Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Εφαρμογή Μηχανικής Μάθησης για την Πρόβλεψη Επιτάχυνσης Μετασχηματισμών σε Μεταγλωττιστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ ΠΛΑΚΑΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2020

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Εφαρμογή Μηχανικής Μάθησης για την Πρόβλεψη Επιτάχυνσης Μετασχηματισμών σε Μεταγλωττιστές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΙΩΑΝΝΗΣ ΠΛΑΚΑΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 26η Νοεμβρίου 2020.

.....	.....	.....
Νικόλαος Σ. Παπασπύρου Καθηγητής Ε.Μ.Π.	Γεώργιος Στάμου Αν. Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Επίκ. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2020

Ιωάννης Πλάκας

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

Περίληψη

Στη σημερινή εποχή, η ανάγκη για βελτίωση της μεταγλώττισης λογισμικού γίνεται ευρύτερα αντιληπτή. Οι μεταγλωττιστές χρησιμοποιούν προκαθορισμένες σημαίες (π.χ. `-O2`) για να βελτιστοποιήσουν την απόδοση του λογισμικού, είτε ως προς το χρόνο εκτέλεσης, είτε ως προς τη μνήμη και την ενέργεια που καταναλώνει. Κάθε τέτοια σημαία αντιστοιχεί συνήθως σε μια προκαθορισμένη σειρά περασμάτων βελτιστοποίησης που εφαρμόζονται στο μεταγλωττισμένο κώδικα. Παρ' όλα αυτά, δεν είναι λίγες οι φορές που αυτές οι προεπιλεγμένες σημαίες επιφέρουν τα αντίθετα αποτελέσματα στον κώδικα. Αυτό γίνεται διότι κάθε εφαρμογή έχει τις δικές της ιδιαιτερότητες. Συνεπώς αποτελεί καταλυτικής σημασίας η "προσωποποίηση" της μεταγλώττισης έτσι ώστε κάθε εφαρμογή να μεταγλωττίζεται με βάση τα χαρακτηριστικά της.

Σκοπός της παρούσας διπλωματικής είναι η υλοποίηση ενός συστήματος για την πρόβλεψη, κατά τη διάρκεια της μεταγλώττισης, της σειράς περασμάτων βελτιστοποίησης που θα έχει το καλύτερο αποτέλεσμα για το συγκεκριμένο πρόγραμμα που μεταγλωττίζεται. Κάθε πρόγραμμα προσδιορίζεται από ένα σύνολο στατικών χαρακτηριστικών (π.χ. αριθμός αριθμητικών εντολών), τα οποία συλλέγονται χωρίς να είναι απαραίτητη η εκτέλεση του προγράμματος. Στη συνέχεια, με τη χρήση μοντέλων μηχανικής μάθησης γίνεται πρόβλεψη με βάση τα στατικά χαρακτηριστικά της επιτάχυνσης των προς εξερεύνηση βελτιστοποιήσεων. Δίνοντας ουσιαστικά στο μοντέλο πρόβλεψης κάθε πιθανή σειρά περασμάτων βελτιστοποίησης από ένα προκαθορισμένο χώρο αναζήτησης για ένα σύνολο στατικών χαρακτηριστικών κώδικα, επιλέγεται η σειρά περασμάτων βελτιστοποίησης που αντιστοιχεί στη μέγιστη προβλεπόμενη απόδοση.

Λέξεις κλειδιά

Μεταγλωττιστής, Στατική ανάλυση, Μηχανική Μάθηση, Πρόβλεψη επιτάχυνσης.

Abstract

The need for improvement in code compilation is nowadays widely understood. Compilers use predefined flags (e.g., `-O2`) in order to optimize code performance, in terms of execution time, memory or energy consumption. Each such flag typically corresponds to a predefined series of optimization passes, which are applied to the compiled code. Sometimes, however, the predefined flags have negative results and end up deteriorating code performance, instead of improving it. This happens because every piece of code has its own complexities. It is thus critical for every program to be compiled with respect to its own attributes.

The purpose of this diploma dissertation is the implementation of a system for statically predicting the best sequence of optimization passes for the specific code that is being compiled. Every program is characterized by a set of static features (e.g., number of integer operations), that can be collected without executing the program. Subsequently, using machine learning models, a prediction can be made using these characteristics. By giving also as input to the machine learning model a list of all possible optimization sequences from a predetermined search space of compilation passes, a compiler can select the series of optimization passes that corresponds to the best predicted improvement.

Key words

Compiler, Static Analysis, Machine Learning, Speedup Prediction.

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω βαθύτατα τον επιβλέποντα καθηγητή μου κ. Νικόλαο Παπασπύρου, για την υπομονή και καθοδήγηση που μου παρείχε καθ' όλη τη διάρκεια της διπλωματικής αυτής άλλα πάνω από όλα για τις πλούσιες γνώσεις που έχω αποκομίσει από τα μαθήματα και τις διαλέξεις του, χάρις τις οποίες διαμορφώθηκε το ενδιαφέρον μου στον τομέα. Επίσης θα ήθελα να ευχαριστήσω τον Διδάκτορα Θάνο Τάγαρη, για της πολύωρες συζητήσεις και συμβουλές πάνω στα θέματα Μηχανικής Μάθησης που χρειάστηκαν για την εκπόνηση της διπλωματικής. Ακόμα θα ήθελα να ευχαριστήσω από τα βάθη της καρδιάς μου όλους τους φίλους μου, που με στήριξαν καθ' όλη τη διάρκεια της προσπάθειας μου, καθώς γέμισαν τα φοιτητικά μου χρονιά με υπέροχες στιγμές, που θα μείνουν για πάντα χαραγμένες στη μνήμη μου. Επίσης θα ήθελα να εκφράσω την ολόψυχη ευγνωμοσύνη μου στη Νάγια για την πολύτιμη υποστήριξη και συμπαράσταση σε κάθε έκφανση αυτής της πορείας. Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου, που πάντα ήταν και είναι δίπλα μου, σε κάθε μου βήμα. Την μητέρα μου, που πάντα μου έδινε κίνητρο να συνεχίσω να βελτιώνομαι. Τον πάτερα μου, για την εμπιστοσύνη που μου έχει δείξει όλα αυτά τα χρόνια. Την αδερφή μου, για την αμέριστη στήριξη. Τέλος την γιαγιά μου, που πάντα με ενθάρρυνε και μου έδινε κουράγιο από την αρχή της ακαδημαϊκής μου πορείας.

Ιωάννης Πλάκας,

Αθήνα, 26η Νοεμβρίου 2020

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-5-20, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Νοέμβριος 2020.

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

Περιεχόμενα

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Η εργασία αυτή αποσκοπεί στην υλοποίηση ενός συστήματος για την πρόβλεψη, κατά τη διάρκεια της μεταγλώττισης, της σειράς περασμάτων βελτιστοποίησης που θα μεγιστοποιήσουν την απόδοση του κώδικα. Ουσιαστικά, με αυτό το σύστημα ο προγραμματιστής θα δίνει σαν είσοδο ένα πρόγραμμα. Στη συνέχεια, θα γίνεται στατική ανάλυση στα χαρακτηριστικά του προγράμματος που εξάγονται. Τέλος, θα επιστρέφεται ο συνδυασμός των περασμάτων που μπορούν βελτιστοποιήσουν (ως προς τον χρόνο) το πρόγραμμα, με τη χρήση μηχανικής μάθησης.

1.2 Κίνητρο

Καθώς διανύουμε το τέλος του Νόμου του Moore [Moor06], η αυτοματοποίηση της επιλογής σημαιών των μεταγλωττιστών, αποτελεί πλέον αναγκαία συνθήκη για την παραγωγή αποδοτικού λογισμικού. Ουσιαστικά, για να ικανοποιηθούν οι αυξανόμενες υπολογιστικές ανάγκες [Zahr16], αυξάνεται όλο και περισσότερο η χρήση ετερογενών αρχιτεκτονικών (FPGA, GPU, TPU) μαζί με τις CPU. Ωστόσο, για να γίνει αποτελεσματικά η χρήση των ετερογενών αρχιτεκτονικών, απαιτείται η αξιοποίηση Γίνεται λοιπόν αντιληπτό, πως στη σύγχρονη εποχή η αυτοματοποίηση των μεταγλωττιστών για την βελτιστοποίηση του κώδικα αποτελεί κρίσιμο ζήτημα.

Οι μεταγλωττιστές έχουν σχεδιαστεί έτσι ώστε να εφαρμόζουν μετασχηματισμούς στα τμήματα του κώδικα, έτσι ώστε να τα φέρουν σε πιο αποδοτική μορφή. Οι μετασχηματισμοί αυτοί ουσιαστικά εφαρμόζονται σε τρία στάδια: 1) front-end 2) middle-end (Intermediate Representation) 3) back-end. Η βελτιστοποίηση στο επίπεδο του IR μπορεί να έχει το σημαντικότερο ρόλο στην επίδοση του συστήματος. Η επίδοση μπορεί να είναι είτε συνάρτηση του χρόνου [Park12, Park11b, Park14], είτε συνάρτηση της κατανάλωσης ισχύος του συστήματος [Asho14, Asho16b, Park11a]. Σε κάθε περίπτωση, όμως, πρέπει να γίνει κατάλληλη επιλογή των μετασχηματισμών που θα ενεργοποιηθούν (π.χ. loop-unrolling, register-allocation).

Η επιλογή των μετασχηματισμών (selection problem), όσο και η σειρά με την οποία θα εφαρμοστούν (phase ordering problem), αποτελούν τα πιο σημαντικά ερευνητικά πεδία των μεταγλωττιστών [Park11a]. Αυτό συμβαίνει, διότι η πληθώρα των συνδυασμών και των μετασχηματισμών καθιστά εξαιρετικά πολύπλοκη την πρόβλεψη για την επίδοση της επίδραση τους πάνω σε συγκεκριμένα κομμάτια κώδικα. Συνεπώς, ολοκληρωτικές μέθοδοι όπου εφαρμόζουν την ίδια σειρά μετασχηματισμών για κάθε τμήμα κώδικα, μπορεί να επιφέρουν μείωση της επίδοσης (π.χ. -O2, -O3). Επομένως, είναι απαραίτητη η "προσωποποιημένη" εφαρμογή των συνδυασμών μετασχηματισμών για κάθε πρόγραμμα. Δηλαδή, η εφαρμογή τους με βάση τα κύρια χαρακτηριστικά του προγράμματος.

