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.

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

Κατάλογος πινάκων

Κατάλογος σχημάτων

Κώδικες

List of Algorithms

Κεφάλαιο 1

Εισαγωγή

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

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

Race conditions Όπου μία δρομολόγηση έχει μη αναμενόμενο αποτέλεσμα.

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

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

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

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

1.1 Δοκιμή,΄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 [Virdd96]. 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 γεγονότα 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 μέχρι 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,$subtree(\langle B, \prec\rangle), p) = \ langle B', \ prec' \ rangle$ 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

Parallelizing source-DPOR Algorithm

Concuerror utilizes primarily DPOR algorithms to systematically test concurrent Erlang programs. Therefore, parallelizing Concuerror entails designing parallel versions for its DPOR algorithms. Since Concuerror is written in Erlang, which is a functional language that is based on message passing to share data between processes, we are going to follow a message passing approach, while developing our algorithms.

In this chapter, we are going to present the parallel version of the source-DPOR algorithm. Let us first discuss some existing work in parallelizing persistent-set based DPOR algorithms.

4.1 Existing Work

When parallelizing DPOR algorithms, ideally we would like to develop parallel algorithms that explore exactly the same number of interleavings as their sequential versions. In other words, we want to retain the soundness of our algorithms, while simultaneously not exploring more interleavings than necessary

At a first glance, parallelizing DPOR algorithms may seem straightforward. Since the state space of a program contains no cycles, we should simply distribute that state space into multiple workers-schedulers. For example, in the case of Algorithm 1, we would assign a prefix of the form $E.p$ to a scheduler. That scheduler would be responsible for the execution of $explore(E.p, Sleep)$. However this approach leads to two main issues [Yang07].

Firstly, DPOR algorithms detect races and update the exploration frontier in a non-local manner. While calls to $explore(E.p, Sleep)$ may guarantee that for all maximal execution sequences of form $E.w$, the algorithm has explored some execution sequence E' which is in $[E.w]_{\simeq}$, backtrack points may also be inserted in the prefixes of $E.p$. For instance, lets assume that we assign to one scheduler the exploration of $E.p$ and to another scheduler the exploration of $E.q$. Since we are using message passing as our programming model, our schedulers would use different copies of the prefix E. Both of those explorations may lead to adding the same process r to the backtrack at E. This would mean that both those schedulers would end up calling $explore(E.r, Sleep)$. Therefore, different schedulers may end up fully exploring identical interleavings. In order to combat those redundant explorations, Yang et al. [Yang07] suggest a heuristic that simply modifies the lazy addition of backtrack entries to the exploration frontier [Flan05] to become more eager. Adding backtrack entries more eagerly, i.e., earlier in the exploration phase, reduces the chances of two different workers exploring identical interleavings. In the above scenario, this strategy could have led to r being added to the backtrack at E before assigning the exploration of $E.p$ and $E.q$ to different schedulers. Therefore, both of our schedulers would have know that r exists in the backtrack of E and none of them would have added it, avoiding the duplicate exploration of $E.r$. However, this is simply a heuristic, which means that depending on the tested program, a significant amount of redundant computations may still occur. Particularly, this heuristic fails to prevent redundant explorations, when branches

in the code lead to different races.

Secondly, the size of different chunks of the state space cannot be known a priori. This means that some form of load balancing is essential to achieve linear speedup. Yang et al. [Yang07] suggest using a centralized load balancer to unload work from a scheduler. Specifically, a scheduler calls the load balancer when the total number of backtrack entries in the execution sequence of the scheduler exceed a threshold. However, for different programs and different number of workers, different threshold values should be used [Sims12]. Still, Yang et al. provide no insight into the problem of selecting an appropriate threshold.

Simsa et al. [Sims12] provide a more appropriate way to solve these issues. By using a centralized $Controller$, which keeps track of the current $execution\ tree$ (a tree whose branches correspond to the current execution sequences E of the schedulers), they assure that no redundant explorations occur. They also suggests the use of $time\ slicing$ to achieve load balancing.

4.2 Parallel source-DPOR

Here we are going to present how to efficiently parallelize the source-DPOR algorithm, by modifying the parallel algorithm presented by Simsa et al. [Sims12].

4.2.1 Basic Idea

Normally, DPOR algorithms perform a depth-first search of the state space to check for erroneous interleavings. Instead, we are going to use multiple depth-first searches (by partitioning the frontier of our search) to explore our state space.

We are going to use a centralized $Controller$ who will be responsible for distributing the exploration frontier to different worker-schedulers. The Controller is also going to oversee the parallel exploration, so we avoid exploring more interleavings than the sequential version. In order to do this, the controller will keep track of the frontier that is being explored in the form of an $execution\ tree$. In short, the execution tree represents the state space of our program. Nodes of the execution tree represent non-deterministic choice points and edges represent program state transitions. A path from the root of the tree to a leaf then uniquely encodes a program an execution sequence. We are also going to use the term branch to refer to execution sequences within the execution tree.

In order to avoid redundant explorations, we are going to use the concept of $ownership$ [Sims12] of a node of a state space. A scheduler exclusively owns a node of the state space if it is either contained as a backtrack entry within the part of the frontier that was assigned to that specific scheduler, or if it is a descendant of a node that the scheduler owns. All other nodes, are considered to have a $disputed$ ownership.

The scheduler, when conducting its depth-first search, is going to be allowed to explore only nodes that it owns. When encountering disputed nodes, the scheduler will report back to the Controller, which will keep track of the complete active frontier. If that disputed node does not exist within the complete frontier, then no other scheduler has explored that node and therefore, the scheduler can $claim\ ownership$ over the disputed node and continue with exploring it. Otherwise, the ownership of the disputed node has been claimed by some other scheduler and therefore, that node can be discarded from the frontier of our scheduler.

For example, if a scheduler is responsible for exploring an execution sequence E that has a single backtrack entry of p at E, then that scheduler owns every node that is a descendant of $E.p$ i.e., it owns every execution sequence that begins with the prefix $E.p$ (or else, the complete subtree that is rooted at $E.p$). Now lets assume that during the

exploration of the subtree that is rooted at $E.p$, r has been added at the backtrack at E. Since this node is disputed, the scheduler will not explore this backtrack entry. Instead, it will report back to the Controller, in order to determine whether some other entry has already claimed ownership over $E.r$.

4.2.2 Algorithm

Algorithm 3: Controller Loop

```
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);
```

The logic of the Controller is shown in Algorithm 3. The Controller maintains a $Frontier$, which is a set of execution sequences E, and an execution tree T, which contains as branches the execution sequences of the $Frontier$. For as long as there exists an execution sequence at the $Frontier$ ($Frontier \neq \emptyset$), the Controller will partition its $Frontier$ to at most N execution sequences. Then, the Controller will try to assign all of its unassigned execution sequences to any idle scheduler, by spawning $explore\_loop(E_c, Budget)$ functions. Finally, it will block until it receives a response from a scheduler.

Algorithm 4: Frontier Partitioning

```
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;
```

During the *partitioning* phase (Algorithm 4), we inspect the current $Frontier$ to determine whether we should create additional execution sequences. Every execution sequence that contains more than one backtrack entry is split into multiple sequences until either the $Frontier$ contains N sequences or all sequences have exactly one backtrack entry. It is vital to modify sleep sets appropriately because, if we were to simply remove

backtrack entries, our algorithm would have an increased amount of sleep-set blocked interleavings. In addition, we would also end up potentially re-adding the same backtrack entries, which would lead to exploring duplicate interleavings.

Algorithm 5: Scheduler Exploration Loop

```
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 ;
```

Algorithm 5 details how the schedulers explore their assigned state space. A call to $explore\_loop(E_0, Budget)$ guarantees that for all maximal execution sequences of form $E_0.w$, the algorithm has explored some execution sequence E'_0 which is in $[E_0.w]_{\simeq}$ We use $explore(E)$ and $plan\_more\_interleavings(E')$ as a high level way to describe the main phases (state exploration and race detection) of the sequential source-DPOR. The $plan\_more\_interleavings(E')$ function could add backtrack points in prefixes of E_0. This could lead to different schedulers exploring identical interleavings. We avoid this by having the function $get\_next\_execution\_sequence(E')$ return the largest prefix of E' that has a non-empty backtrack set. This leads to a depth-first exploration of the assigned state space before considering interleavings outside of the state space. The exploration continues until we encounter a prefix of E_0 ($size(E'') \leq size(E_0)$). This is necessary to assert that the specific scheduler will not explore interleavings outside of its state space. When the exploration terminates, the backtrack points added to the prefixes of E_0 will be reported back to the Controller.

Algorithm 6: Handling Scheduler Response

```
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;
```

When the Controller receives a response (an execution sequence E) from a scheduler (Algorithm 6), it will try and report any new backtrack entries in E to the execution tree T. This is done through the $update\_execution\_tree(E, T)$ function. This function iterates over the execution sequence and the execution tree simultaneously and those backtrack entries of E that are not found in the execution tree, are added to it and they are not removed from the execution sequence. This means that this execution sequence is the first to *claim ownership* over those entries and the state space that exists under them. Any backtrack entries that already exist in T are removed from the execution sequence E (and added to the sleep sets at the appropriate prefixes of E), because some other execution

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

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

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

Σχήμα 4.1: Simple readers-writers example

sequence has already claimed their ownership. This updated execution sequence is then added to the $Frontier$ of the Controller. Through this we make sure that two schedulers cannot explore identical interleavings.

