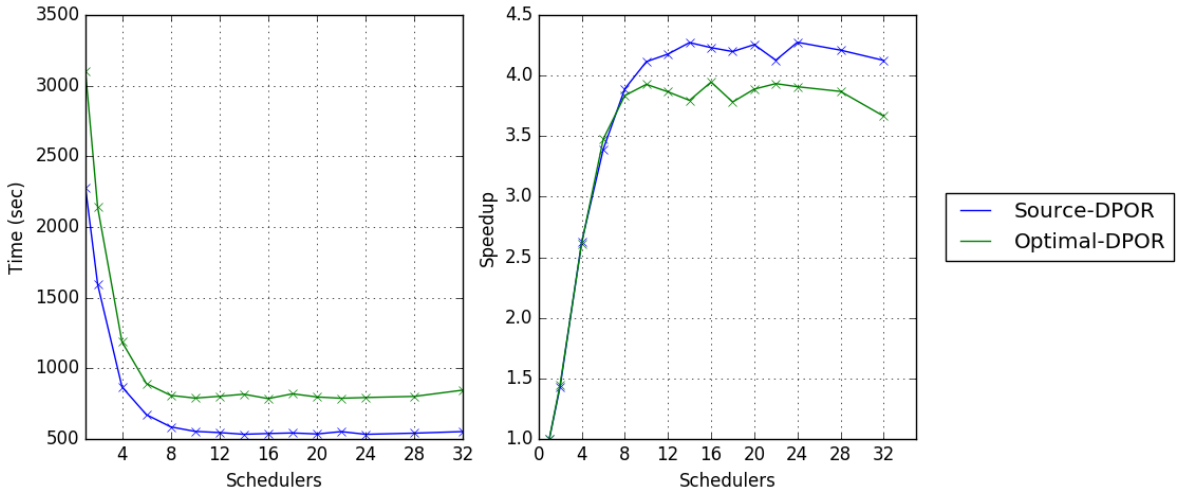
Τα σημεία αναφοράς έγιναν σε έναν πολυεπεξεργαστή με 64 πυρήνες AMD Opteron 6276 (2,3 GHz), 126 GB μνήμης, τρέξιμο Linux 4.9.0-8amd64 και τρέχοντας την μεταγενέστερη έκδοση Erlang (Erlang / OTP 21). Ενώ εκτελούμε τις δοκιμές μας, χρησιμοποιούμε το - flag$keep_going$ για να συνεχίσετε την εξερεύνηση του χώρου κατάστασής μας, ακόμα και μετά τη διαπίστωση σφάλματος. Το κάνουμε αυτό ώστε να μπορούμε να αξιολογήσουμε πόσο γρήγορα διερευνάται ο πλήρης χώρος της κατάστασης.

Benchmark	Traces for source-DPOR	Traces for optimal-DPOR	Time for source-DPOR	Time for parallel source-DPOR with 1 scheduler	Time for optimal-DPOR	Time for parallel optimal-DPOR with 1 scheduler
lastzero 11	60073	7168	49m8.510s	53m59.169s	14m8.266s	17m50.494s
indexer 17	262144	262144	186m8.136sec	205m24.872sec	193m54.320sec	252m21.033sec
readers 15	32768	32768	37m68.865s	46m28.711s	51m40.792s	67m50.643s
rush hour	46656	46656	52m36.889s	56m3.521s	51m11.184s	58m32.962s