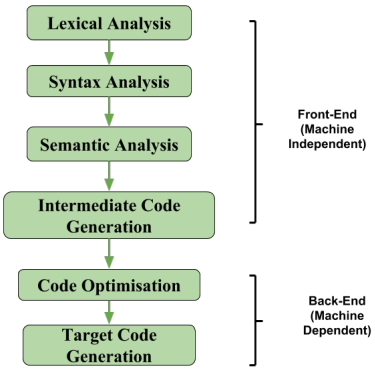
Κεφάλαιο 2

Οι Προκλήσεις στη Βελτίωση των Μεταγλωττιστών

2.1 Φάσεις Μεταγλώττισης

Ο μεταγλωττιστής είναι ένα πρόγραμμα, το οποίο με είσοδο κώδικα σε μια γλώσσα προγραμματισμού, τον μεταφράζει και τον μετατρέπει σε ισοδύναμο κώδικα σε μια άλλη γλώσσα προγραμματισμού. Συνήθως, η μετατροπή αυτή γίνεται από μια υψηλού επιπέδου σε μια χαμηλού επιπέδου γλώσσα μηχανής. Αυτό γίνεται έτσι ώστε να μπορέσει σε επίπεδο κυκλώματος (“hardware”) να μετατρέψει της εντολές του κώδικα σε υπολογιστικές διεργασίες, ώστε να “τρέξει” το συγκεκριμένο πρόγραμμα.

Η διαδικασία μεταγλώττισης χωρίζεται σε τρία μέρη ουσιαστικά: Το front-end, που υλοποιεί την σημασιολογική ανάλυση του κώδικα και θα το μετατρέψει σε μια δομή αναπαράστασης (IR). Το middle-end, που θα φροντίσει να “περάσουν” οι μετασχηματισμοί (flags) πάνω από το IR και να βελτιστοποιήσουν την δομή του. Το back-end, είναι αυτό που ουσιαστικά θα μετατρέψει το κώδικα σε γλωσσά μηχανής κάνοντας και τους απαραίτητους μετασχηματισμούς. Τα επιμέρους τμήματα των διαδικασιών αυτών χωρίζονται σε φάσεις.



Σχήμα 2.1: Φάσεις μεταγλώττισης

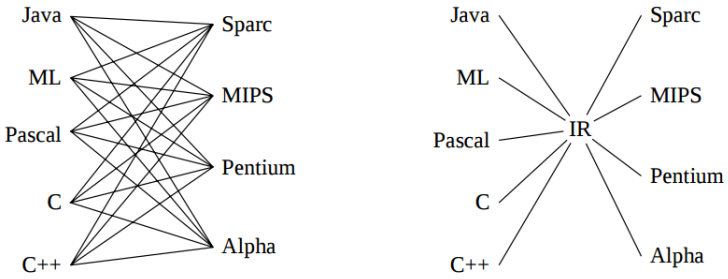
Στη φάση της λεκτικής ανάλυσης, ο μεταγλωττιστής θα δεχθεί ως είσοδο το πρόγραμμα σε μορφή συμβολοσειράς και θα δώσει σαν έξοδο ένα ισοδύναμο πρόγραμμα με τη μορφή λεκτικών μονάδων. Στη φάση της συντακτικής ανάλυσης, θα ελεγχθεί αν το πρόγραμμα ανήκει στη γλώσσα της οποίας η σύνταξη ορίζεται από μια δεδομένη γραμματική. Κατά τη διάρκεια αυτής της φάσης θα κατασκευαστεί το συντακτικό δένδρο. Στη συνέχεια, κατά την σημασιολογική ανάλυση, θα ελεγχθεί η ερμηνεία του προγράμματος. Δηλαδή, θα γίνει στατικός έλεγχος για τον εντοπισμό σημασιολογικών σφαλμάτων. Αφού το πρόγραμμα περάσει και τον σημασιολογικό έλεγχο, ακολουθεί η φάση της παραγωγής του ενδιάμεσου κώδικα. Δηλαδή, η μετατροπή του κώδικα σε μια ενδιάμεση γλώσσα (IR) πριν από την μετατροπή του σε γλώσσα μηχανής. Ουσιαστικά σε αυτό το στάδιο θα εφαρμοσθούν και οι περισσότεροι μετασχηματισμοί για την βελτιστοποίηση του κώδικα ανεξάρτητα από την αρχιτεκτονική την οποία θα τρέξει. Τέλος, η τελευταία φάση του back-end αποτελεί την παραγωγή τελικού κώδικα. Σε αυτή την φάση θα γίνουν οι βελτιστοποιήσεις που συνδέονται με την αρχιτεκτονική όπου θα τρέξει

ο τελικός κώδικας, ώστε να υπάρχει μέγιστη απόδοση.

2.2 Ενδιάμεση Αναπαράσταση

Μετά τον λεκτικό και σημασιολογικό έλεγχο του προγράμματος θα γίνει η μετατροπή σε μια ουδέτερη αναπαράσταση, ανεξάρτητη από την αρχιτεκτονική στην οποία θα τρέξει. Η ενδιάμεση αυτή αναπαράσταση (IR) έχει καταλυτικό ρόλο στη βελτιστοποίηση της απόδοσης του κώδικα είτε σε επίπεδο χρόνου, είτε σε επίπεδο ενέργειας.

Ουσιαστικά, η δομή αυτή αποτελεί μια μορφή κώδικα εντελώς ανεξάρτητη από την γλώσσα που είναι γραμμένο το πρόγραμμα (source language) και από την γλώσσα που θα μεταγλωττιστή το πρόγραμμα (target language). Η ανεξαρτησία αυτή δίνει ευελιξία, καθώς δεν καθίσταται αναγκαίο πλέον κάθε γλώσσα προγραμματισμού να έχει ένα ξεχωριστό front-end για κάθε back-end. Χωρίς αυτό το ενδιάμεσο στάδιο θα έπρεπε, μετά τη λεκτική και σημασιολογική ανάλυση (με βάση τη γραμματική της γλώσσας), να γίνει η μετατροπή του κώδικα σε γλώσσα μηχανής, συμβατή με την αρχιτεκτονική που θα τρέξει. Με την ενδιάμεση δομή αποσυνδέονται, τόσο η παραγωγή γλώσσας μηχανής από τη γλώσσα που είναι γραμμένο το πρόγραμμα, όσο η λεκτική και σημασιολογική ανάλυση από την μέριμνα της αρχιτεκτονικής που θα "τρέξει" το πρόγραμμα. Ωστόσο, το σημαντικότερο προτέρημα της

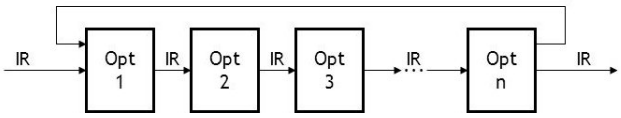


Σχήμα 2.2: Διασύνδεση του IR με το front-end και back-end

δομής αυτής, είναι η δυνατότητα επαναπροσδιορισμού της πριν περάσει στο back-end, έτσι ώστε να βελτιωθεί.

2.3 Βελτιστοποίηση ενδιάμεσης αναπαράστασης

Η ανασύνθεση του IR, που θα συντελέσει στην αποδοτικότερη λειτουργία του κώδικα, συνιστά μέρος του middle-end. Μετά την παραγωγή του IR από το front-end, το middle-end θα έχει το ρόλο της βελτιστοποίησής του, διατηρώντας την σημασιολογική του αξία. Αυτό γίνεται, διότι η παραγωγή του IR από το front-end δεν εγγυάται την βέλτιστη δομική συνοχή του. Επίσης, είναι σύνηθες το φαινόμενο εισαγωγής ανενεργών εκφράσεων ή τμημάτων κώδικα και πολύπλοκων αριθμητικών εκφράσεων από τον προγραμματιστή. Συνεπώς, η απλοποίηση ή απαλοιφή των εκφράσεων αυτών μπορεί να μεταφραστεί σε μείωση των εντολών που πρέπει να μετατραπούν σε γλώσσα μηχανής, χωρίς ωστόσο να υπάρχει σημασιολογική απώλεια. Έτσι, η απλοποίηση του IR αποτελεί κομβική διεργασία για την μεγιστοποίηση της απόδοσης του προγράμματος.



Σχήμα 2.3: Διαδικασία βελτίωσης IR από το middle-end

Η απόδοση ενός προγράμματος και συνεπώς η βελτίωση της μπορεί να κριθεί από ποικίλους παράγοντες. Επομένως, όταν εφαρμόζεται ο κάθε μετασχηματισμός από το middle-end στο IR, θα

πρέπει να είναι ήδη γνωστά τα κριτήρια με τα οποία μετράμε τη βελτίωση αυτήν. Συνήθεις δείκτες της απόδοσης είναι οι εξής:

- **Χρόνος εκτέλεσης**: στόχος είναι να ελαττωθεί ο χρόνος εκτέλεσης του προγράμματος, με ενδεχόμενο κόστος μνήμης και ενέργειας. Μπορεί με αυτό το κριτήριο απλές εντολές να αντικατασταθούν από πιο εξειδικευμένες, για να εκτελέσουν ταχύτερα της υπολογιστικές διεργασίες (π.χ. αριθμητικές πράξεις),
- **Ενέργεια**: Οι μετασχηματισμοί που περνούν πάνω από το IR έχουν τελικό στόχο να κάνουν την κατανάλωση ενέργειας του κώδικα μικρότερη, αντικαθιστώντας σύνθετες εντολές και τμήματα κώδικα που είναι αντιπαραγωγικά. Παράδειγμα αποτελεί η απαλοιφή επαναλαμβανόμενης πρόσβασης σε στοιχείο μνήμης που δεν συμβάλει στην σημασιολογία του προγράμματος.
- **Μνήμη**: Η βελτιστοποίηση με αυτό το κριτήριο αποσκοπεί στην μετατροπή των αντικειμένων ή του τρόπου που αποθηκεύονται στη μνήμη, έτσι ώστε να υπάρχει και μικρότερη χρήση καταχωρητών (register pressure), αλλά και καλύτερη πρόσβαση στα δεδομένα. Η απαλοιφή δασμών μνήμης που εν τέλει δε χρησιμοποιούνται από των προγραμματιστή αποτελεί ένα τέτοιο παράδειγμα.