When updating the execution tree, we also use the initial execution sequence that was assigned to a scheduler (the one denoted as E_0 at Algorithm 5) to figure out which parts of the execution tree have already been explored. Those parts are deleted from the execution tree. This is mandatory in order to keep the size of the execution tree proportionate to the size of our current $Frontier$.

4.2.3 Load Balancing

In order to achieve decent speedups and scalability it is necessary to have load-balancing [Sims12]. This done through time-slicing the exploration of execution sequences. This is the reason behind the use of $Budget$ in Algorithm 5. By having schedulers return after a certain time-slice, we can make sure that even if their assigned state space was larger compared to that of other schedulers, they will eventually exit and have their execution sequence and subsequently, their state space, partitioned. How effective is this method is determined by two variables, the upper limit N to the number of execution sequences in our $Frontier$ and the $Budget$ of a scheduler.

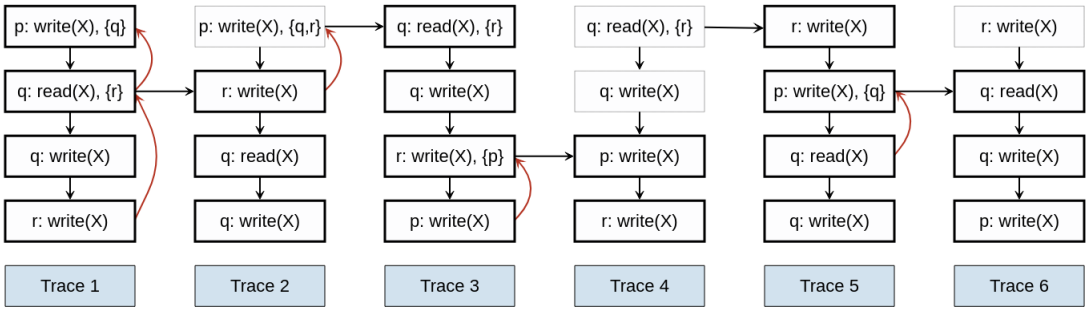
Higher upper limit N means that a larger work pool is available to the workers and they do not always have to wait for the $update\_execution\_tree(E, T)$ function to terminate. Therefore, the utilization of the schedulers increases. However, a larger N means increased memory requirements, since more execution sequences are active at each time. More importantly, it also means that the state space splits into smaller fragments. This increases how frequently a scheduler discovers disputed nodes, which leads to an increase in the communication between the schedulers and the controller. Setting this limit to double the amount of schedulers, produces decent results for most test cases[Sims12].

Smaller $Budget$ values lead to more balanced workload, since the work is distributed more frequently. However, extremely low values may lead to an increased communication overhead between the Controller and the schedulers. This can also cause the Controller to become a significant bottleneck. The best way to deal with this, is to pick an initial value $Budget$ of around 10 seconds. When a scheduler starts a new exploration, the value of its budget will be dynamically assigned by the Controller depending on the amount of idle schedulers. For instance, the first execution should have a budget of $\frac{Budget}{n}$ were n is the total amount of schedulers, which are all idle. When half the schedulers are idle, this value should be $\frac{Budget}{2}$, etc. This makes it possible to have reduced communication (higher budget) during periods with many busy schedulers and a better balancing (lower budget) during periods with many idle schedulers.

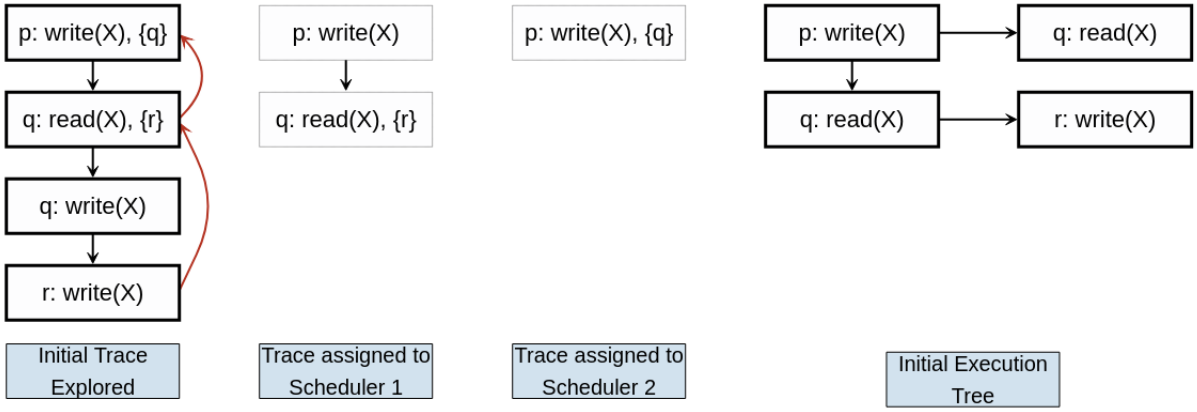
4.2.4 A Simple Example

Let us consider the example in Figure 4.1. In this case we have 3 processes that write and read a shared variable x. Figure 4.2 represents the traces explored during the sequential source-DPOR. We use a bold rectangle to represent a new event and a faint

rectangle to denote a replayed event. The red edges represent the races that are detected and planned. The source set at a state is represented inside the brackets.

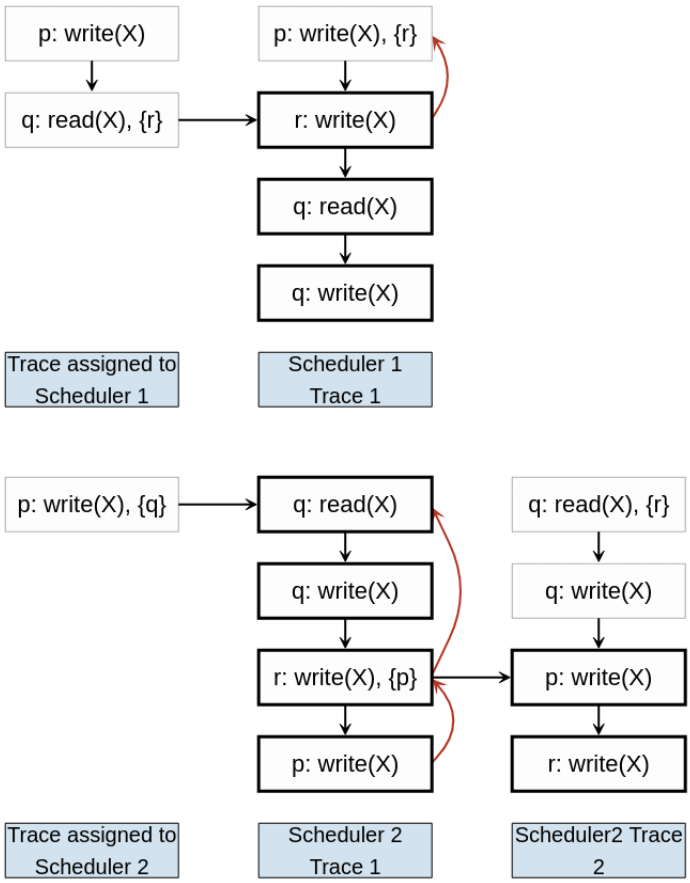


Σχήμα 4.2: Interleavings explored by the sequential source-DPOR.



Σχήμα 4.3: Initial interleaving explored by the parallel algorithm.

Figure 4.3 depicts the initial step of the parallel source-DPOR. This initial execution sequence, along with detected races is partitioned into fragments which get assigned to different schedulers. This image also contains the initial execution tree which represents the state space that exists in our exploration frontier at this point.



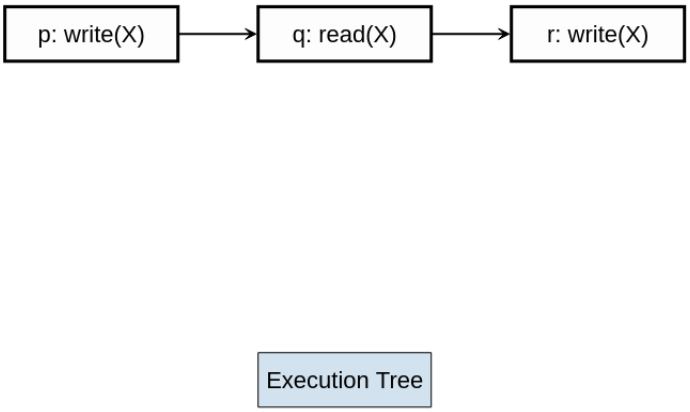
Σχήμα 4.4: Exploration of the assigned traces by each scheduler.