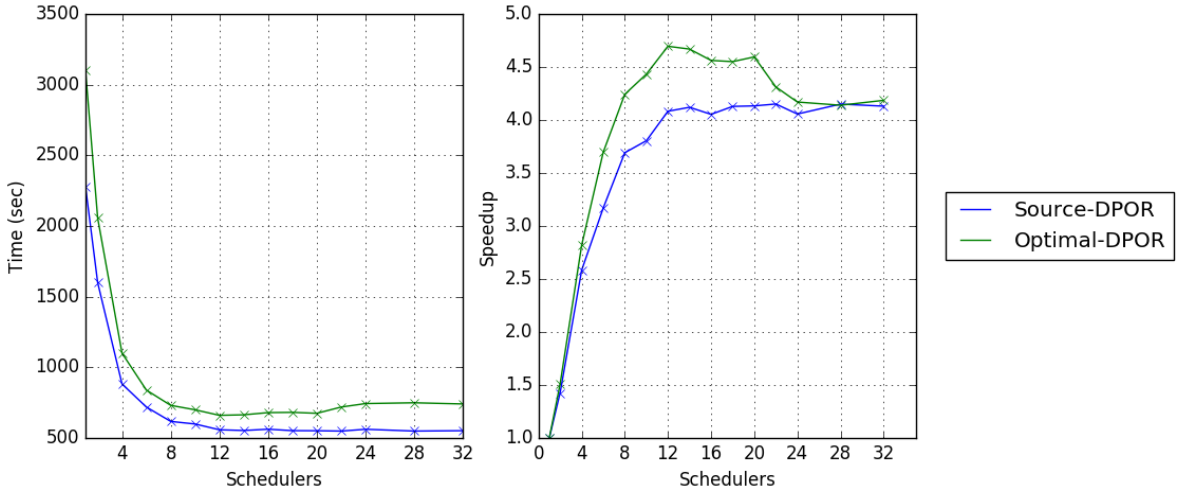
Πίνακας 7.1: Sequential performance of source-DPOR and optimal-DPOR on four benchmarks.

Ο πίνακας 7.1 περιέχει πληροφορίες σχετικά με τα διερευνηθέντα ίχνη και τη διάρκεια αυτών διερευνήσεις για τις διαδοχικές εκδόσεις του πηγαίου κώδικα DPOR και optimal-DPOR, καθώς και για τη διάρκεια των παράλληλων εκδόσεων με έναν προγραμματιστή, μια τιμή κατακερματισμού ενός και έναν προϋπολογισμό 10000.

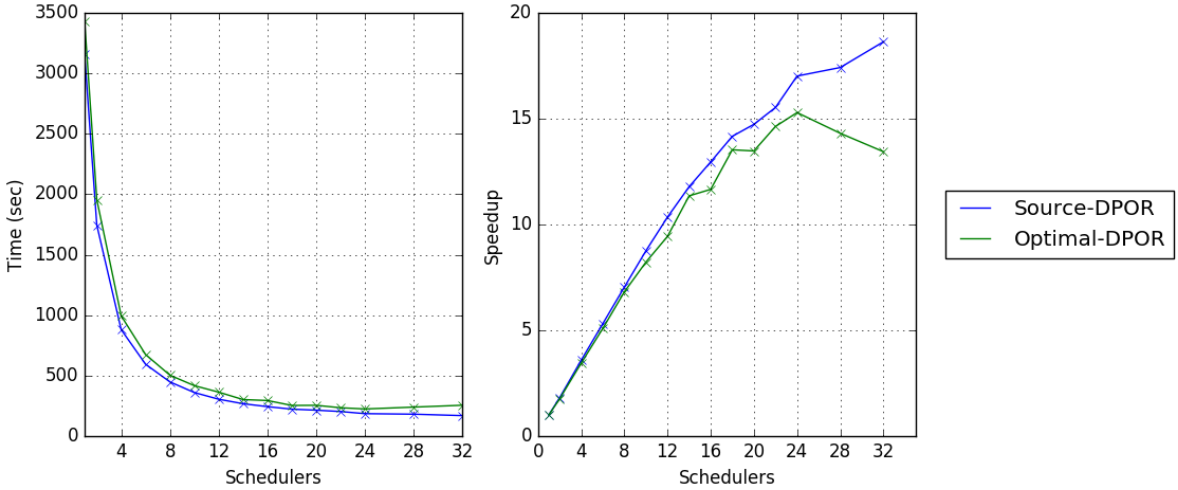
Αφήνει επίσης να παρουσιάσει γραφήματα που απεικονίζουν τον χρόνο εκτέλεσης και την επιτάχυνση ($\frac{T_{serial}}{T_{parallel}}$) of the source-DPOR and optimal-DPOR algorithms για διαφορετικούς αριθμούς χρονοπρογραμματιστών και για διάφορες περιπτώσεις δοκιμής.