2.4 Οι Προκλήσεις στη Χρήση Σημαιών Μεταγλωττιστή

Ένα σύνηθες πρόβλημα στο πεδίο τον αλγορίθμων είναι ο τρόπος με των οποίο αξιοποιείται η αλληλεξάρτηση μεταξύ των δεδομένων του προβλήματος. Για παράδειγμα, δεδομένου ενός συνόλου επιλογών, μπορεί να είναι εύκολη η απόφαση για την επιλογή που θα δώσει το μέγιστο κέρδος τη δεδομένη στιγμή που παίρνεται η απόφαση, αλλά όχι σε βάθος χρόνου (Knapsack, TSP). Στους μεταγλωττιστές αυτό αποτελεί ένα από τα σημαντικότερα προβλήματα, καθώς η αλληλεξάρτηση μεταξύ σημαιών μπορεί δύσκολα να αξιολογηθεί [Cast11, Furs11b]. Η κατανόηση της αλληλεπίδρασης των μετασχηματισμών βελτιστοποιήσεις είναι ένα δύσκολο μοντελοποιήσιμο πρόβλημα, καθώς οι σύγχρονοι μεταγλωττιστές έχουν πάνω από 100 περάσματα μεταγλώττισης. Επομένως, η εξαντλητική αναζήτηση για το ποια περάσματα είναι κατάλληλα και με ποια σειρά πρέπει να "τρέξουν", βελτιστοποιώντας την ενδιάμεση αναπαράσταση του κώδικα, αποτελεί το κύριο πρόβλημα για τους συγχρόνους μεταγλωττιστές.

2.5 Το Πρόβλημα της Σειράς Εφαρμογής Σημαιών

Δοσμένου ενός μεγάλου συνόλου μετασχηματισμών, το πρόβλημα είναι ότι δεν είναι εξ αρχής γνωστή η σειρά με την οποία πρέπει να εφαρμοσθούν, ώστε να μεγιστοποιηθεί η επίδοση του προγράμματος. Για παράδειγμα, έστω ενα πέρασμα ***A*** παρ' ότι βελτιώνει την μορφή της ενδιάμεσης αναπαράστασης επιφέρει τέτοιες αλλαγές, έτσι ώστε να μην μπορεί να εφαρμοσθεί ο εξίσου κερδοφόρος μετασχηματισμός ***B*** και αντίστροφα. Αυτού του είδους η εξάρτηση πρέπει να ληφθεί υπόψιν όταν εφαρμόζεται μια αλληλουχία μετασχηματισμών. Αυτού του είδους η εναλλαξιμότητα δίνει το έξης άνω όριο στη πολυπλοκότητα λόγο των μεταθέσεων που μπορούν να γίνουν σε μια αλληλουχία μετασχηματισμών βελτιστοποιήσεις:

$$|\Omega_{phases}| = n!$$

οπού το n είναι ο αριθμός των περασμάτων. Ωστόσο, η παραπάνω εξίσωση δίνει την απλοποιημένη έκφανση του προβλήματος. Επιτρέποντας τις επαναλήψεις στη εφαρμογή περασμάτων και θέτοντας ανώτατο όριο στο μέγεθος της αλληλουχίας των μετασχηματισμών η εξίσωση παίρνει την μορφή:

$$|\Omega_{phases}| = \sum_{i=0}^{l} n^i$$

όπου ***n*** είναι ο αριθμός των μετασχηματισμών και ***l*** δηλώνει το μέγεθος της αλληλουχίας μετασχηματισμών.

Κεφάλαιο 3

Εξαγωγή Χαρακτηριστικών Κώδικα

3.1 Στατικά Χαρακτηριστικά Κώδικα

Η εξαγωγή χαρακτηριστικών από προγράμματα αποτελεί κομβικής σημασίας, ώστε να μπορεί να υπάρχει ένα σύνολο στοιχείων που να αποδίδει τις λειτουργίες τους. Όσο πιο πιστή είναι η δομή αναπαράστασής στο πρόγραμμα, τόσο πιο ακριβή μπορούν να γίνουν τα μοντέλα μηχανικής μάθησης, που θα επεξεργαστούν αυτά τα στοιχεία και θα εξάγουν συμπεράσματα με βάση αυτά. Η στατική ανάλυση ή στατική εξαγωγή χαρακτηριστικών αποτελεί μια μέθοδο για την συλλογή χαρακτηριστικών κώδικα, ανεξάρτητων από την αρχιτεκτονική στην οποία τρέχει. Συνήθως τα εργαλεία που χρησιμοποιούν αυτή τη μέθοδο συλλέγουν τα στοιχεία αυτά από την ενδιάμεση αναπαράσταση κώδικα. Επομένως, γίνεται αντιληπτό ότι η συλλογή αυτών των χαρακτηριστικών δεν απαιτεί την εκτέλεση του κώδικα. Για παράδειγμα, μερικά από τα πιο συνήθη στατικά χαρακτηριστικά είναι ο αριθμός και ο τύπος των εντολών. Επίσης, ο σύνδεσμος των μετρικών αυτών μπορεί να δώσει επιπλέον χρήσιμες πληροφορίες για την δομή του προγράμματος. Παραδείγματος χάριν, το ποσοστό εντολών αποθήκευσης στη μνήμη αποτελεί τη διαίρεση του αριθμού εντολών αποθήκευσης στη μνήμη με των αριθμό εντολών όλου του προγράμματος. Ένα από τα πιο διαδεδομένα εργαλεία για την εξαγωγή χαρακτηριστικών αποτελεί το Milepost GCC [Furs11a, Furs16].

3.2 Δυναμική Εξαγωγή Χαρακτηριστικών

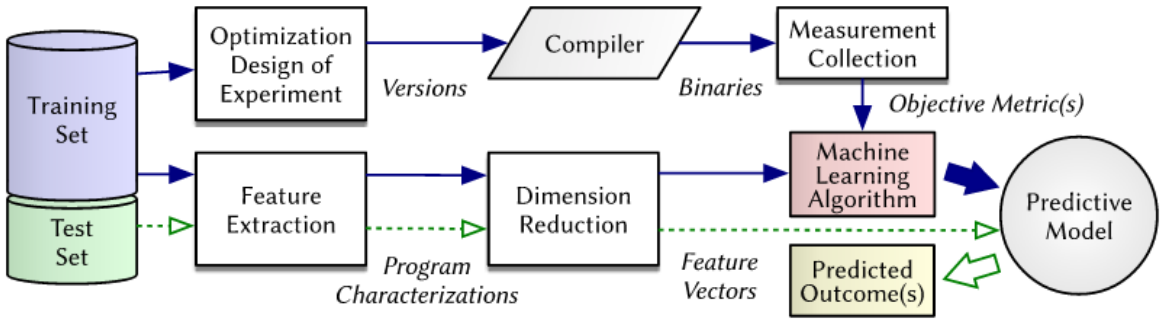
Ένα σημαντικό μειονέκτημα της στατικής ανάλυσης χαρακτηριστικών είναι ότι δεν μπορεί να αποδώσει την συμπεριφορά του προγράμματος ως προς την εκτέλεσή του. Για παράδειγμα, όταν λαμβάνονται υπόψιν ανενεργές εντολές κώδικα στο συνολικό αριθμό εντολών του IR ως feature δημιουργείται λάθος μέγεθος του συνολικού αριθμού εντολών. Αυτό μπορεί να μπερδέψει το μοντέλο πρόβλεψης και να έχει περαιτέρω επίπτωση στην ακρίβεια του μοντέλου. Για την πιο εμπεριστατωμένη αναπαράσταση του κώδικα, αξιοποιούνται στοιχεία της κατανάλωσης πόρων της εφαρμογής σε επίπεδο λειτουργικού. Κατ' αυτόν τον τρόπο πληροφορίες, όπως η επιβάρυνση του επεξεργαστή και αριθμός διεργασιών I/O μπορούν να καταγραφούν. Επίσης, κάνοντας χρήση performance-counters είναι εφικτή η εξαγωγή χαρακτηριστικών, όπως η κατανάλωση ενέργειας της εφαρμογής. Κατά αυτόν τον τρόπο μπορεί να αποκρυσταλλωθεί η συμπεριφορά της εφαρμογής και να δώσει μεγαλύτερη ακρίβεια στο μοντέλο πρόβλεψης. Κατά αυτόν τον τρόπο η δυναμική εξαγωγή χαρακτηριστικών, αποτελεί την εικόνα του προγράμματος, μετά την εκτέλεση του στο επίπεδο της αρχιτεκτονικής και του λειτουργικού.

Κεφάλαιο 4

Μοντέλα Μηχανικής Μάθησης

4.1 Επιβλεπόμενη Μηχανική Μάθηση

Η μηχανική μάθηση αποτελεί την μελέτη των αλγορίθμων που, με βάση ένα σύνολο χαρακτηριστικών, μπορούν να κάνουν προβλέψεις πάνω σε αυτά [Wang09]. Στους μεταγλωττιστές, η διαδικασία της μηχανικής μάθησης μπορεί να χρησιμοποιηθεί, ώστε δοσμένου ενός συνόλου χαρακτηριστικών $\boldsymbol{F}$ και μίας σειράς μετασχηματισμών $\boldsymbol{T}$, να φτιαχτεί ένα μοντέλο πρόβλεψης που να μπορεί να προβλέψει την επίδραση των μετασχηματιστών αυτών. Η έξοδος ενός αντίστοιχου μοντέλου μηχανικής μάθησης θα μπορούσε να είναι η πρόβλεψη επιτάχυνσης της σειράς μετσχηματισμών ή πρόβλεψη της κατανάλωσης ενέργειας. Ουσιαστικά, για να επιτευχθεί η εύρεση της συνάρτησης συσχέτισης των δεδομένων γίνεται χρήση τεχνικών επιβλεπόμενης μηχανικής μάθησης. Η παλινδρόμηση αποτελεί τη στατιστική διαδικασία που χρησιμοποιείται για να σχηματιστεί ένα μοντέλο πρόβλεψης. Κάτα αυτόν τον τρόπο, αποτελεί μια διδικασία αποκρυστάλλωσης της σχέσης μεταξύ των δεδομένων εξόδου με τα χαρακτηριστικά εισόδου (feature vector) του μοντέλου. Για παράδειγμα, έστω ο πίνακας χαρακτηριστικών εισόδου ενός προγράμματος $\boldsymbol{X}$ και ένα μοντέλο παλινδρόμησης που προβλέπει τον χρόνο εκτέλεσης των προγραμμάτων. Εκπαιδεύοντας το μοντέλο κατάλληλα η συσχέτιση των δεδομένων εισόδου με τα δεδομένα εξόδου μπορεί να δώσει ακριβείς προβλέψεις. Κατ' αυτόν τον τρόπο, μέσω ενός νέου διανύσματος χαρακτηριστικών, θα μπορεί να γίνει εκτίμηση της εξόδου, καθώς το μοντέλο έχει μάθει τη συνάρτηση συσχέτισης.