In Figure 4.4 we illustrate how each scheduler explores its assigned execution sequence. The first scheduler explores its first trace: $p.r.q.q$ (equivalent to the second trace at Figure 4.2). After the race detection takes place, there are no backtrack entries added below its assigned trace. The only execution sequence planned is the sequence r. However, this sequence is not explored since it does not belong to the state space of the scheduler. The Controller assigned to the second scheduler the execution sequence q. After exploring its first trace, two more races are detected. It is important to notice here that the first trace of the second scheduler is equivalent to the third trace of the sequential algorithm. However, the sequential algorithm detects only one race. This happens because on the sequential algorithm the race between $q : read(x)$ and $r : write(x)$ had already been detected at the second interleaving and so its planning gets skipped. On the contrary, on the parallel algorithm this interleaving is explored by another scheduler and therefore, there is no knowledge of this race been already detected. This leads to both our schedulers having detected the same race. Nevertheless, in both schedulers this new backtrack entry is outside of their state space.

This means that they will have to report their results back to the Controller. The schedulers that reports first its results, will be the one to update the execution tree by inserting the new entry found. This scheduler will add its execution sequence to the frontier, which will be again partitioned (no need for a partition here since the unexplored frontier will only have one race). Then this execution sequence will be assigned to an idle scheduler. The scheduler that reports second to the Controller, will not be able to insert its backtrack entry into the execution tree, because that entry will already be there, and the backtrack entry gets removed from its execution sequence. This execution sequence will

be left with no more backtrack entries and as such it will not be inserted into the frontier. This guarantees that we do not explore identical interleavings more than once.

Lets assume that Scheduler 1 was the one that managed to report first to the Controller. Then the execution tree will be the one depicted in Figure 4.5. Notice here that the states explored by the Scheduler 1 were deleted from the execution tree. This keeps the size of the execution tree proportional to the size of the current exploration frontier. At this point Scheduler 1 will add its execution sequence to the frontier. Then the Controller will assign this sequence to Scheduler 1 (since Scheduler 2 has not yet returned), and Scheduler 1 will explore the last 2 traces (trace 5 and 6 from the sequential example). After both Scheduler 1 and 2 have returned the execution will have finished since there will no more traces left to explore.



Σχήμα 4.5: Execution Tree if Scheduler 1 returned first.

Κεφάλαιο 5

Parallelizing optimal-DPOR Algorithm

In this chapter, we are going to present a first attempt in parallelizing the optimal-DPOR algorithm, provide an analysis on why this attempt fails to achieve any speedup and then present an improved parallel optimal-DPOR algorithm.

5.1 Parallel optimal-DPOR - A First Attempt

5.1.1 Basic Idea

Parallelizing the optimal-DPOR algorithm is significantly more complicated than source-DPOR. We do know that whenever a call to $Explore(E, Sleep, WuT)$ returns during Algorithm 2, then for all maximal execution sequences of form $E.w$, the algorithm has explored some execution sequence E' which is in $[E.w]_{\simeq}$ [Abdu14]. In other words, calls to $Explore(E, Sleep, WuT)$ guarantee that the complete subtree rooted at E will be explored. However, the complete WuT at some execution sequence E cannot be known until we have completed exploring all execution sequences which are ordered before E, according to the total order of our state space (Definition 3.4). This happens because the $insert_{[E']}(v, wut(E'))$ function can add entries to any wakeup tree of an execution sequence that is ordered after the current execution sequence.

Therefore, when assigning an incomplete wakeup tree to a scheduler, there is no guarantee that the scheduler will explore the complete assigned state space. This means that if a scheduler inserts a fragment into a wakeup tree owned by a different scheduler, we cannot know if that fragment (or a different but equivalent fragment) was indeed explored. As a result, the concept of the ownership of a backtrack entry, as defined in Chapter 4, cannot remain the same for the optimal-DPOR algorithm.

Another main issue with optimal-DPOR, is the fact that the $insert_{[E]}(v, wut(E))$ function may end up and inserting at $wut(E)$ and execution sequence that is different than v (but it will lead exploring an equivalent subtree). This means that two execution sequences, while different, may be equivalent. This can potentially lead to different schedulers inserting in their wakeup trees execution sequences that while different, may produce equivalent interleavings and therefore, the optimality of the algorithm could be lost.

However, we do know that any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$ [Abdu14]. This means that, during the sequential algorithm, any fragment that is inserted into a wakeup tree is a fragment that must be explored, unless it is removed during the exploration phase of the algorithm. Therefore, when we insert a fragment into a wakeup tree, we can explore it out of order, as long as we are careful to remove execution sequences that would have been removed on the sequential algorithm. We can take advantage of this to create an algorithm that can explore many interleavings in parallel but race detects each explored interleaving sequentially.

5.1.2 Algorithm

Due to the fact that we need to have parallel exploration of interleavings but sequential planning, we need to decouple the normal exploration loop of a scheduler into two different parts: $state\ exploration$ and $race\ detection - planning$. Our workers (the schedulers) will be responsible for the first part. For the second part, we are going to use a centralized Planner. However, in order to be able to better distribute the available work to the schedulers when the Planner is busy, we are also going to use a Controller.

Algorithm 7: Controller for optimal-DPOR - First Attempt

```
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);
```

Algorithm 7 describes the functionality of the Controller. Similarly to the source-DPOR parallel version, the Controller is responsible for maintaining the current Frontier (as well as partitioning it) and the current Execution Tree and for assigning execution sequences to schedulers, for as long we have idle schedulers and available work.

Apart from that, the Controller also maintains a queue of fully explored execution sequences that need to be race detected. When the Planner is idle and the queue is not empty, the execution sequence is updated (through $update\_trace(E, T)$) and then is sent to the Planner so its races can be detected. When updating the execution sequence from the execution tree, the subtrees of the execution tree which are ordered after the execution sequence (according to the ordering of our state space Definition 3.4) are inserted into the execution tree as not_owned wakeup trees. This guarantees that no redundant fragments are going to be inserted for future explorations and therefore, the algorithm remains optimal.

The $plan(E)$ function race detects the fully explored execution sequence E according to the logic of optimal-DPOR (Algorithm 2). When the planning of the sequence is finished the results are reported back to the Controller. The $explore(E)$ function explores the execution sequence E until a maximal execution sequence has been reached and reports back that execution sequence to the Controller.

Partitioning the exploration frontier (Algorithm 8) has two main differences, compared to the parallel source-DPOR. Firstly, the frontier gets partitioned completely, so we can maximize the parallelization of the exploration phase. Secondly, the entries that are distributed from one execution sequence, are not simply removed from the backtrack and added to the sleep set. It is vital here to maintain the correct ordering between the interleavings

Algorithm 8: Optimal Frontier Partitioning

```
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;
```

(Definition 3.4), because during the exploration phase of the optimal-DPOR algorithm, sequences can be removed from the wakeup tree. Maintaining the ordering will keep this behavior intact in the parallel version. Therefore, the given entry is marked as not_owned at the distributed sequence. The function $split\_wut\_at(v, wut(E'_c))$ splits the copy of the wakeup tree to three parts: the $Prefix$ (the wakeup tree entries ordered before the sequence v), the leaf v and the $Suffix$ (the wakeup entries ordered after v). The first processes processes of the entries of the $Prefix$ are added to the sleep set at the new execution sequence E'_c (e.g. if $p.q.r$ is a leaf in $Prefix$, then p is added to $sleep(E'_c)$). The $Suffix$ entries are marked as not_owned at E'_c.

Algorithm 9: Handling Scheduler and Planner Response

```
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;
```

After assigning the available work to the available schedulers and the Planner, the Controller will wait for a response either from a scheduler or the Planner (Algorithm 9). When a response is received from a scheduler, the fully explored received execution sequence will be added to the queue of the Planner and the Controller will continue with its loop (Algorithm 7). If a response is received from the Planner, the Controller will update the execution tree T by adding the new wakeup trees that were inserted by the Planner and by deleting the suffix of the execution sequence that was just explored and has no wakeup trees. We delete this part in order to have the execution tree only contain the part of the state space that is either currently getting explored or is planned to be explored. If we were to not delete those suffixes, the size of the execution tree would eventually be the size of our complete state space.