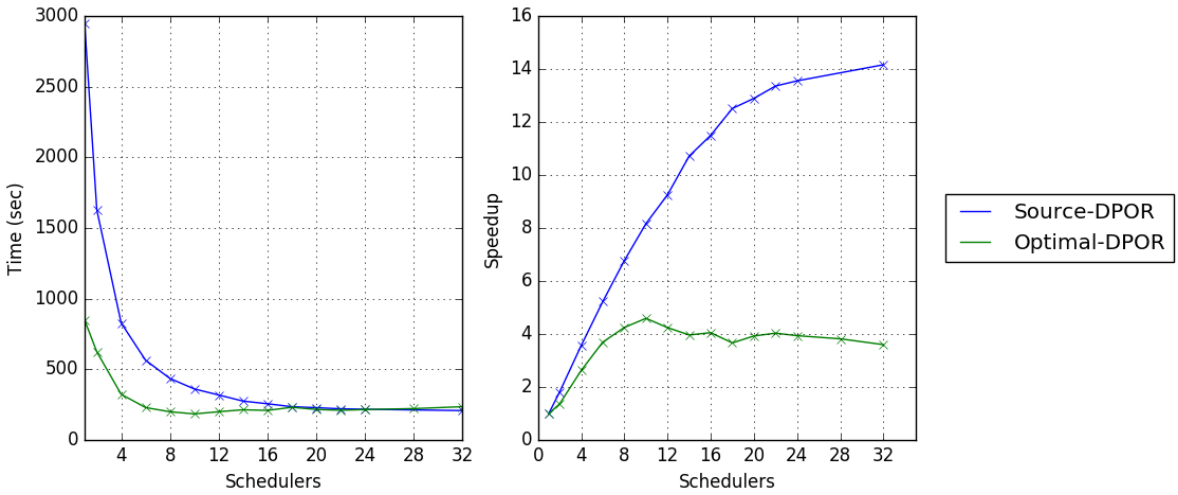
Σχήμα 7.1: Performance of readers 15 with Budget of 10000.



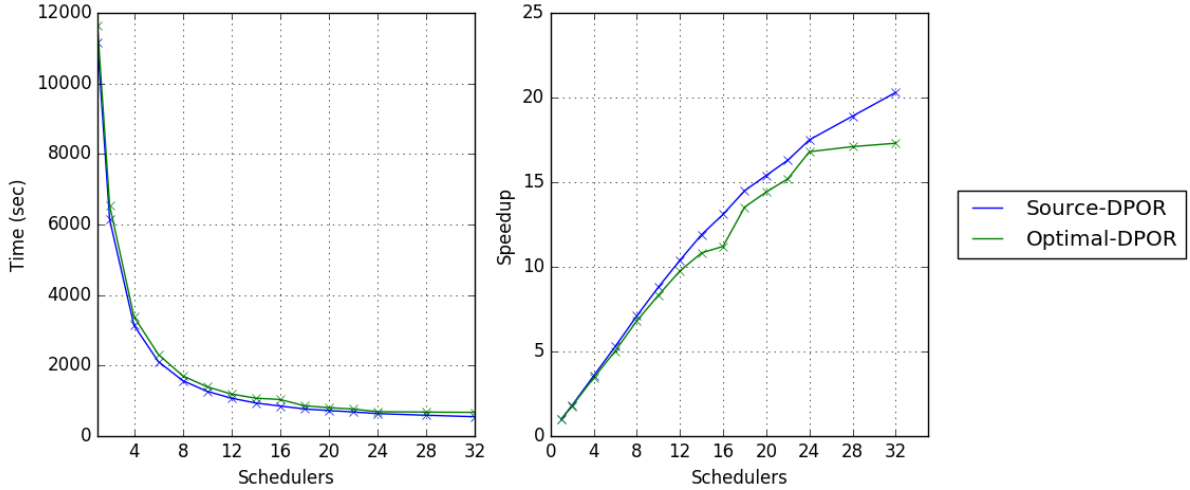
Σχήμα 7.2: Performance of readers 15 with budget of 30000.



Σχήμα 7.3: Performance of rush hour with Budget of 10000 for source and 30000 for optimal.



Σχήμα 7.4: Performance of lastzero 11 with Budget of 10000 for source and 30000 for optimal.



Σχήμα 7.5: Performance of indexer 17 with Budget of 10000 for source and 30000 for optimal.