Σχήμα 4.1: Αρχιτεκτονική Μοντέλου Μηχανικής Μάθησης

Κεφάλαιο 5

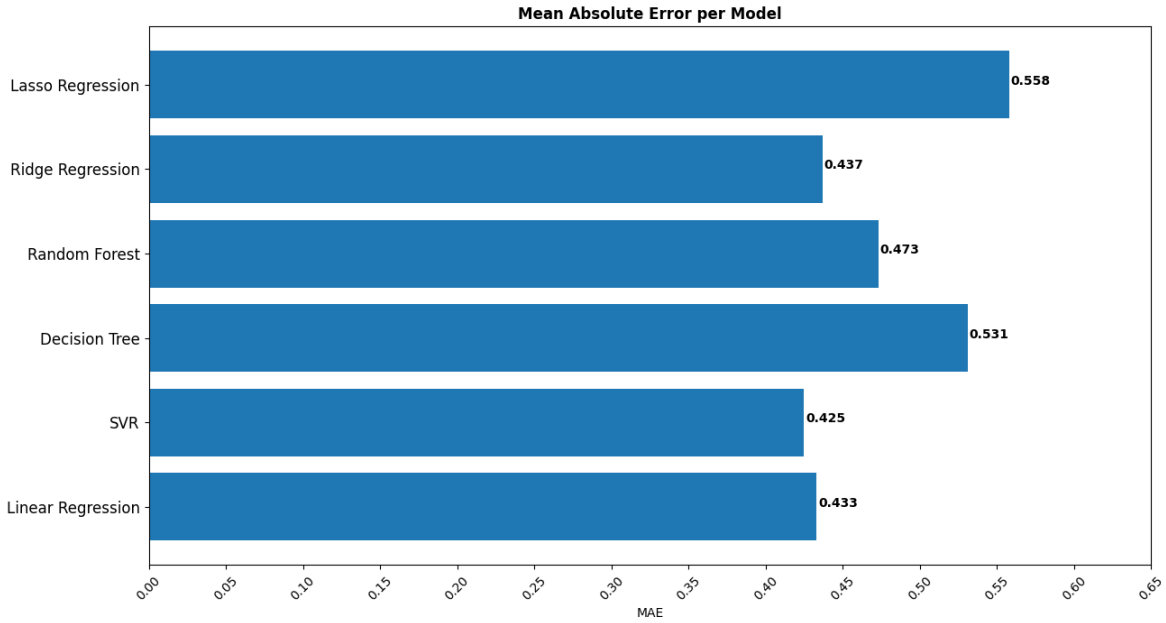
Στατική Ανάλυση στην Επιλογή Σειράς Σημαιών Μεταγλώττισης

Ο χρόνος που μπορεί να πάρει η μεταγλώττιση ενός προγράμματος, καταναλώνει σημαντικό ποσοστό του χρόνου των προγραμματιστών. Τις περισσότερες φορές ωστόσο, χρησιμοποιούνται για τη μεταγλώττιση οι προκαθορισμένες σημαίες (π.χ. -O3, -O2), αντί για τις σειρές περασμάτων, που είναι βέλτιστες για το κάθε πρόγραμμα. Ωστόσο, το σύνολο των πιθανών συνδυασμών σημαιών είναι τόσο μεγάλο, που η εξαντλητική αναζήτηση καθίσταται κοστοβόρα ως προς το χρόνο. Αυτός είναι και ο λόγος της χρήσης, των προκαθορισμένων σημαιών. Για αυτό, στη δουλειά μας ασχοληθήκαμε με την υλοποίηση ενός συστήματος για την πρόβλεψη των πιο υποσχόμενων σημαιών, με βάση τα στατικά χαρακτηριστικά του προς μεταγλώττιση προγράμματος. Για να το πετύχουμε αυτό, αρχικά τρέχουμε σε ένα σύνολο προγραμμάτων ένα επιλεγμένο σύνολο σειρών περασμάτων, ώστε να καταγράψουμε τις επιταχύνσεις που κερδίζουμε. Έπειτα, συλλέγουμε και τα στατικά χαρακτηριστικά των προγραμμάτων, αυτών και προπονούμε έναν αλγόριθμο μηχανικής μάθησης με τα στατικά χαρακτηριστικά και τις επιταχύνσεις. Κατά αυτόν τον τρόπο, ο αλγόριθμος μηχανικής μάθησης συσχετίζει τα χαρακτηριστικά και τις σειρές περασμάτων που εφαρμόζονται στα προγράμματα με τις επιταχύνσεις. Επομένως, με βάση τις προβλέψεις του αλγορίθμου ο προγραμματιστής μπορεί να οδηγηθεί στις σειρές μετασχηματιμών που πραγματικά έχουν τη βέλτιστη απόδοση.

5.1 Σχεδιασμός Πείραματος

Για να μπορέσουμε να ελέγξουμε διαφορετικές σειρές περασμάτων μεταγλώττισης, χρησιμοποιήσαμε το μεταγλωττιστή Clang. Ουσιαστικά, με τα εργαλεία που αποτελούν κομμάτι του συγκεκριμένου μεταγλωττιστή (π.χ llvm-opt), δημιουργήσαμε της δικές μας σειρές μεταγλώττισης ώστε να μπορέσουμε να εκτιμήσουμε πώς οι διαφορετικοί συνδυασμοί επιδρούν πάνω στα προγράμματα. Επίσης, καταλυτικής σημασίας είναι ότι η εξαγωγή χαρακτηριστικών γίνεται στο επίπεδο του IR, οπότε με τη χρήση εργαλείων όπως το llvm-opt μπορούμε να έχουμε το IR του προγράμματος μετά την μεταγλώττισή του. Για την εξαγωγή χαρακτηριστικών χρησιμοποιήθηκε το Milepost GCC. Τροφοδοτώντας το, ουσιαστικά, με τα IR των προγραμμάτων που έχουν μεταγλωττιστεί μπορούμε να εξάγουμε τα χαρακτηριστικά κάθε εφαρμογής. Τα χαρακτηριστικά αυτά, μαζί με την επιτάχυνση λόγω των μετασχηματισμών που έχουμε επιβάλει, είναι τα δεδομένα εισόδου του αλγόριθμου μηχανικής μάθησης που εκπαιδεύσαμε. Κομβικής σημασίας αποτελεί η δημιουργία των σειρών μετασχηματιμών, καθώς η συνεχής μεταγλώττιση των προγραμμάτων και η εκτέλεση με αυτές της σημαίες. Για να έχουμε ένα επαρκές σύνολο συνδυασμών μετασχηματισμών, επιλέγουμε πέντε βασικούς μετασχηματισμούς του LLVM, όπου οι συνδυασμοί τους με μέγεθος 5 θα δώσει τον χώρο των συνδυασμών μετασχηματισμών που εξετάζουμε. Οι μετασχηματισμοί που επιλέξαμε είναι οι ακόλουθοι:

- **mem2reg:** Το IR ξαναγράφεται απαλείφοντας τις διεργασίες αναφοράς στη μνήμη και εισάγοντας τη χρήση καταχωρητών.
- **dce:** Διαγράφει τα κομμάτια του κώδικα στα οποία δεν θα φτάσει ποτέ η εκτέλεση του προγράμματος ή δεν επηρεάζουν το τελικό αποτέλεσμα του κώδικα.



Σχήμα 5.1: Μέσο Απόλυτο Σφάλμα ανά Μοντέλο

- **loop-unroll:** Μετατρέπει της συνθήκες των βρόχων, έτσι ώστε να χρειαστούν λιγότερες επαναλήψεις.
- **instcombine:** Θα συνδυάσει απλές εντολές σε λιγότερο σύνθετες, διατηρώντας τη σημασιολογική αξία του προγράμματος.
- **constprop:** Αποτιμά και αντικαθιστά τις τιμές στις πράξεις με σταθερές (π.χ add, sub).