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

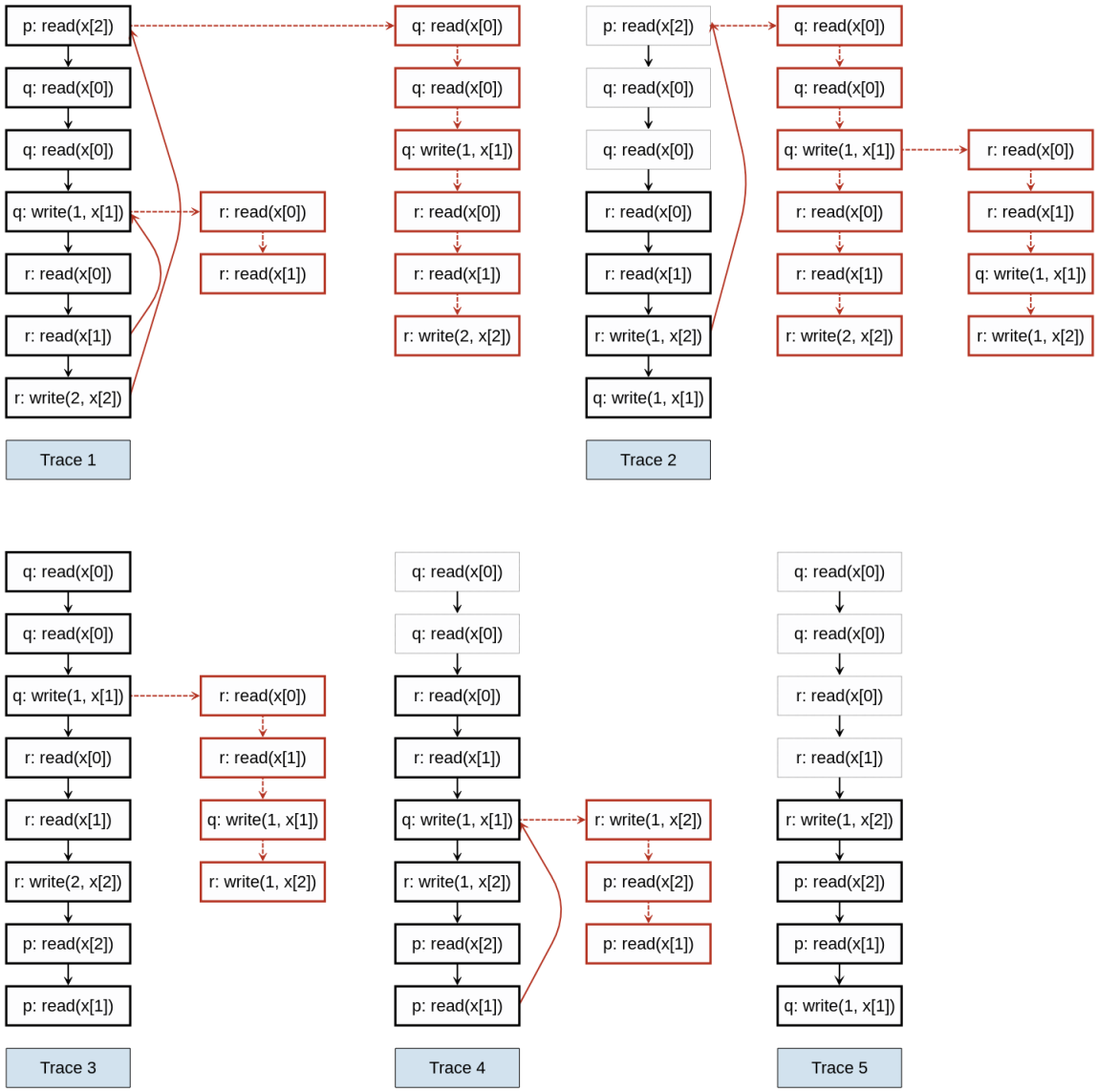
```
q:
R1 = x[0]
R2 = x[0]
assert(R1 == R2)
x[1] = R2 + 1
```

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

Σχήμα 5.1: *Lastzero* 2 example

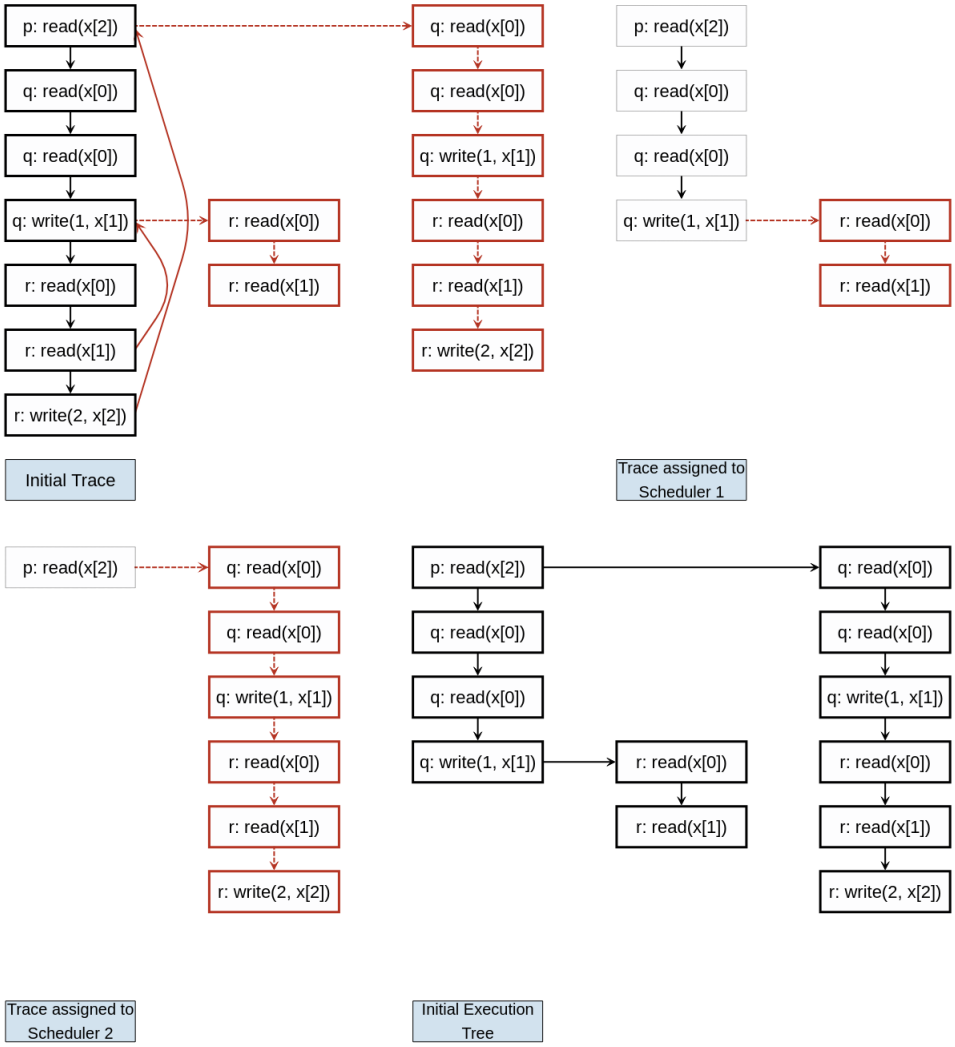
5.1.3 Example

In Figure 5.1 we can see the pseudocode of *lastzero* 2, were we have an array of 3 elements (initially all elements have a zero value) and three processes. The first process (p) searches the array for the zero element with the highest index. The other two processes increase their assigned element by a value of 1.



Σχήμα 5.2: Interleavings explored by the sequential optimal-DPOR

Figure 5.2 represents the traces explored during the sequential optimal-DPOR. We use a black bold rectangle to represent a new event and a faint rectangle to denote a replayed event. The continuous red edges represent the races that are detected and planned. The red nodes represent the wakeup tree entries at each trace.



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

Figure 5.3 depicts the initial step of the parallel optimal-DPOR. An arbitrary execution sequence is explored initially and then its races are detected and planned in the form of wakeup trees. The wakeup trees are distributed into different fragments and all unassigned fragments are assigned into the idle schedulers. Also the execution tree is initialized with the current exploration frontier.

In this example, lets assume that Scheduler 2 finishes first the exploration of its assigned trace. The controller will then receive the new explored trace and will add this trace to the queue of the Planner. Since the Planner is idle, this trace will be sent to the Planner to be race detected. While race detecting this trace, no more interleavings will be planned. This trace is equivalent to the 3rd trace of the sequential execution (Figue 5.2). Notice here that in the sequential algorithm this trace had an additional wakeup tree. This wakeup tree was planned by the 2nd trace of the sequential algorithm, which has yet to be race detected in our example. Therefore, traces 4 and 5 of the sequential algorithm cannot be planned from the 3rd trace but only from the 2nd. This makes apparent the main issue

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.

of the parallelization of the optimal-DPOR: the complete wakeup tree at some execution sequence E cannot be known until we have completed exploring all execution sequences which are ordered before E, according to the total order of our state space (Definition 3.4)

5.1.4 Performance Evaluation

We are going to use the following two synthetic benchmarks to evaluate our first attempt in parallelizing the optimal-DPOR algorithm:

- $readers\ N$: 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.

Our parallel optimal-DPOR fails to achieve any type of speedup. Table 5.1 holds information about how our implementation runs on various test cases. The exploration and planning time rows show what percentage of the time of the sequential algorithm is spend on exploring and planning interleavings respectively. Those measurements will help us explain the reason for this lack of speedup.