7.3 Ανάλυση Απόδοσης

Όπως μπορούμε να δούμε στα γραφήματα, η παράλληλη εφαρμογή μας, τόσο της πηγής-DPOR όσο και της βέλτιστης-DPOR, μειώνει σημαντικά τη δοκιμή χρόνο των δοκιμαστικών μας περιπτώσεων. Συγκεκριμένα, στις περιπτώσεις ώρας αιχμής και ευρετηρίου, η πηγή-DPOR παρέχει μια ταχύτητα με συντελεστή περίπου 1,8 για 2 προγραμματιστές και από 3,5 για 4 προγραμματιστές και 7,1 για 8 προγραμματιστές. Σε αυτά τα σημεία αναφοράς, το βέλτιστο DPOR παρέχει μια ταχύτητα περίπου 1,7, 3,4 και 6,8 για 2, 4 και 8 χρονοπρογραμματιστές αντίστοιχα. Αυτό κάνει τις παράλληλες υλοποιήσεις μας εξαιρετικά χρήσιμες σε προσωπικούς υπολογιστές.

Επιπλέον, η πηγή-DPOR επιτυγχάνει μια αξιοπρεπή επεκτασιμότητα σε δοκιμαστικές περιπτώσεις, όπως η ώρα αιχμής, το lastzero 11 και το indexer 17. Ιδιαίτερα, με 32 χρονοπρογραμματιστές καταφέρνουμε να επιταχύνουμε το Concuerror κατά παράγοντα 18,6 για την ώρα αιχμής και 20,3 για το indexer 17. Ωστόσο, μέσα από τα διαγράμματα κλιμάκωσης μπορούμε να παρατηρήσουμε ότι με περισσότερους χρονοπρογραμματιστές η κλιμάκωση αυτών των δοκιμαστικών περιπτώσεων αρχίζει να μειώνεται. Αυτό συμβαίνει επειδή εμείς χρησιμοποιούν έναν κεντρικό ελεγκτή, ο οποίος γίνεται εμπόδιο, δεδομένου ότι όσο περισσότερο οι προγραμματιστές, το μεγαλύτερη πιθανότητα ο Ελεγκτής να είναι απασχολημένος και συνεπώς να μην μπορεί να διανείμει εργασία. Ο πιο σημαντικός λόγος πίσω από αυτή την πτώση στην επεκτασιμότητα όμως είναι το γεγονός ότι με περισσότερους διαθέσιμους χρονοπρογραμματιστές οι ακολουθίες εκτέλεσης των συνόρων χωρίζονται με μεγαλύτερη λεπτομέρεια και ως εκ τούτου τα υποκείμενα του κρατικού χώρου που αντιστοιχούν σε κάθε χρονοπρογραμματιστή είναι επίσης πιο λεπτές. Μια πιο εκλεπτυσμένη αποστολή οδηγεί τους προγραμματιστές να διερευνήσουν το δικό τους αναθέτει τα γκρουπ γρηγορότερα και έχοντας περισσότερους αγώνες που βρέθηκαν εκτός των προκαθορισμένων υποστρωμάτων τους. Κατά συνέπεια, η επικοινωνία με τον ελεγκτή είναι πιο συχνή. Η κατάσταση αυτή είναι προβληματική για δύο λόγους: η επικοινωνία μεταξύ ενός προγραμματιστή και του ελεγκτή έχει μη αμελητέα γενικά έξοδα και όταν το ποσό του οι επικοινωνίες μεταξύ των προγραμματιστών και του ελεγκτή αυξάνονται, ο ελεγκτής καθίσταται ακόμη μεγαλύτερη συμφόρηση. Ως εκ τούτου, καταλήγουμε στο συμπέρασμα ότι μια δοκιμαστική περίπτωση δεν μπορεί να κλιμακωθεί πέρα από ένα συγκεκριμένο σημείο με βάση τον αριθμό των παρεμβολών