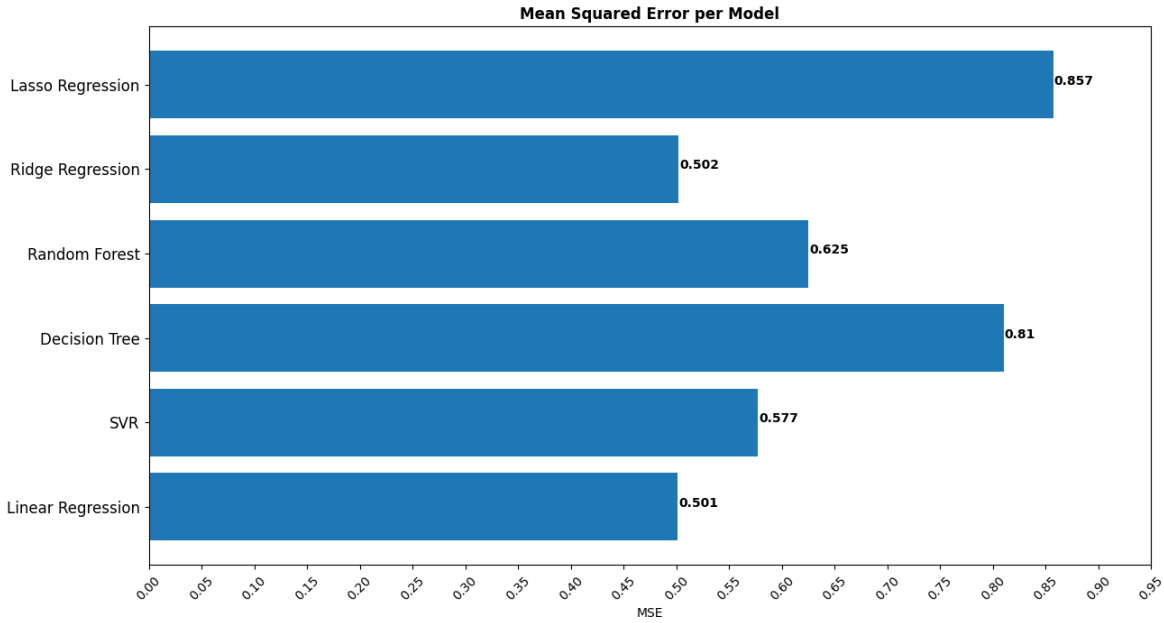
5.2 Αποτελέσματα

Για να αξιολογήσουμε το σύστημά μας, εκπαιδεύσαμε με τα δεδομένα μας μοντέλα μηχανικής μάθησης, ώστε να αποτιμήσουμε την χρήση τους, στο πεδίο της πρόβλεψης επιτάχυνσης μεταγλωττιστών. Αυτό διότι, κάθε μοντέλο αποκτά γνώση από τα δεδομένα που εισάγουμε με διαφορετικό τρόπο. Στη διαδικασία που ακολουθούμε κάθε πρόγραμμα ανήκει στο σύνολο προγραμμάτων επαλήθευσης (test-set) μεταγλωττίζεται με τον συνδυασμό περασμάτων που προβλέπεται να έχει τη μεγαλύτερη επιτάχυνση. Επομένως, αποτελεί κομβικής σημασίας να εξακριβωθεί το καταλληλότερο μοντέλο. Οι αλγόριθμοι μηχανικής μάθησης που εξετάστηκαν είναι οι ακόλουθοι: Linear Regression, Lasso Regression, SVR, Decision Trees και Random Forest.

5.3 Μελλοντικές Επεκτάσεις

Η σχέση που έχουν οι μετασχηματισμοί και οι βελτιστοποιήσεις μεταξύ τους και ο τρόπος που επιδρούν πάνω στα προγράμματα αποτελεί ένα από τα ανοιχτά προβλήματα στο πεδίο των μεταγλωττιστών. Στη διπλωματική αυτή, εξετάσαμε το πρόβλημα της βέλτιστης επιλογής περασμάτων, αξιοποιώντας μεθόδους στατικής ανάλυσης. Ωστόσο, με την πάροδο των χρόνων, αναπτύσσονται νέες τεχνικές που μπορούν να βοηθήσουν στην βελτίωση της επιλογής μετασχηματισμών, όπως οι μέθοδοι επεξεργασίας φυσικής γλώσσας. Μεσά από αυτή την εργασία αναδεικνύονται τα εξής περαιτέρω θέματα:

- Από τα αποτελέσματα μας είναι εμφανές ότι για να έχουμε πιο ακριβείς προβλέψεις χρειαζόμαστε περισσότερα δεδομένα. Έχοντας ένα μεγάλο σύνολο δεδομένων οι αλγόριθμοι μηχανικής



Σχήμα 5.2: Μέσο Τετραγωνικό Σφάλμα ανά Μοντέλο

μάθησης θα μπορούν να δώσουν με μεγαλύτερη ακρίβεια τα αποτελέσματα, καθώς θα έχουν αποκρυσταλλωθεί οι σχέσεις μεταξύ των δεδομένων εισόδου-εξόδου. Στην κατασκευή ενός συνόλου δεδομένων προγραμμάτων είναι σημαντικό να ληφθεί υπόψη η διαφορετικότητα των χαρακτηριστικών των προγραμμάτων, ώστε να μην επαναλαμβάνεται αυτή η γνώση στην εκπαίδευση των μοντέλων. Η εκτέλεση των προγραμμάτων θα πρέπει επίσης να ληφθεί υπόψη, καθώς η διαδικασία της εξαγωγής των επιταχύνσεων είναι χρονοβόρα, συναρτήσει του μεγέθους του χώρου μετασχηματισμών.

- Παρ' ότι τα στατικά χαρακτηριστικά των προγραμμάτων μπορούν να συλλεχθούν γρήγορα, δεν αποδίδουν τη λειτουργία των προγραμμάτων σε βάθος. Η χρήση, ωστόσο, δυναμικών χαρακτηριστικών μπορεί να αποτελέσει μια πιο περιεκτική μορφή απεικόνισης. Η μορφοποίηση αυτή των προγραμμάτων μπορεί να δώσει σημαντική γνώση στα μοντέλα και να βελτιώσει τις επιδόσεις τους, καθώς οι αλληλεπιδράσεις των μετασχηματισμών θα αποκωδικοποιηθούν πιο καθαρά. Ακόμα, ο συνδυασμός των στατικών με τα δυναμικά χαρακτηριστικά θα μπορούσε να δώσει ακόμα καλύτερες εκτιμήσεις.

- Η εξελικτική φύση της διαδικασίας σχηματισμού σειρών μετασχηματισμών κώδικα αποτελεί ιδανικό περιβάλλον, για την χρήση γενετικών αλγορίθμων. Δημιουργώνας την σειρά βήμα βήμα, αντί να προβλέπεται εξαρχής, αποτιμώνται καλύτερα οι αλληλοεξαρτήσεις των μετασχηματισμών. Ωστόσο, σε αυτή τη μέθοδο σημαντική είναι επιλογή του αρχικού πληθυσμού του αλγορίθμου, ώστε η συνάρτηση βελτίωσης να μην "κολλήσει" σε τοπικό ελάχιστο.

- Τέλος, κομβικής σημασίας αποτελεί η αρχιτεκτονική πάνω στην οποία θα τρέξει κάθε πρόγραμμα. Αυτό διότι, η εφαρμογή ορισμένων μετασχηματισμών μπορεί να έχει διαφορετικά αποτελέσματα ανάλογα με τις δυνατότητες που μπορεί να προσφέρει η αρχιτεκτονική πάνω στην οποία τρέχει. Σύνηθες παράδειγμα αποτελούν οι αρχιτεκτονικές που επιτρέπουν ουσιαστικά την εφαρμογή μετασχηματισμών παραλληλοποίησης προγραμμάτων.

Κείμενο στα αγγλικά

Chapter 1

Introduction

1.1 Objective

The purpose of this diploma dissertation is the implementation of a system that statically predicts the sequence of optimization passes that maximize the performance of the specific code being compiled. In that way, by giving as input the code of the program, its features are extracted via static analysis and are propagated to a machine learning model in order to output the most promising optimization sequences.

1.2 Motivation

The automation of selecting the best compilation passes in compilers is essential for producing efficient software, especially as we are approaching the end of Moore's law. Compilers are designed to apply optimization passes in order to make the software more efficient. There are 3 main parts that compose the functionality of compilers: 1) front-end, 2) middle-end (IR) and 3) back-end. Optimization of the IR has a major impact on the efficiency of the compiled code. That kind of improvement can be measured in terms of time or energy consumed. In every case there must be precise selection of enabled passes that will run over the IR, so that significant gains can be achieved.

The selection of optimization passes (selection problem) and the order in which they are applied (phase ordering problem) form the most important research problems in the field of compilers. This happens because there are many combinations of passes that result in an enormous search space. Notably, the use of predetermined flags (e.g., -O2, -O3) can have significant drawbacks in terms of performance in some cases, although they have been widely adopted.

Thus, it is essential for compilers to produce optimization sequences that depend on features of the programs being compiled.

1.3 Thesis Structure

The content of the thesis is structured as follows:

- **Chapter 2:** An introduction to some of the core concepts of compiler theory and functionality of optimization. Also a brief overview of the phase ordering problem and phase selection problem.
- **Chapter 3:** An analysis of feature extraction techniques for code characterization.
- **Chapter 4:** An interpretation of machine learning models used in compiler's research problems.
- **Chapter 5:** Implementation details of our proposed system for static feature extraction from code and speedup prediction of optimization sequences.
- **Chapter 6:** Concluding remarks and future implementations and extensions to our proposed solution.

Chapter 2

Challenges in compiler's optimization

2.1 Compilation Stages

The compiler is a program that transforms the input programming language to machine language or code that computer processors use. In other words it translates code from a high-level language to a low-level language with equal semantic value. That happens because in order for code to be executed instructions have to be "understood" by the hardware so that its process and functions can run on a various architectures.

Compilation processes consist of three main parts: front-end, middle-end and back-end. Front-end is the part that implements the lexical and semantic analysis of the code and transforms it to an intermediate representation (IR). Middle-end applies the optimization passes to the IR, so its structure and the execution performance can be improved. Finally the back-end is the part that translates the IR into low level machine language. The three main parts can be separated in phases.

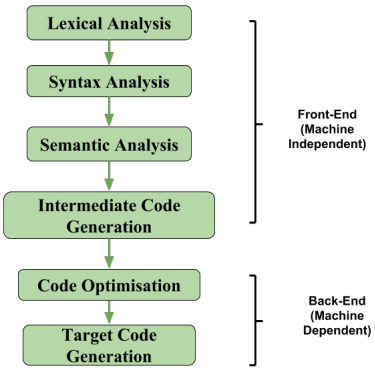


Figure 2.1: Phases of Compiler

During lexical analysis, the code is examined as a series of characters, and it is reformed as a structure of lexical tokens. During syntax analysis, this structure is evaluated in order to be determined whether the predefined grammar rules are followed. After the correctness of program has been verified, a syntax tree is generated. During the phase of the semantic analysis, the correctness of the syntax tree is verified so that semantic value of the program is to be validated. Then generation of the IR takes place. During this phase, the architecture independent optimizations are applied so that the performance of the program is improved. Finally, back-end translates the intermediate representation form into a machine language compatible with the architecture that will run the code.

2.2 Intermediate Representation

After the semantic and syntax analysis have been made, the program is transformed into the intermediate representation(IR), which is independent from the architecture that will run the code. It also plays a major role in improving the performance in terms of execution time or energy consumption.

Mainly IR is a structure that capture the semantic value of source code despite the fact that is independent from the source and target language. This kind of flexibility is important because there is no need for a back-end for every front-end. Without this structure, after the lexical and syntax analysis thatare based on the grammar rules, code would be translated into the low level language depending on the architecture that would run the code. Which means that for every different architecture another compiler would be needed. But rather with the use of IR, there is a common formation for every different front-end.