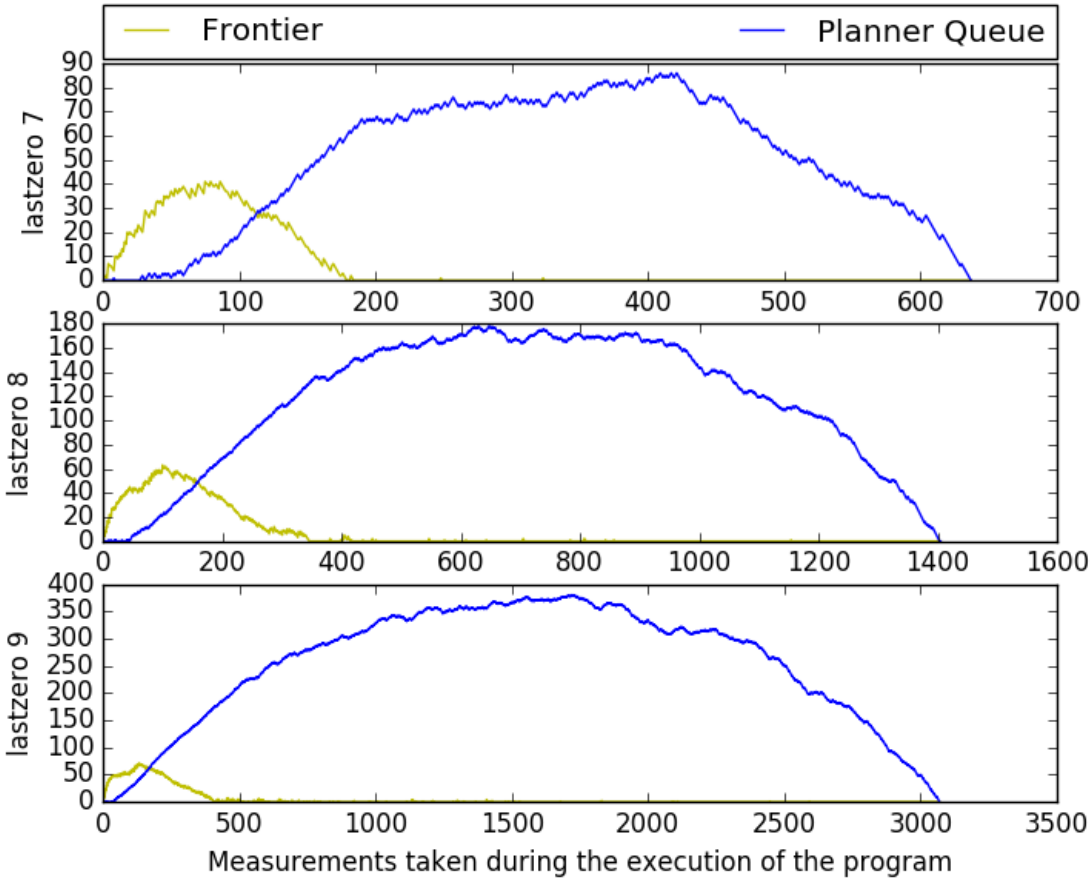
5.1.5 Performance Analysis

This lack of performance is justified by the following reasons.

Firstly, lets look at the exploration and planning percentages of the sequential algorithm, presented at Table 5.1. While the percentage of exploration time in the case of a small test case like readers10 is around 40%, we have observed that for larger test cases this figure varies between 10%-30%. This means that even on an ideal setup, with zero overhead and infinite schedulers, our speedup could never exceed a factor of 1.429 (on a test case with 70% planning time, if we consider that the exploration phase happens instantly the speedup would be $1.429 = \frac{100}{70}$, since the planning phase would still have to take place sequentially). This reason, by itself, makes it impossible for our algorithm to achieve good performance and scalability.

What is more, the rate with which new interleavings are planned for exploration leads to schedulers not being sufficiently utilized throughout the execution of the algorithm. Specifically, we have noticed that the execution of the DPOR algorithm can be split into three consecutive phases. In the first phase, race detecting a single interleaving generates a large amount of interleavings that need to be explored and therefore the exploration Frontier is relatively large. This means that the schedulers have enough work and are being kept utilized. During the second phase, race detecting an interleaving generates a relatively small amount of new interleavings that need to be explored. Generally, throughout this

phase, not enough work is being generated and the schedulers tend to stay idle. In the third phase, barely any new interleavings are planned, until the complete state space has been explored, at which point the algorithm terminates.



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

This pattern becomes visible from Figure 5.4. We have run our algorithm with four schedulers for the lastzero 7, 8 and 9 benchmarks. To create this graph we have taken measurements regarding the size of queue of the Planner and the exploration Frontier of the schedulers. The measurements are taken after either the exploration or the planning of an interleaving has finished. In this graph, the three aforementioned phases can be clearly observed. After the first phase ends, we notice that the relative large exploration Frontier that has been created, gets "consumed" in an extremely fast rate, due to the fact that we have four scheduler consuming from it, while only one Planner producing in it. Also, like we mentioned before, the exploration of a single trace is significantly faster than its race detection. Consequently, during the second phase there are very few traces within the exploration Frontier and therefore, the schedulers stay underutilized. We can also notice from the graph that this effect intensifies while our state space increases. The main reason behind this is the fact that as the state space of a program increases, the planning of an interleaving becomes even more slow, compared to the exploration of an interleaving.

To make matters worse, our algorithm also has a significant overhead. As mentioned before, communication between the controller and the workers can be a substantial bottleneck. The parallel source-DPOR deals with it by assigning a state space to the workers, minimizing the need for communication. However, in this attempt the use of the centralized planner leads to the schedulers having to report a trace back to the controller every single time an

exploration reaches its end.

5.2 Scalable Parallel optimal-DPOR

5.2.1 Basic Idea

Like we mentioned in the previous section, trying to develop a parallel optimal-DPOR algorithm based on the parallel source-DPOR has two main issues.

Firstly, in the source-DPOR algorithm, the backtrack is a set of processes (Definition 3.3), which means it contains single steps that the algorithm is going to "take" in the future. In the parallel algorithm, as we mentioned in Chapter 4, when a backtrack entry p at an execution sequence E is assigned to a scheduler, that scheduler will own every trace that starts with $E.p$ i.e., the scheduler can explore the subtree that is rooted at E, without reporting back to the controller (unless its Budget is not exceeded). However, in optimal-DPOR, backtrack entries are trees that contain, as nodes, sequences of steps that will be explored. A scheduler could not own the tree due to the fact that in Algorithm 2, new fragments keep getting inserted into the wakeup trees and therefore, the trees are not complete until they are about to be explored. A wakeup tree is complete, when all the smaller nodes w.r.t the ordering of the tree ($p = min_{\prec}\{p \in wut(E)\}$) have been explored. At each point throughout the execution of the algorithm, there exists only one wakeup tree with that property. Nonetheless, any leaf of $\langle B, \prec \rangle$ remains a leaf of $insert_{[E]}(w, \langle B, \prec \rangle)$ [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:

- A leaf sequence can be marked as $owned$ in the wakeup tree of a scheduler. This means that this scheduler also own every node in the subtree rooted at that leaf. For example, if v is an owned leaf sequence in $wut(E)$ of some scheduler, then that scheduler owns every execution sequence that has a prefix of $E.v$.
- A leaf sequence is marked as not_owned when some other scheduler is responsible for the corresponding subtree.
- All other nodes are considered $disputed$. When a leaf sequence is inserted underneath a disputed node, that leaf sequence is considered disputed.

The second issue has to do with the fact that the $insert_{[E]}(v, wut(E))$ function may end up inserting a different the sequence (but an equivalent one)than v. This means that two leaf entries, while different, may be equivalent. Therefore, the ownership of disputed nodes cannot be resolved by simply checking whether those nodes exists in the execution tree. However, we can modify $insert_{[E]}(v, wut(E))$ to insert execution sequence within subtrees of the execution tree, instead of wakeup trees. If a disputed sequence can be inserted into the execution tree, then this means that no other scheduler has discovered an equivalent execution sequence and therefore, ownership is claimed over that execution sequence. Otherwise, this sequence is marked as not_owned.

5.2.2 Algorithm

The Controller has the same logic as with the one from parallel source-DPOR (Algorithm 3). Similarly with the source-DPOR algorithm, The Controller maintains a $Frontier$, which is a set of execution sequences E, and an execution tree T, which contains as branches the execution sequences of the $Frontier$. For as long as there exists an execution sequence at the $Frontier$ ($Frontier \neq \emptyset$), the Controller will partition its $Frontier$ to at most N execution sequences. Then, the Controller will try to assign all of its unassigned execution

sequences to any idle scheduler, by spawning $explore\_loop(E_c, Budget)$ functions. Finally, it will block until it receives a response from a scheduler.

Algorithm 10: Optimal Frontier Partitioning - Scalable Algorithm

```
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;
```

The *partitioning* phase (Algorithm 10) works pretty similarly with Algorithm 4. The main difference here, is that when we distribute sequences from the wakeup tree, we do not simply add them on the sleep set of the original execution sequence, but we mark them as not owned. The reason behind this, can be seen at lines 20-21 of Algorithm 2. After optimal-DPOR has finished exploring a wakeup tree at some execution sequence E, the beginning process of that wakeup tree is added at the sleep set at E and sequences that begin with that process are removed from the remaining wakeup trees at E. In order to keep this behavior intact, we have modified the optimal-DPOR exploration phase, so that when encountering nodes, whose every child is *not_owned*, those nodes are added to the sleep set and the appropriate sequences are removed from the wakeup tree.