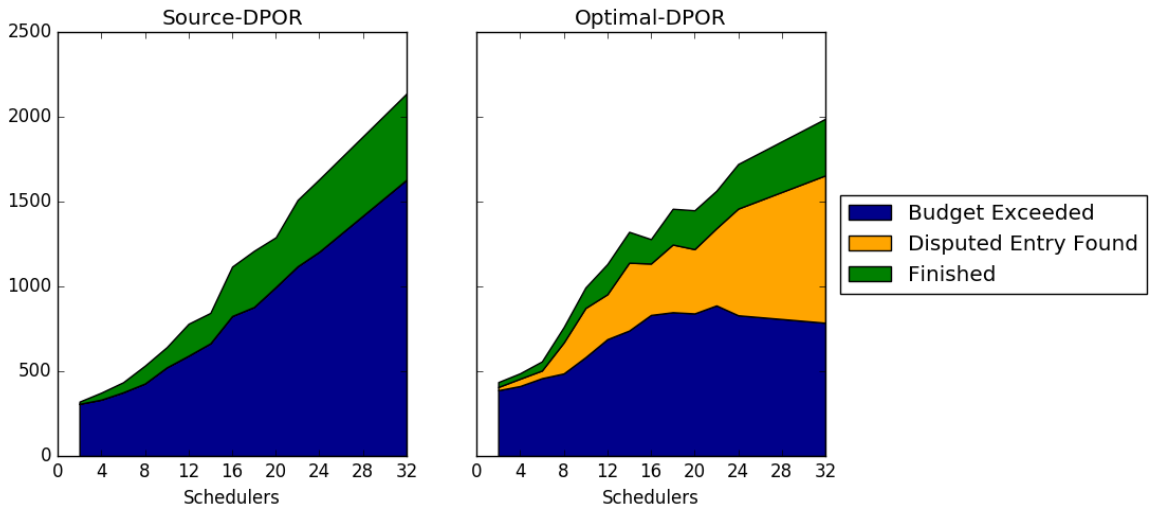
της.

Στην περίπτωση του βέλτιστου-DPOR, ενώ εξακολουθούμε να έχουμε αξιοπρεπή επιτάχυνση και κλιμάκωση, ο αλγόριθμος φαίνεται να σταματά την κλιμάκωση ταχύτερα από την πηγή-DPOR. Για παράδειγμα, το βέλτιστο-DPOR με 32 schedulers επιτυγχάνει ταχύτητα 17,3 για indexer 17, ενώ για ώρα αιχμής έχει ένα speedup 13,4, παρόλο που με 24 schedulers έχει ένα speedup 15,3. Μπορούμε να παρατηρήσουμε εδώ ότι η κλιμάκωση του αλγορίθμου αρχίζει να σπάει. Επομένως, ο παράλληλος αλγόριθμος βέλτιστου-DPOR εκτελεί χειρότερη από την παράλληλη πηγή-DPOR. Αυτή η συμπεριφορά αναμένεται από:

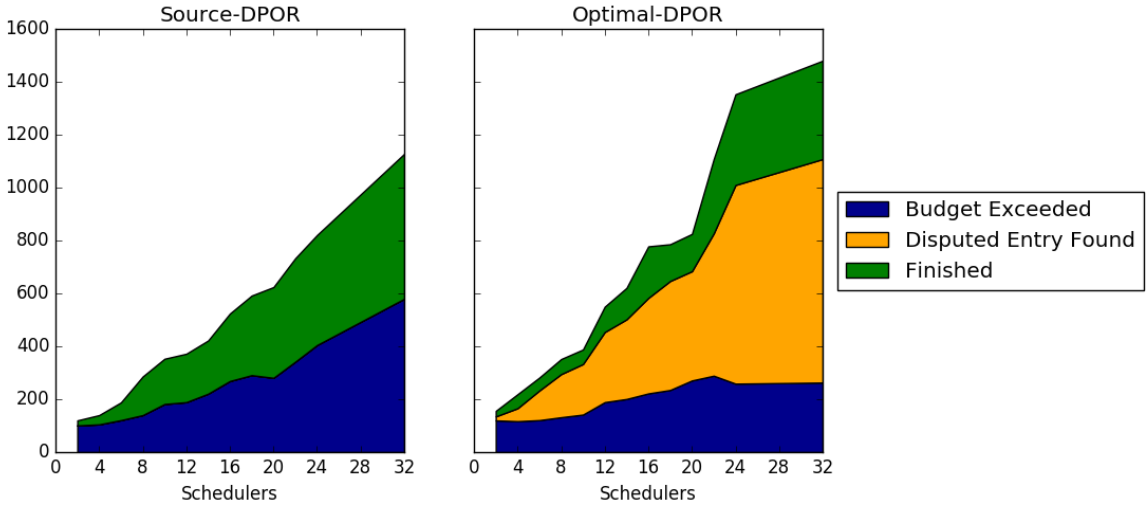
- Στην πηγή-DPOR, οι προγραμματιστές θα έχουν μια ρίζα που έχει ρίζες σε μια εκτέλεση της φόρμας $E.p$ (όπου p είναι μια διαδικασία) που τους έχει εκχωρηθεί. Ωστόσο, στους βέλτιστους προγραμματιστές DPOR αποδίδονται subtrees με ρίζες σε μια αλληλουχία εκτέλεσης της μορφής $E.w$ (όπου w είναι μια ακολουθία διεργασιών). Επομένως, οι υποτάσεις του κρατικού χώρου που έχουν εκχωρηθεί στους προγραμματιστές, στη βέλτιστη-DPOR, είναι μικρότερες, σε γενικές γραμμές, σε σύγκριση με τις υποσέλιδες του πηγαίου-DPOR. Αυτό οδηγεί στο βέλτιστο-DPOR να συναντήσει περισσότερες καταχωρήσεις backtrack που έχουν μια αμφισβητούμενη ιδιοκτησία και ως εκ τούτου, η βέλτιστη-DPOR πρέπει να επικοινωνεί με τον ελεγκτή πιο συχνά.