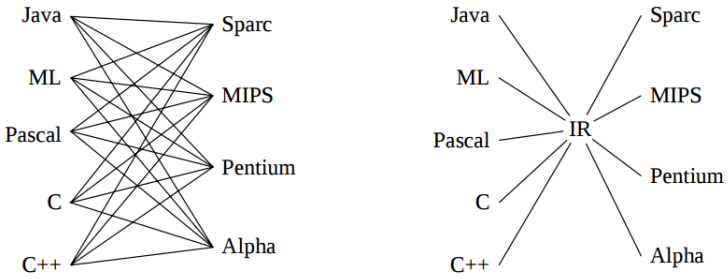


Figure 2.2: Conncection between front-end and back-end

Most importantly this kind of structure can be optimized, before entering the back-end is entered.

2.3 Performance Improvement

The reformation of IR through passes results in the better performance of code execution. After the IR has been generated from the front-end, the middle-end applies the passes selected from the programmer. Thus, although the IR undergoes changes, it keeps its semantic value. This happens because after the IR has been generated, there can be redundancies inserted from the programmer or expressions that do not add any semantic value to the program, thus their elimination results in the better performance of the code. Hence, it is important to simplify the IR form in order to make the low level code more efficient.

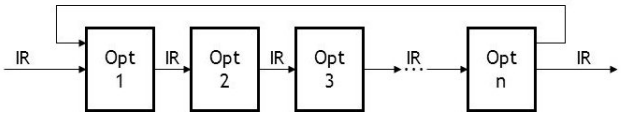


Figure 2.3: IR optimization

The performance of code and furthermore its optimization can be determined by a number of factors. In that spirit when a pass is applied to the IR, the goal of the optimization and the criteria that have to be met must be predefined. In most cases the performance indicators are the following

- **Execution time**: The goal is to reduce the time that the program needs in order to execute with the trade-off of memory and energy consumption. For instance simple instruction can be replaced with, more purpose-built instructions because they can execute computations faster (e.g., arithmetic operations).
- **Energy**: The optimization passes that run over the IR will result in lower levels of energy consumed, while the code is being executed. This happens by replacing heavy instructions with simpler ones that do not need many hardware operations, and by eliminating parts of code that are inefficient and can lead to such results. For example, iterative memory loads that do not add semantic value to the program can be eliminated.
- **Memory**: Optimization in terms of memory happens by transforming the formation of the objects or the way they are stored in memory, so that memory accesses are more “affordable”.

2.4 Optimization Passes

The following examples are some of the most common optimization-passes:

Despite the different factors that can affect the optimization of the IR, every compiler has various passes in order to achieve this kind of performance improvement. GCC has more than 300 different passes. Thus, the selection of combinations that can result in speedup or efficiency in terms of memory is not a trivial task. However there are passes that result in improvements of IR structure most of the time. The examples below are some of the most common optimization-passes:

Constant Folding

The basic idea behind this optimization is that, if an expression has constant operands, they can be evaluated statically at compile time, without adding the overhead of runtime execution of the evaluation and thus improving the runtime performance.

```
x := 42 + 17
y := 17 * 0
z : = 42 * 1
```

```
x := 59
y := 17
z := 42
```

Algebraic simplification A very similar pass to constant folding is the algebraic simplification. Here, their difference lies in that the algebraic simplification uses mathematical and logical simplification rules, despite the fact that the operands might not be constant.

```
a * 1
a * 0
a + 0
b | false
b && true
```

```
a
0
a
false
b
```

Constant propagation

In this optimization variables are replaced with their value, if they are known to be constant. The capacity of improvement in terms of efficiency is drastically increased if its used with the above passes. This kind of improvement is shown in the below example

```
x := 5
y := x * 2
z := a[y]
```

```
y:= 5 * 2
z := a[y]
```

```
y:=10
z:=a[y]
```

```
z:=a[10]
```

Copy propagation

When a variable is assigned to another, this means that every call of the assigned variable will reflect to the copy variable. By replacing this kind of indirect reference to the actual variable a significant amount of speed-up can be gained. The example below shows that.

Dead code elimination After programmers have practiced various optimization passes fallaciously, many statements may be inactive, which means that they do not add any semantic value to the aim of the source code. Thus, the removal of those parts will not only retain the reasoning of the code but also cut out all useless operations. Such an action may not only benefit the code size and speed-up but

```
x := y
while (x < 42){
        x := f(x-17) + x
}
```

```
x :=  y
while (y < 42){
        x:=f(y-16) + y
}
```

also enable other optimization to run and improve further the performance. In the example below, it becomes conspicuous how helpful this optimization can be.

```
x := 15
y : = 42
x := x* y ;
return (y)
x := x - 17
return (x)
```

```
y:=42
return (y)
```

Common subexpression elimination The objective of this pass is to replace instances of expressions that are reused, while variables can be holding the computed value. The benefit of performing such actions is that there is no need to reevaluate expressions that have already been examined, hence there can be major speed-up improvements.

```
z := x + y +42
w := x + y + 17
m := z + w
```

```
tmp := x + y
z := tmp + 42
w := tmp + 17
m := z + w
```

Loop unrolling The concept of this optimization is to reduce the iterations of loops. This is achieved when/if the instruction that control the loop is minimized or eliminated. In that respect loops may be rewritten, by repeating instructions to keep their semantic value unaffected.

```
for (i =0 ; i < 50 ; i++){
        print(”Hello World”)
}
```

```
for (i=0 ; i < 50 ;i+10){
        print(”Hello World”)
        print(”Hello World”)
        print(”Hello World”)
        print(”Hello World”)
        print(”Hello World”)
}
```

Although the word optimization indicates improvement in terms of performance, this is not always the case, in fact aggressive optimizations can degrade the performance of the code [Tria03]. Consequently there is no guarantee that the optimization will pull off a better version of the source code. In order for the middle-end to deliver the promised goods via performing a series of optimizations, their interaction must be taken into account. This kind of unsolved research challenges in compiler’s domain are what engineers have been focused to untangle as processors can’t keep up with the Moore’s law promises [Esma11].

2.5 Challenges of Compiler Optimizations

A common problem in the field of algorithms is the way of handling inter-dependencies. For example, given a set of choices it might be trivial to take the best option that will give greatest value right away, but it is not easy to say what choices overall that could be made (Knapsack, TSP). In a compiler's domain that is what has been a major problem: the interdependency of optimization passes, something is difficult to evaluate [Furs11b, Vegd82]. Understanding the behavior of optimizations with each other are complex modeling problems due to the fact that hundreds of different optimizations must be considered during the various compilation phases. For example, GCC and LLVM-CLang have more than 100 optimization passes. Along these lines, it is clear that in order to have a better selection, a wide search of solution space is needed. In the compiler's theory: "feasible set, search space, or solution space is the set of all possible points of an optimization problem that satisfy the problem's constraints" [Steu08]. To optimize the IR a huge search space is usually pruned and down sampled with the optimization algorithm. So, finding which optimizations to use, which set of parameters (e.g., loop-unroll factor) and in which order to "run" them over the IR so to yield the best performance improvement is the main challenge to advance the compiler's capabilities. At this point, it becomes obvious that the optimization problem can be split into two sub-problems: (i) to decide how to enable/disable a set of optimizations (optimization selection problem); or (ii) to change the ordering of optimizations (phase ordering problem).

Given several optimization passes, the enabling or disabling a part of them, can form various optimization sequences. When the ordering is ignored the problem scope is which optimizations have the best impact on the source code in terms of performance. Researches have shown that interaction between enabling or disabling optimisations in a sequence can drastically improve the performance without even taking in consideration the order of phases [Asho16b, Bodi00].

Having a search space O with size n, an optimization element o_i can be either enabled *(*$o_i = 1$*)* or disabled *(* $o_i = 0$ *)* . Hence an optimization sequence can be specified by a vector with n dimensions:

$$|\Omega_{selection}| = \{0, 1\}^n$$

So the selection problem's optimization space has an exponential space as its upper-bound, which means when there are ten optimizations to be taken in consideration ($n = 10$) the search space to find the best sequence without ordering equals 2^n (1024). Also another thing that must be taken in consideration is that some passes may need additional parameters, like loop unrolling (factor of tuning 2, 4, 8, 16). In that case it the total number of choices a compiler optimization can have is m, like so the above equation takes the following form:

$$|\Omega_{selection\_extented}| = \{0, 1, , ..m\}^n$$

2.5.1 Phase Ordering Problem

Given a set of optimizations the problem is that there is no ideal ordering of phases. An optimization pass ***A*** can transform the IR in a way that prohibits the effect of some optimizations which otherwise could have been performed from the following pass ***B***. Moreover if ***B*** is enabled before ***A*** its effect might be dull. This kind of linkage must be considered, before multi-phase optimizations have been applied. "A set of optimization phases may be interdependent in the sense that each phase could benefit from transformation produce by the other" [Leve80].

This kind of interchangeability gives a factorial upper bound due to permutations of the phases an optimization sequence can have:

$$|\Omega_{phases}| = n!$$

where n is the number of optimizations that should be taken into account. However the equation above is a simplified version of the phase ordering problem. By allowing various length sequences

and repetitions of optimizations the search-space equation has the following form:

$$|\Omega_{phases}| = \sum_{i=0}^{l} n^i$$

where n is the number of optimizations under study and l is the maximum desired length for optimization sequences. Even for small-scale values of n and l the search space is massive. For instance if n and m are both equal to 10, they lead to an optimization search space of more than 11 billion different optimization sequences. The problem of finding the right phases does not have a deterministic upper-bound given an unbounded optimization length [Asho16a].

2.5.2 Autotuning

In order for the autoation of processes to accomplish one or more objectives with minimal or no interaction with user is called autotuning [Wils94]. In the compiler's design autotuning usually refers to the implementation of a model where a tuning parameter space is defined. Many versions of the input program are created in this parameter space. The exploration of those variants is called iterative exploration, which can be done exhaustively either when all different versions of the program are considered or when a subset of the space is sampled by using heuristic search based on exploration. Thus, there is a trade-off between the number different versions used to construct the model (accuracy) and the computational complexity to construct the model (time).‘ The goal to compose such models is defined in terms of performance. Knowledge extracted from different applications and their variants are fed in machine learning algorithms from which a predictive model is emitted. By extracting features from "unseen" applications, a train set can be formed and fed into the predictive model. A predicted outcome is yielded which can be compared to the known outcome so that error measurements can be evaluated and predicted. Therefore, feature engineering plays a vital role in that process. Getting a representation of a program and feeding it in a machine learning model is a complex problem. Thus lot of research has been done [Cumm17] to find ways to characterize programs, in a way that its semantic value be captured in a representation, that can be fed to machine learning models.