Algorithm 11: Scheduler Exploration Loop - Scalable optimal-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 ;
```

The scheduler exploration loop (Algorithm 11)mostly remains the same. We have simply changed the termination check to reflect the change in the concept of ownership. Lets not here, that when $get\_next\_execution\_sequence(E')$ finds nodes that are of *not_owned* ownership, the initial process p of those nodes gets added to the appropriate sleep set and the sequences of form begin $p.w$ get removed from the appropriate wakeup tree. In other words, this is responsible for lines 20-21 of Algorithm 2.

The pseudocode of the handling of the scheduler response remains the same as in Algorithm 6. However the $update\_execution\_tree(E, T)$ function changes. This functions iterates over the execution sequence and the execution tree simultaneously, and for any

disputed wakeup tree leaves that are found in the execution sequence a modified version of $insert_{[E]}(v, wut(E))$ is called. This version tries to insert the leaves from the wakeup trees of the execution sequence to the execution tree. If it succeeds, it claims ownership over this leaf and over every equivalent leaf. If no insertion is made, this means that some other scheduler has claimed ownership over an equivalent leaf and therefore, this leaf is marked as not_owned in the execution sequence E.

When updating the execution tree, we also delete parts from the execution tree that have already been explored. Lastly, the load balancing mechanism remains the same with the parallel source-DPOR algorithm.

Κεφάλαιο 6

Implementation Details

In Chapter 4 we described how the Source-DPOR and Optimal-DPOR algorithms can be parallelized, by using multiple schedulers to explore different interleavings concurrently. In order to accomplish this, some modification are to be made. Here we summarize those necessary modifications.

We should briefly describe how the scheduler works in the sequential implementation. The scheduler starts by exploring completely an arbitrary interleaving (with a maximal sequence E), through the function *explore/1*. It continues by calling *plan_more_interleaving/1*, in order to detect the races of the explored interleaving and plan the exploration of future interleavings according to the logic of the used algorithm. Let's assume that we must explore a sequence $E'.p$, where E' is a prefix of E and p a process. The next invocation of *explore/1* will reset all actor processes, to force them back to their initial state, and then it will replay the prefix E' to recreate the global state at E', without completely recreating the events in the prefix. After the replay is done, the processes in the backtrack (or in the wakeup tree) will be scheduled and finally the remaining events will be scheduled arbitrarily so as no more processes are enabled. The scheduler will then try and plan more interleavings. We are finished when there are no more interleavings left to explore (or until an error is found if the option *keep_going* is set to *false*).

Let us, also, describe the main datatypes used in the scheduler:

- $event()$: corresponds to the event e of a process p, according to our notation. It contains the Erlang Pid (process identifier) of the actor process p, as well as information about the code (e.g. the BIF) of this specific event.
- $\#event\_tree\{\}$: refers to either the backtrack or the wakeup tree at a specific point.
- $\#trace\_state\{\}$: holds information about an execution step of an execution sequence E, such as the backtrack (or the wakeup tree) and the sleep set at this point.
- $\#scheduler\_state\{\}$ a record that contains information regarding the state of the scheduler 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 Dealing with Processes

In the sequential Concuerror, for each process of a tested program, the scheduler needs to spawn only one process. The scheduler will then control the execution of the processes of a program to produce different interleavings. In order to concurrently explore different interleavings, for every process in the tested program, each parallel scheduler must spawn its own process. This technically means that we should have different Erlang processes that correspond to the same process of the tested program. Erlang processes are characterized by their Pid, which is globally unique. The Pid of a process is also used in Concuerror to

identify a process and, therefore, characterize a trace. When transferring traces between schedulers any Pid found anywhere in the trace should change to reflect the Pids of the different schedulers.

This means that a mapping should be created between the Pids of the different schedulers. This mapping can be established through the *symbolic names* that Concuerror assigns to the tested process with the following logic:

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

However, creating such mappings is not enough to guarantee that a trace can be transferred between different schedulers. It is important that the same execution sequence leads to the same global state regardless of the scheduler that explores it. Specifically, Erlang gives the ability to compare Pids. For instance, the ordering of two Pids could change the outcome of a branch in a program. This could result in the same trace leading to different global states on different schedulers. What is more, through the use of the BIF *pid_to_list/1*, a Pid could exist in the form of a string in some trace and as a result we would have to try and parse every string in a trace to check whether is refers to a Pid.

We solve these issues by having each scheduler run on its own Erlang node. It is possible for two processes, located on different nodes, to have the same local Pid. While trying to implement such a mechanism, we have encountered two main issues:

- 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 Execution Sequence Replay

Even after fixing the Pid issue, replaying traces on different schedulers is not going to work. There are two basic reasons behind this.

Firstly, during the execution of certain events (such as events related to ETS tables or spawning) Concuerror uses various ETS tables to keep track of specific information. When Concuerror explores a trace in replay mode, it makes sure that such information exists and drives the tested processes to the appropriate state. Therefore, when Concuerror explores a trace it creates some side-effects on the its global state. Those side-effects will not exist on a different Erlang node, since ETS tables are not shared between nodes.

```
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

Secondly, user defined lambda functions can have some environment variables. The value of those variables is immutable once the lambda function is defined. In Listing 6.1, if a trace contains an event that applies this function, this event will not be able to be properly replayed by a scheduler other than the one that created it, since the Pid environment variable cannot be changed. The only reasonable way to solve this is to change how replaying works.

Specifically, we need to have two different replay modes: *pseudo* and *actual*. Pseudo replay is used when replaying traces that were created by the same scheduler and works exactly like the replay of the sequential Concuerror. Actual replay recreates the events and the side-effects of a trace and is used for replaying interleavings received from other schedulers. On built-in events, we achieve this by setting an *actual_replay* flag to *true* and by making changes on how the *concuerror_callback* module handles those replays. On other events, we set their *event_info* value to *undefined* forcing those events to be recreated.

Κεφάλαιο 7

Performance Evaluation

In this chapter, we are going to present the performance results of our parallel source-DPOR and optimal-DPOR algorithms, implemented in Concuerror. We are going to evaluate our results on some "standard" and synthetic benchmarks that are normally used to test DPOR algorithms, as well a real erlang program. Finally, we will try to explain the behavior of the parallel program as reflected in those charts, drawing related conclusions whenever possible.

7.1 Tests Overview

First we are going to give a brief overview of the programs tested:

- $indexer\ N$: 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 Performance Results

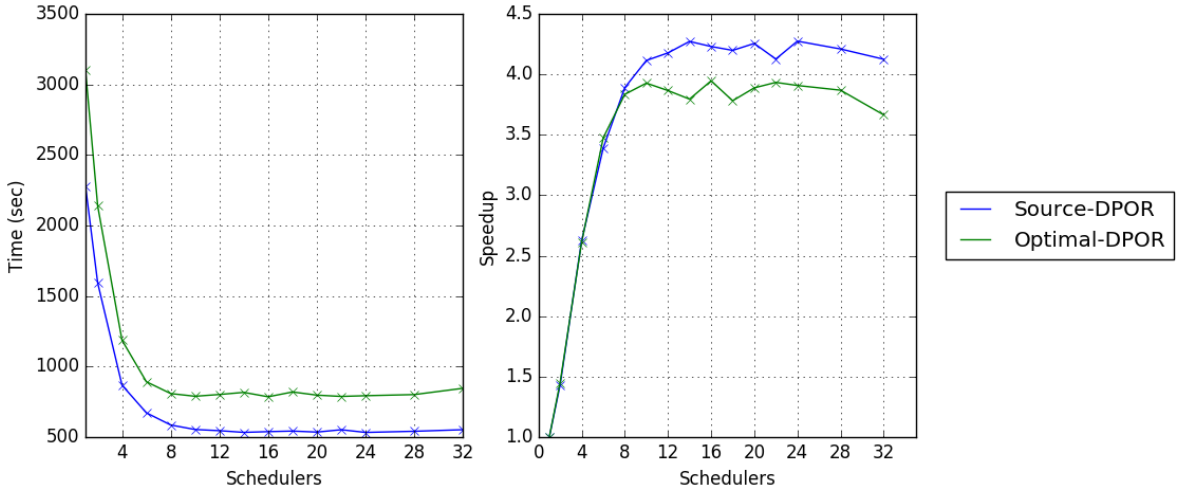
The benchmarks were performed on a multiprocessor with 64 AMD Opteron 6276(2.3 GHz) cores, 126 GB of memory, running Linux 4.9.0-8amd64 and running the later Erlang version (Erlang/OTP 21). While running our tests, we are using the $–keep\_going$ flag to continue exploring our state space, even after an error is found. We do this so we can evaluate how fast the complete state space gets explored.

Table 7.1 contains information about the traces explored and the duration of those explorations for the sequential versions of source-DPOR and optimal-DPOR, as well the duration of the parallel versions with one scheduler, a fragmentation value of one, and a Budget of 10000.