- Στην πηγή-DPOR, για κάθε αμφισβητούμενη είσοδο backtrack, ο ελεγκτής ελέγχει απλώς εάν η καταχώρηση υπάρχει στο δέντρο εκτέλεσης. Ωστόσο, στην περίπτωση του βέλτιστου-DPOR, ο ελεγκτής προσπαθεί να $insert$ μια ακολουθία διαδικασιών στο δέντρο εκτέλεσης. Αυτό οδηγεί στον ελεγκτή του βέλτιστου-DPOR που έχει μεγαλύτερη πολυπλοκότητα.

Κατά συνέπεια, στο βέλτιστο-DPOR, η συμφόρηση, δηλαδή ο ελεγκτής, δεν είναι μόνο πιο αργός, αλλά υπάρχει και αυξημένη επικοινωνία με αυτόν. Αυτό προκαλεί τη βέλτιστη-DPOR να είναι λιγότερο κλιμακωτή από την πηγή-DPOR.



Σχήμα 7.6: Number of times schedulers stopped their execution with a Budget of 10000.



Σχήμα 7.7: Number of times schedulers stopped their execution with a Budget of 30000.

Από το σχήμα 7.1 παρατηρούμε ότι η κλιμάκωση των αναγνωστών 15 σπάει σημαντικά ταχύτερα, σε σύγκριση με εκείνη των άλλων σημείων αναφοράς, παρά το γεγονός ότι οι αναγνώστες 15 δεν έχουν αξιοποιήσει πολύ λιγότερα ίχνη από ό, τι το lastzero 11 (Πίνακας 7.1). Ενώ προσπαθούμε να καταλάβουμε τους λόγους πίσω από αυτή τη συμπεριφορά, έχουμε δημιουργήσει ένα γράφημα που δείχνει πόσες φορές οι προγραμματιστές επικοινωνούσαν με τον ελεγκτή και γιατί έγινε μια τέτοια επικοινωνία, ανάλογα με τον αριθμό των προγραμματιστών. Στο γράφημα 7.6, μπορούμε να δούμε ότι οι προγραμματιστές μας που υπερβαίνουν τον κατανεμημένο προϋπολογισμό τους, παίζουν σημαντικό παράγοντα στην επικοινωνία προς τον ελεγκτή. Για να δοκιμάσουμε και να μετρήσουμε τον αντίκτυπο του προϋπολογισμού, προσπαθήσαμε να αυξήσουμε τον προϋπολογισμό και για τους δύο αλγόριθμους σε 30000ms. Οι χρόνοι εκτέλεσης που προκύπτουν μπορούν να φανούν στο γράφημα 7.2. Παρατηρούμε ότι η πηγή-DPOR εκτελεί χειρότερα με μειωμένο προϋπολογισμό, λόγω του γεγονότος ότι οι προγραμματιστές της γίνονται πιο ισορροπημένοι, δεδομένου ότι ο ελεγκτής κατανέμει τα σύνορα εξερεύνησης λιγότερο συχνά. Ωστόσο, το βέλτιστο-DPOR αρχίζει να λειτουργεί καλύτερα με αυξημένο προϋπολογισμό. Όπως φαίνεται από τα γραφήματα 7.6 and 7.7, the reason for this increase is that optimal-DPOR also finds disputed entries in readers 15. Finding disputed entries causes communication with the Controller, which also leads to the frontier getting repartitioned. Therefore, optimal-DPOR can have a larger budget, without affecting its load balance, since the extra communication makes up for it.