Chapter 3

Feature Extraction

3.1 Static Features

Given that there is no upper-bound to potential features to characterize a program, composing a high-quality set of them is a non-trivial task. Complexity of the problem is increased when machine learning models have restrictions in order to be fed with data (e.g., fixed length inputs) are taken into consideration. The accuracy of outcome predictions depends heavily on the train set that formed these machine learning models. “After all you are as good as your data” [Goog].

Static analysis or static feature collection is a method that tries to collect features which are independent from the architecture that runs the code. Commonly such features are extracted from compiler intermediate representations [Wang09, Tayl17]. Hence, feature vectors of programs can be obtained without the source code being executed; for example, two frequently used features are the number and the type of instruction. Likewise raw features can be combined to generate new features, such as memory load ratio that is yielded when the number of load instruction is divided by the number of total instructions. One of the most used static feature extractors is Milepost GCC [Furs11a] which was propose by Fursin *et al.*, as a plugin to GCC compiler.

Another way for programs with respect to their semantics to be represented is by graph representations. In such representations data and control dependencies are explicit for each program operation. “Data graphs have provided an explicit representation of the definition-use relationships implicitly present in a source code” [Ferr87]. Control flow relationships are usually represented by control flow graph [Alle70]. Some tools that can extract these kind of dependencies (data flow and control flow) are MinIR [Asse11] and LLVM. A distinct graph based method for feature representation was introduce by Park *et al.* [Park12]. The authors used MinIR to extract the control flow graph of many applications and composed feature vectors by using shortest path graph kernels. In that fashion they where able to feed the features into a SVM model. Another graph based approach was introduced by Koseki *et al.* [Kose98]. Their method used control flow graphs as feature vector to predict good loop unrolling factors. A different approach was used by Nobre *et al.* [Nobr16] in order to target the phase ordering problem. In their graph based approach each node of the graph is an optimization pass and edges are weighted, so the sub-sequences with higher aggregated weights are more likely to achieve a better performance.

3.2 Dynamic Characterization

A major drawback when using static analysis to collect features and characterize applications is that the actual behavior of the program might not been captured. For example, when the instructions of a basic block in the IR which is not executed are taken into account, they create an illusory feature of values. This can confuse the prediction model and further decline its accuracy. In order to have a more robust representation of the source code in terms of behavior (I/O, resource contention), features should be extracted from different layers of runtime environment. Information, like loop iteration counts and frequently executed code regions, can be obtained when the runtime trace of the application is examined, given its input. Furthermore, at the lever of operating systems metrics like CPU load and

I/O magnitude can mirror the behavior of the application. Last but not least, at the hardware level performance counters can track down information such as number of cache loads/stores and branch misses [Jeff12]. Thus, energy consumption and hardware resources needed from the application can be tracked down. This kind of low-level access to "behind the scenes" behavior of the application can drastically increase the accuracy of predictive models. Because of this feature vectors of dynamic analysis tend to have a hardware-specific view of application behavior. Furthermore, execution pattern of the program might change with different inputs or architecture that will run the source code, thus predictions will be drawn on unreliable observations.

3.3 Feature Generation

The rise of machine learning has solved scientific problems across different domains of programming languages. In the field of Program Synthesis Xiao Liu *et al.* [Liu19] who based on a natural language processing model, composed a C-program generator. By generating well-formed C-programs, the researchers where able to perform fuzz-testing to GCC and CLang/LLVM compilers. In the field of code summarization Allamanis *et al.* [Alla15] created a model inspired by Milikovs natural language embeddings [Miko13] that was able to suggest variable and method names given the source code. Models accuracy depended heavily on "learning" the context of variables and methods that would be renamed. In this fashion Uri.aln *et al.* constructed a neural model for representing snippets of code as continuous distributed vectors [Alon18].

Feature extraction has also been benefited form the use of such neural models and natural language processing techniques. According to the Naturalness hypothesis [Alla18] forms of code could be treated as text: "Software is a form of human communication software corpora have similar statistical properties to natural language corpora and these properties can be exploited to build better software engineering tools". The first who used natural language processing techniques to extract features from source code for compiler optimization were Cummins *et al.* [Cumm17]. Their method automatically abstracts and selects features from the raw source code. This kind of automation drastically decreases the complexity of feature extraction because no software-tool or performance counter is required. Programs are fed through a series of neural network based language models which learn their semantic value. Hence, feature vectors can be yielded and represent the source code without human-involvement. Consequently by capturing the behavior of a program their were able to predict which loop unrolling factor would increase the performance. Similarly Tal-Ben-Nun *et al.* [BenN18] constructed a general purpose processing pipeline geared towards representing code semantics in a robust and learnable manner. They used LLVM-IR to construct contextual flow graphs that in turn are used to train an embedding space. By merging LLVM instructions and dataflow information in contextual flow graphs, the researchers where able to tackle the same problem (loop unrolling factor prediction) while providing better accuracy.

3.4 Feature Cleaning

It is clear that there are many ways to mirror the behavior of a program in feature vectors. Machine learning uses features to establish prediction models. Hence in order for the accuracy of such models to be improved, it is vital to choose the features, that models will be trained on. In compiler optimization research, an initial large feature space is commonly pruned via feature selection [Step05] or projected into a lower dimensional space [Magn14].

3.4.1 Feature Selection

Feature selection requires understanding how each features affects the accuracy of the model. A method to get this information is by applying the correlation coefficient [Wang10, Jian10]. Redundant features are filtered out by removing features that have a strong correlation with an already selected

feature. Another method for correlation estimation is mutual information [Cava06], which evaluates the interdependence of features. For example if knowledge that feature A transfers to the model can be largely acquired from feature B, feature A can be cutted out, with out losing information.

3.4.2 Feature Dimensionality Reduction

While feature selection allows us to select the most important features, the resulted feature set can still be too large to train a good model. Reducing dimension of the feature vector can improve drastically the accuracy of the model. This kind of reduction may be a necessity when prediction is yielded from learning algorithms such as KNN, so the curse of dimensionality [Beye97] is to be avoided. Although reduction will produce a smaller feature vector there can be no information loss when these techniques are performed. Principal component analysis is widely used for dimension reduction in compiler research works. PCA merges features together and constructs new features. In that way little to no original knowledge is lost. The output feature vector will summarize the information of the initial vector. Another useful method to reduce dimensionality of features is via autoencoder. Autoencoders have to two parts: 1) encoder, 2) decoders. Encoders take as input data and outputs a much more compressed structure that encapsulates the original input. After compression is done, decoder tries to reconstruct the original input based on the low -dimension structure generated from the encoder. More specifically in compiler optimization autoencoder have been used to model program source code to obtain a feature vector that will reveal its behavior [Mou16, Whit19].

Chapter 4

Machine Learning Models

Machine learning is the study of algorithms or techniques that are able to learn data characteristics and make predictions on them [Mohr12]. In compilers optimization when machine learning is adopted the general problem is to construct a function that "takes as input a tuple (F, T), where F is the feature vector of the collected characteristics of an application being optimized and T is one of the several possible compiler optimization sequences applied to this application" [Park11b]. The output of such model may be a predicted speed-up value of T or the predicted optimization sequence of T. Two major subclasses of machine learning have been widely used in research: supervised and unsupervised learning.

The essence of supervised learning is that a predictive model will be trained on experimental performance data and features of programs. After it has been trained, the model has learned the correlation between these feature values and optimizations that will bring the best performance. These learned correlations play major role in predicting the best optimization for a new unseen application. Predictive models can be either regression models or classification models. This kind of distinction depends on the form of the output. On the one hand, Regression models are used to predicting continuous variables, whereas on the other hand classification model is used to extracting discrete outputs.

Whereas a supervised learning needs to have labelled data so that models are trained, unsupervised learning is not the same case. When unsupervised learning is used there is no error or reward signal to evaluate the potential solution. Thus, an environment is generated that permits model evaluation; for example dividing program runtime information into clusters, such points within each cluster are similar to each other in terms of program structure [Pere03].

Choosing the right model heavily depends on the feature data of programs and what kind of prediction is needed (speedup, phase-ordering etc). Thus there is no "one model fits all" solution in this problem.

4.1 Supervised Learning

4.1.1 Regression

Regression analysis is a statistical process used for prediction or forecasting. It works by estimating relationships between outcome variables and features. In compilers domain it has been used for various tasks, such as predicting the execution time speed-up [Wen14].

As an example consider we heave a model which takes as input a feature vector X and predicts the execution time Y of the program. Lets say the model has been trained with five known data points (dataset) and each of the five points that compromise the data is called training example. Each training example is a tuple, (x_i, y_i), defined by a feature vector, xi and the program's execution time y_i. By obtaining that knowledge and learning the correlation between the given input and output data, it is understood that the model can be used to make predictions for any new, unseen program's features. The rationale behind the model's ability to make prediction is that function f, which corresponds to the relation between input (x_i) and output (y_i) has been learned. Once the function, f, is in place a prediction, y, can be made for any input feature vector x. This is why regression is essentially curve-fitting.

There are many machine learning algorithms that can be used for regression. These include the simple linear regression model and more complex models like support vector machines (*SVMs*) and artificial neural networks (*ANNs*). Linear regression is effective when a strong linear relationship is established between input (feature vector) and output (labels). On the other hand *SVM* and *ANNs* can both exploit linear and non-linear relationships, but in order for them be accurate they need much more training data.

4.1.2 Classification

The statistical process of categorizing a given set of data into classes is called classification. In this method a new sample is categorized based on other samples whose categorized membership has already been known. A distance function is what will determine the similarity of samples, and thus group them. This method has been widely used in compiler optimizations prediction; for example, classification can be used to predicting which unroll factor should be used for a given loop by associating the input feature vector with "similar" feature vectors.