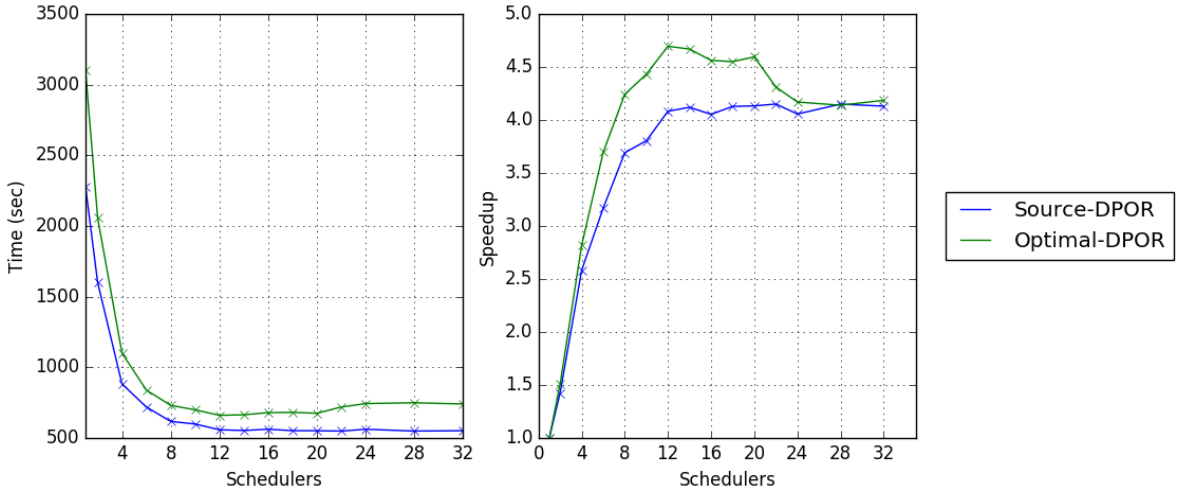
Benchmark	Traces for source-DPOR	Traces for optimal-DPOR	Time for source-DPOR	Time for parallel source-DPOR with 1 scheduler	Time for optimal-DPOR	Time for parallel optimal-DPOR with 1 scheduler
lastzero 11	60073	7168	49m8.510s	53m59.169s	14m8.266s	17m50.494s
indexer 17	262144	262144	186m8.136sec	205m24.872sec	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.

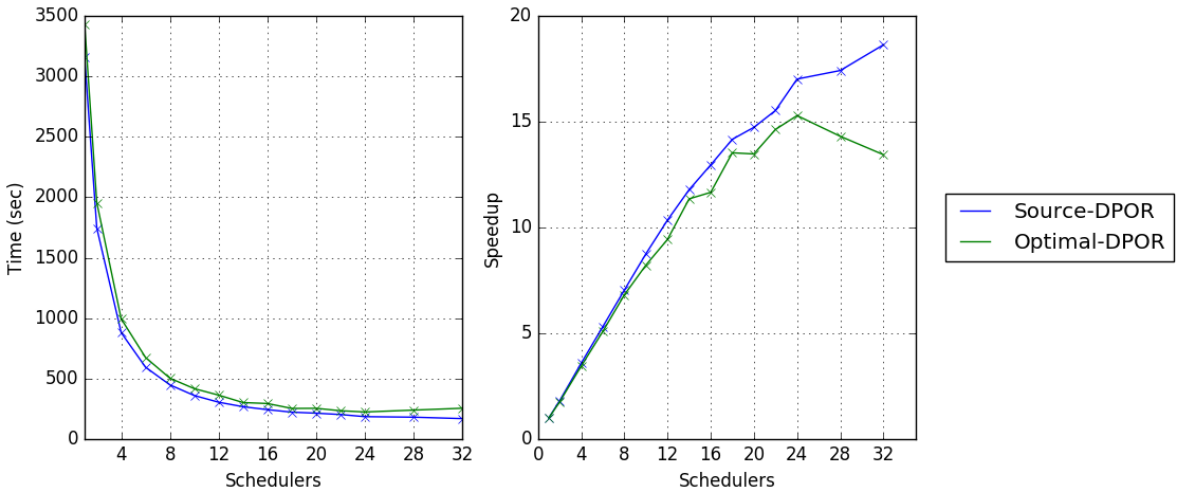
Lets also present graphs depicting the execution time and the speedup ($\frac{T_{serial}}{T_{parallel}}$) of the source-DPOR and optimal-DPOR algorithms for different numbers of schedulers and for various test cases.



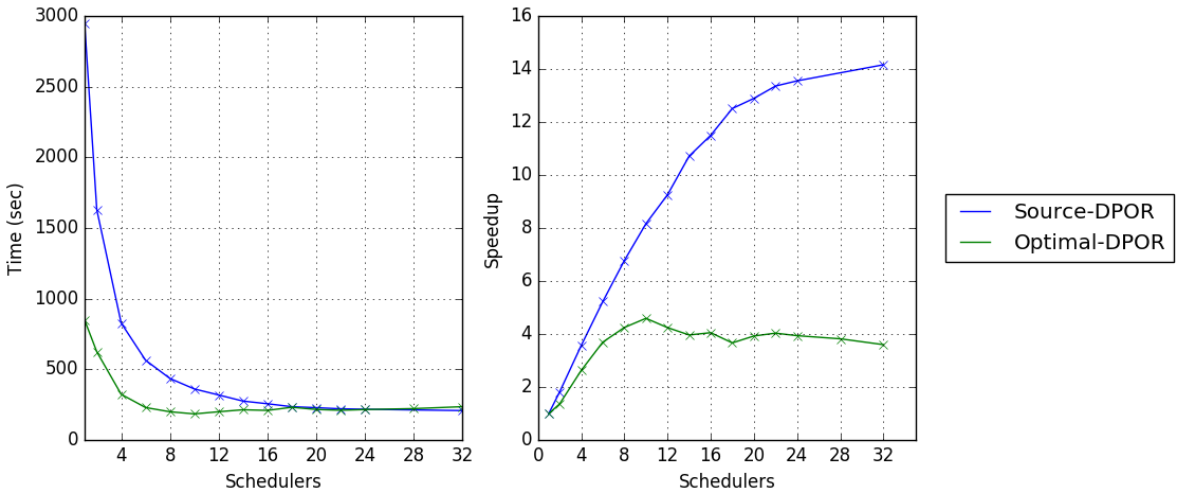
Σχήμα 7.1: Performance of readers 15 with Budget of 10000.



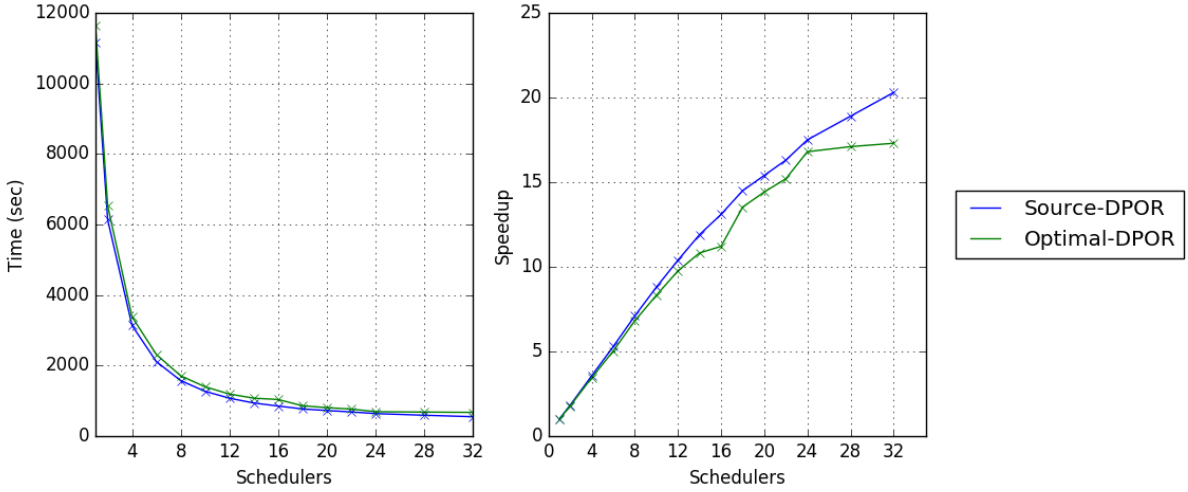
Σχήμα 7.2: Performance of readers 15 with budget of 30000.



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 Performance Analysis

As we can see in our charts, our parallel implementation, both of source-DPOR and of optimal-DPOR, significantly reduces the testing time of our test cases. Namely, in the cases of rush hour and indexer, source-DPOR provides a speedup by a factor of around 1.8 for 2 schedulers and of 3.5 for 4 schedulers and 7.1 for 8 schedulers. In these benchmarks, optimal-DPOR provides a speedup of around 1.7, 3.4 and 6.8 for 2, 4 and 8 schedulers respectively. This makes our parallel implementations highly usable in personal computers.

Additionally, source-DPOR achieves a decent scalability in test cases like rush hour, lastzero 11 and indexer 17. Particularly, with 32 schedulers we manage to speed up Concuerror by a factor of 18.6 for rush hour and 20.3 for indexer 17. However, through the scalability charts we can notice that with more schedulers the scalability of those test cases starts to decline. This happens because we are using a centralized Controller, which becomes a bottleneck, since the more the schedulers, the higher the chance the Controller will be busy and therefore, unable to distribute work. The most important reason behind this drop in scalability though, is the fact that with more available schedulers, the execution sequences of the frontier are partitioned more finely and therefore, the subtrees of the state space assigned to each scheduler, are also more finely grained. A more fine grained assignment leads to the schedulers exploring their assigned subtrees faster and having more races found outside their assigned subtrees. Consequently, the communication with the Controller is more frequent. This situation is problematic for two reasons: the communication between a scheduler and the Controller has a non-negligible overhead and when the amount of communications between the schedulers and the Controller increases, the Controller becomes an even bigger bottleneck. As such, we come to the conclusion that a test case cannot scale beyond a certain point based on the number of its interleavings.