Στην περίπτωση του lastzero 11, η κλιμάκωση της βέλτιστης-DPOR σπάει σημαντικά ταχύτερα από την πηγή-DPOR. Ο κύριος λόγος πίσω από αυτό, έχει να κάνει με το γεγονός ότι η πηγή-DPOR διερευνά ένα πολύ μεγαλύτερο αριθμό ιχνών. Δηλαδή, το βέλτιστο DPOR εξετάζει 7168 ίχνη, ενώ η πηγή-DPOR 60073 (52905 μπλοκαρισμένες παρεμβολές). Ένας μεγαλύτερος αριθμός παρεμβολών σημαίνει ότι οι προγραμματιστές έχουν περισσότερη δουλειά για να διανέμουν μεταξύ τους. Είναι σημαντικό να σημειωθεί εδώ ότι η πηγή-DPOR καταφέρνει να καλύψει τη βέλτιστη-DPOR με 16 ή περισσότερους χρονοπρογραμματιστές.

7.4 Τελικά Σχόλια

Για να συνοψίσουμε, και οι δύο παράλληλοι αλγόριθμοί μας καταφέρνουν να μειώσουν σημαντικά τον χρόνο εκτέλεσης του Concuerror και είναι σε θέση να κλιμακώσουν σε μεγάλο αριθμό από schedulers. Η απόδοση, ωστόσο, διαφέρει από τα διάφορα σημεία

αναφοράς, ανάλογα κυρίως με τον αριθμό των interleavings που πρέπει να διερευνηθούν. Συγκεκριμένα, όσο μεγαλύτερος είναι ο αριθμός των interleavings, τόσο καλύτερη είναι η κλιμάκωση των αλγορίθμων μας.

Ο source-DPOR εμφανίζεται να κλιμακώνει καλύτερη από τον optimal-DPOR, λόγω του γεγονότος ότι ο optimal-DPOR έχει υψηλότερη επικοινωνία γενικά με τον Controller. Αυτό οδηγεί σε παράλληλο source-DPOR που υπερβαίνει το παράλληλο optimal-DPOR σε σημεία αναφοράς χωρίς sleep-set blocking. Αυτό δεν πρέπει να αποτελεί έκπληξη, καθώς αυτό ισχύει και για τις διαδοχικές εκδόσεις.

Επιπλέον, η παράλληλη εφαρμογή του source-DPOR, με αρκετούς schedulers, φαίνεται να είναι σε θέση να ξεπεράσει τη βέλτιστη έκδοση, ακόμη και σε δοκιμαστικές περιπτώσεις, με μεγάλο αριθμό μπλοκαρισμένων παρεμβολών σε set sleep. Ωστόσο, σε πολύ μεγάλες δοκιμαστικές περιπτώσεις, δεν αναμένω να συνεχιστεί αυτή η τάση.

Κεφάλαιο 8

Συμπεράσματα

Σε αυτή τη διπλωματική, παρουσιάσαμε την παραλληλοποίηση του Concuerror. Για να παραλληλοποιήσουμε τον Concuerror, ήταν απαραίτητο να σχεδιάσουμε παράλληλες εκδοχές για τους δύο βασικούς αλγορίθμους του: τον source-DPOR και τον optimal-DPOR. Η παραλληλοποίηση του optimal-DPOR προέκυψε εξαιρετικά δύσκολη. Επίσης, για να καταστήσουμε τον Concuerror ικανό να εξερευνεί πολλαπλά interleavings ταυτόχρονα, έπρεπε να ξεπεράσουμε ποικίλα σοβαρά ζητήματα.

Αξιολογήσαμε του αλγορίθμους μας χρησιμοποιώντας τον παραλληλο Concuerror για να ελέγξουμε διάφορα προγράμματα. Κατά τη διάρκεια αυτής της αξιολόγησης, ανακαλύψαμε ότι οι αλγοριθμοι μας επιτυγχάνουν σημαντικές επιταχύνσεις στα περισσότερα benchmarks. Επιπροσθέτως, η υλοποίηση μας είναι ικανή να κλιμακώσει για 32 παράλληλους schedulers, ανάλογα με το benchmark. Επίσης, παρατηρούμε ότι ο παράλληλος source-DPOR αλγόριθμος κλιμακώνει καλύτερα από τον παράλληλο source-DPOR, το οποίο ήταν αναμενόμενο.

Παρόλα αυτά, υπάρχει ακόμα δουλειά που πρέπει να γίνει στο μέλλον:

- Η σχεδίαση μιας παράλληλης υλοποίησης του Concuerror που δουλεύει εντός ενός Erlang κόμβου, ακόμα και αν αυτό περιορίζει τη χρήση των Erlang Pids εντός του προγράμματος που ελέγχουμε. Αυτό θα εξυπηρετεί σαν μία ελαφριά εναλλακτική, η οποία θα μπορεί να ελεγχθεί και να επαληθευτεί απο τον σειριακό Concuerror.
- Να αξιολογήσουμε την υλοποίηση μας σε ένα κατανεμημένο περιβάλλον
- Να εφαρμόσουμε τους παράλληλους αλγορίθμους μας στον optimal-DPOR με observers.
- Να εξετάσουμε και να εφαρμοσουμε bounding τεχνικές στους παράλληλους DPOR αλγορίθμους.

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, Berlin, Heidelberg, 2015, Springer-Verlag.

[Arms07] Joe Armstrong, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf, 2007.

[Aron18] Stavros Aronis, *Effective Techniques for Stateless Model Checking*, Ph.D. thesis, 2018.

[Chri13] M. Christakis, A. Gotovos and K. Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 154–163, March 2013.

[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.

[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.

[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", *Form. Methods Syst. Des.*, vol. 26, no. 2, pp. 77–101, March 2005.

[Goto11] Alkis Gotovos, Maria Christakis and Konstantinos Sagonas, "Test-driven Development of Concurrent Programs Using Concuerror", in *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, pp. 51–61, New York, NY, USA, 2011, 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.

[Lei06] Yu Lei and Richard H. Carver, "Reachability Testing of Concurrent Programs", *IEEE Trans. Softw. Eng.*, vol. 32, no. 6, pp. 382–403, June 2006.

[Matt88] Friedemann Mattern, "Virtual Time and Global States of Distributed Systems", in *PARALLEL AND DISTRIBUTED ALGORITHMS*, pp. 215–226, North-Holland, 1988.

[Mazu87] Antoni Mazurkiewicz, "Trace theory", in W. Brauer, W. Reisig and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pp. 278–324, Berlin, Heidelberg, 1987, Springer Berlin Heidelberg.

[Moor06] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.", *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sept 2006.

[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.

[Sims11] J. Simsa, R. Bryant, G. Gibson and J. Hickey, "Efficient Exploratory Testing of Concurrent Systems", *CMU-PDL Technical Report*, vol. 113, November 2011.

[Sims12] Jirí Simsa, Randy Bryant, Garth A. Gibson and Jason Hickey, "Scalable Dynamic Partial Order Reduction", in *RV*, 2012.

[Valm91] Antti Valmari, "Stubborn sets for reduced state space generation", in Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pp. 491–515, Berlin, Heidelberg, 1991, Springer Berlin Heidelberg.

[Vird96] Robert Virding, Claes Wikström and Mike Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*, Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[Yang07] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan and Robert M. Kirby, "Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software", in *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pp. 58–75, Berlin, Heidelberg, 2007, Springer-Verlag.

[Yang08] Y. Yang, X. Chen and G. Gopalakrishnan, *Inspect: A Runtime Model Checker for Multithreaded C Programs*, University of Utah Tech, 2008.