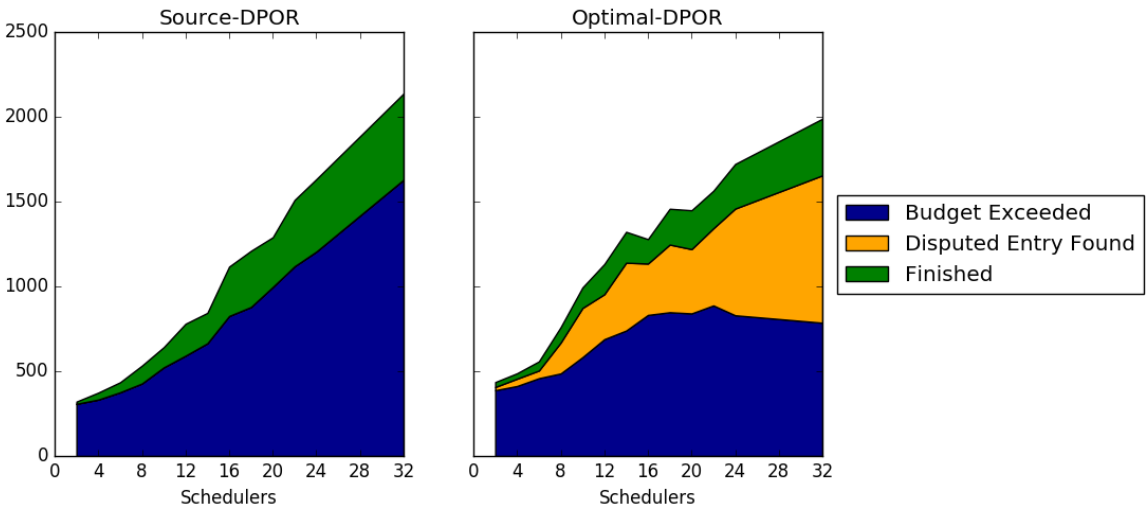
In the case of optimal-DPOR, while we still get decent speedups and scalability, the algorithm seems to stop scaling faster than source-DPOR. For instance, optimal-DPOR with 32 schedulers achieves a speedup of 17.3 for indexer 17, while for rush hour, it has a speedup of 13.4, even though that with 24 schedulers it has a speedup of 15.3. We can notice here the scalability of the algorithm to starts to break. Therefore, the parallel optimal-DPOR algorithm performs worse than parallel source-DPOR. This behavior is expected since:

- In source-DPOR, schedulers would have a subtree rooted at an execution of form $E.p$ (where p is a process) assigned to them. However, in optimal-DPOR schedulers are assigned subtrees rooted at an execution sequence of form $E.w$ (where w is an sequence of processes). Therefore, the subtrees of the state space assigned to schedulers, in optimal-DPOR, are smaller, in general, compared to the ones from source-DPOR. This leads to optimal-DPOR encountering more backtrack entries that have a disputed ownership and therefore, optimal-DPOR has to communicate with the Controller more frequently.

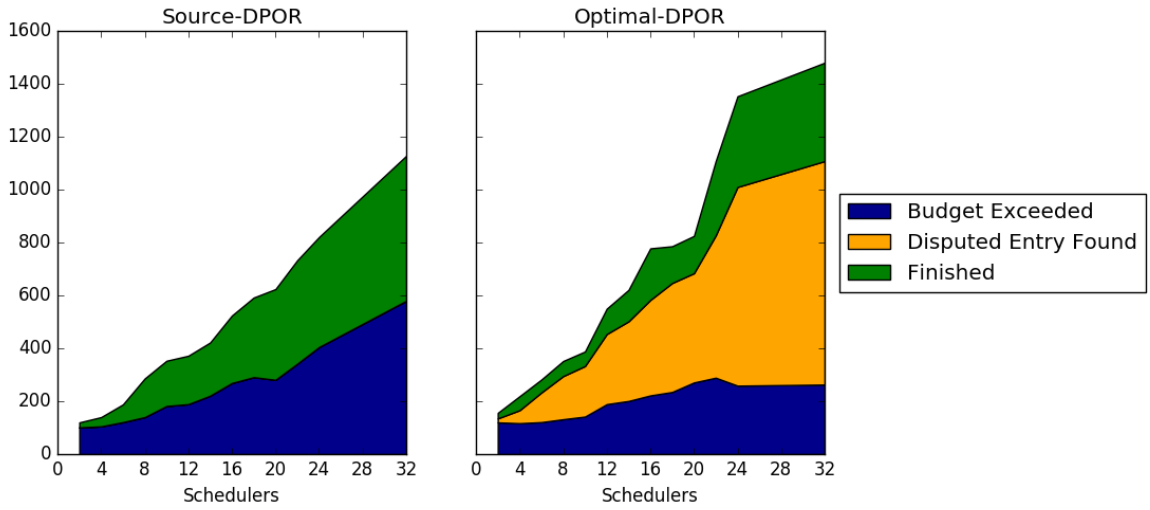
- In source-DPOR, for each disputed backtrack entry, the Controller simply checks whether that entry exists in its execution tree. However, in the case of optimal-DPOR, the Controller tries to $insert$ a sequence of processes into the execution tree. This leads to the Controller of the optimal-DPOR having a higher complexity.

Consequently, in optimal-DPOR, our bottleneck i.e., the Controller, is not only slower, but also there is an increased amount of communication with it. This causes optimal-DPOR to be less scalable than source-DPOR.

From Figure 7.1 we notice that the scalability of readers 15 breaks significantly faster, compared to that of the other benchmarks, despite the fact that readers 15 does not have much less explored traces than lastzero 11 (Table 7.1). While trying to figure out the reasons behind this behavior, we have produced a graph that shows how many times the schedulers communicated with the Controller and why such a communication occurred, depending on the number of the schedulers. In Graph 7.6, we can see that our schedulers exceeding their allotted budget, plays a major factor in the communication towards the Controller. To try and measure the impact of the budget, we tried increasing the budget for both algorithms to 30000ms. The resulting execution times can be seen in Graph 7.2. We notice, source-DPOR performs worse with a reduced budget, due to the fact that its schedulers become more imbalanced, since the Controller repartitions the exploration frontier less frequently. However, optimal-DPOR starts to perform better with an increased budget. As it can be seen from Graphs 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.



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.

In the case of lastzero 11, the scalability of optimal-DPOR breaks significantly faster than source-DPOR. The main reason behind this, has to do with the fact the source-DPOR explores a much larger number of traces. Namely, optimal-DPOR explores 7168 traces, while source-DPOR 60073 (52905 sleep-set blocked interleavings). A larger number of interleavings means that the schedulers have more work to distribute among them. It is important to notice here, that source-DPOR manages to catch up to optimal-DPOR with 16 or more schedulers.

7.4 Final Comments

To sum up, both of our parallel algorithms manage to significantly reduce the execution time of Concuerror and are able to scale to a large number of workers. The performance, however, differs on the various benchmarks, depending, mainly, on number of the interleavings that must be explored. Specifically, the higher the number of interleavings, the better the scalability of our algorithms appears to be.

Source-DPOR appears to scale better than optimal-DPOR, due to the fact optimal-DPOR has a higher communication overhead with the Controller. This leads to parallel source-DPOR outperforming parallel optimal-DPOR on benchmarks with no sleep-set blocked interleavings. This should come as no surprise, since this was also true of the sequential versions.

What is more, our parallel implementation of source-DPOR, with enough schedulers, appears to be able to outperform the optimal version even on test cases with a large number of sleep set blocked interleavings. However, on very large test cases, I do not expect this trend to continue.

Κεφάλαιο 8

Concluding Remarks

In this diploma thesis, we presented the parallelization of Concuerror. In order to parallelize Concuerror, we had to develop parallel versions for its two main algorithms: source-DPOR and optimal-DPOR. Parallelizing the optimal-DPOR algorithm appeared to be particularly daunting. What is more, while making it possible for Concuerror to explore multiple interleavings in parallel, we had to overcome various implementation challenges.

We evaluated our algorithms by using parallel Concuerror to test various programs. During this evaluation, we discovered that our algorithms provide significant speedup on most of our benchmarks. Furthermore, our implementation is able to scale up to at least 32 schedulers, depending on the benchmark. We also noticed that the parallel source-DPOR algorithm is more scalable than the parallel optimal-DPOR algorithm, which was to be expected.

However, there is still work to be done:

- Developing a parallel implementation of Concuerror that works within a single Erlang node, even if it restricts the use of Erlang Pids within the tested programs. This would serve as a lightweight alternative and more importantly, allow for using the sequential Concuerror to test and verify our implementation.
- Test and benchmark our implementation in a distributed setting.
- Modifying our parallel implementation to work with optimal-DPOR with observers.
- Examining and implementing bounding techniques on the parallel DPOR algorithms.

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.