The k-nearest neighbor (*KNN*) algorithm is one of the most used unsupervised learning algorithms because of its effectiveness and simplicity. It works by picking the k closest instances from the training set to the input instance (program) on the feature space. Euclidean distance is chosen in most cases as the similarity function that will determine the "closeness" of instances, but other metrics can also be used. After having been selected, the k nearest neighbors to the given features (programs) use the optimal parameters of them as the prediction output. *KNNs* have been used in compiler's optimization researches to predict the best optimization parameters [Step05]. Despite its effectiveness, *KNN* is a computational-heavy algorithm, because it must compute the distance between the input and all training data at each prediction. This process can be time-consuming when the train set is large. It is also evident that the algorithm does not obtain knowledge from data; instead it simply chooses the k nearest neighbours. Which means that if noisy data are inserted the model can turn-out to be unstable and thus not robust on ill-suited training-set.

As a substitute decision trees have been adopted to solve similar cases; for example in order to decide the profitability of using GPU acceleration [Coop99, Wang14], determinie the loop unroll factor and to choose the parallel strategy for loop parallelisation [Coop99] . Major advantage of using such a model is that it enables users to understand why particular choices have been made and thus there can be a reasoning for its output. The way decision trees work is by starting from the root of the tree and comparing a feature value of the target class (program) with a threshold to determine which branch of the tree to follow; this process is repeated until lead node reaches the output of the final descision. It should be noted that the threshold values and the structure of the tree are constructed by the machine learning algorithm.

A drawback of using a single decision tree is that the model can easily over-fit due to outliers in the dataset. Therefore random forests have been proposed to diminish the problem of overfitting. The way it works is by using multiple decision trees and aggregating their output predictions. The prediction of each tree relies on the values of a random vector sampled independently on the feature value. Prediction is made by summing up the sub-predictions of random trees to form an overall prediction. Besides classification random forests are also usedi as regression models, such as, in predicting energy consumption of CUDA programs [Reji17].

Another model used for classification is support vector machines (*SVM*). In SVMs the similarity measurement on feature vectors is done by kernel functions. The radial basis function is one of the most used. It works by mirroring the input feature vector to a higher dimensional space where it is easier to separate classes. *SMVs* can model linear and non-linear problems and that is a reason that they have been used in prior works [Jose06].

4.1.3 Deep Neural Networks

Advances in the field of hardware and computer architecture have enabled the use of computational-heavy models, like deep learning in many research fields, for example image segmentation, natural language processing [Miko13], speech recognition [Hint12] and composition of generative models have been of much benefit form the use of *DNN*.

One could argue that *DNNs* work exactly as traditional neural networks do, but that is not the case. By having many much more layers between the input layer and output layer *DNNs* manage to learn representations and features interdependencies in a more accurate way than in shallow networks. In this way if they are provided with enough data and establish their weights they can keep abstractions of the knowledge learned and use them to predict the outcome of a new unseen form of representation. In compiler optimizations domain this can be very helpful. For instance, given a large training set of programs representations and their "good" optimization sequences a *DNN* can learn how features correlate with optimization phases and thus when an unseen program is given to the model, it can predict which sequence fits best for it.

4.2 Unsupervised Learning

In contrast to supervised learning, unsupervised learning works with unlabeled data. When a set of training samples is given unsupervised learning tries to categorize the input sample based on that set. In this way the algorithm learns the distribution of the data. Despite there is no information about the output, by learning the structure of the data the algorithm is able to understand rules and associations between samples by learning the structure of the data, and thus group them. A common choice to reveal the distribution of a dataset is clustering. The way clustering works is by forming groups between closely related set of objects. Object or data points of the same group are strongly associated in terms of features, unlike data points belonging to different groups. *K-means* clustering tries to group nearby data points in the feature space into k clusters. In research in compiler's domain, *K-means* has been used to characterize programs behavior [Sher02], by clustering programs execution into phase groups. It has also been used in optimization selection problem, by clustering the code structure of parallel programs that gain speedup from similar optimizations.

In order for clustering algorithms to perform well, data must be easily grouped in feature space. When one works with large feature vectors noisy data won't allow that kind of separation. Thus some kind of dimensionality reduction should be performed. By doing this, it allows features to be represented by a smaller number of independent variables. Thus, a new feature vector is generated with respect to the most important features (variables) of the old one. In other words the new vector is a combination of the most important features, but it needs less independent variables to be expressed in the feature space [Asho18, Asho19].

4.2.1 Evolutionary Algorithms

Evolutionary search or evolutionary computation is a subclass of machine learning. It is inspired from natural selection and Darwin's theory of evolution [Darw]. The fundamental idea in this process is that given an initial set of possible solutions (population), an optimal solution can be derived by combining the fittest of them. There are three key operations in an evolutionary algorithm: *selection*, *cross-over*, *mutation*. Another necessary component of *EAs* is the fitness function which evaluates the quality each candidate (solution). Evaluation of population is an iterative process. In most cases termination criteria are either a maximum number of iterations (generations) or a fitness threshold of the best solution. For instance in an optimization problem there is a given or randomly generated set of candidate solutions (population). First fitness function measures the quality of each individual solution, that is the probability of each optimization to be selected for*cross-over* is proportional to its fitness value. After members of the population have been stochastically selected, the phase of *cross-over* starts. By mixing candidates of the initial dataset, new offsprings are produced. Driving force of

this action is that by merging "good" candidates, bred solutions will enjoy a greater fitness value than their parents. Additionally in order to keep the size of initial set unchanged, lowest-valued solutions are replaced by the offsprings. However in order to guarantee the algorithms progression and betterment, mutation is utilized. In this way, chances to stick with locally optimal solution are reduced, because no fitter solutions are added to the population. Iteratively this process can generate the best solution for the optimization problem. In order to perform such actions each member of the population is usually represented as an array of bits. In that way cross-over and mutation are easily applied. Fringe benefit of *EAs* is that a large solution space can be explored while the enumerations of all possible solutions is unthinkable. A great advantage over supervised machine learning is that *EAs* require no knowledge to converge to optimal solution apart from current values of the population. However despite the ability to cover an enormous search space, *EAs* need experimental evidence for evaluating solutions from fitness function. This process can be extremely time consuming, though. This drawback can be diminished with the additional help of machine learning models that can estimate potential gains of configurations. In research supervised learning models have been used to predict bright area of a design space, thus leading the evolutionary algorithm on which subarea should focus to [Asho18, Furs08]. Another approach is to prune the initial search space, thus reducing the number of options to explore, instead of predicting its prosperous subareas [Furs11b].

In compiler optimizations domain these techniques can be used in variations of the phase ordering problem. For instance, given a population of compiler flag-sequences of a program the algorithm can find the combination of flags that delivers the best performance. In the first place fitness function evaluates each optimization sequence with respect to predetermined criteria (time, speed-up, energy). Fitness function can be a measurement of the execution time of the application, thus sequences that perform better than others will have greater fitness value. After each sequence has obtained a fitness value, the best of them are combined to form new sequences that yield better performance to the application. Newly formed sequences are byproducts of their parents, meaning that a part of them belongs to each parent. Moreover a part of the initial population gets mutated, which means that certain individual passes of sequences get transformed to different optimization passes. Flag-sequences with the lowest fitness values are replaced by the new ones for the sake of continuous upgrade of the population. This results to the formation of a new generation and the restart of the algorithm. The application gets compiled again for the sake of a new round of fitness evaluation. When termination criteria are met, this process yields the best performing compiler flag sequence for the program.

4.2.2 NeuroEvolution Of Augmenting Topologies

Another evolutionary approach that's been adopted recently is *NEAT*. Similarly to evolutionary search it uses the same three key components: *selection*, *crossover*, *mutation*. *NEAT* studies the evolution of neural network structure in order to have better results. The algorithm starts with an initial set of network topologies. In other words representation of node, edges and weights of the networks. Candidate networks get evaluated. Their predictions are measured from a fitness function, so that a hierarchy is established. After evaluation its over, mutation and crossover phase takes place. Comparably to the evolutionary algorithms the best individuals of the population networks are selected, so as to create new topologies which may yield more accurate predictions. Also mutations change part of current topologies of random networks. For instance weights are redefined and nodes are added. The least fit members of the population dies out, so that the new ones take their place. This process is repeated until a fitness threshold is achieved or a number of generations has been reached. An aspect of the algorithm is to keep the network as small as possible. This method has been used in order to provide efficient neural nets as integrated part of compilers in prior works [Simo13, Kulk12].

4.3 Reinforcement Learning

Another technique that has gained momentum in the last years is reinforcement learning. In that method a model or an agent interacts with the given environment. Through that kind of interaction the agent tries to find optimal ways to fulfill its purpose. In other words an objective is given to the model and through trial and error approaches, it gains knowledge of how to perform in the region. In order to form a behavior or a way to do it, the agent follows a policy. This policy will shape the aim of the agent. Moving towards the goal the model can choose between actions that eventually change its current state. With an eye on its purpose, the agent learns from its selection supported by the reword function. The states that the agent, is been transferred after selecting an action are also evaluated. The value of a state is defined as the accumulated average of future rewords that can be obtained from the current state. With the intention to respect agent's policy, reword function gives positive values when the state that the agent transfers is more "desirable" and thus closer to the goal. In this way, it is able to learn from past actions (exploit) and from new ones (explore).

Because of these interactions with environment and the ability to absorb gain insight, reinforcement learning fits best for scheduling problems. For example mapping in an optimal way instructions are executed from registers in a hardware environment, so that maximum performance is obtained. Modeling that problem states could be represented as register usage loads and number of idle registers, action could be the assignment of a process to a register and the reward driven by the throughput of the overall system. In this way the agent will learn how to allocate optimally resources. Reinforcement learning has also been used in similar problems in computer architecture domain, such as in scheduling RAM memory traffics [Ipek08] and configuring virtual machines [Rao09].

Reinforcement learning has also boost the autotuning of parallel programs (OpenMP) [Eman14]. The major problem on running parallel software is that there in no prior knowledge of the optimal number of threads that should be assigned to run the program. The right number of threads can increase the performance in terms of speed-up and energy. Features of source code that runs on multicore-architectures are extracted in order for the algorithm to gain insight to the state. Moreover, information about the execution time is obtained. Then a reword function is learned offline in order to estimate the reward of a runtime scheduling action. After having acquired this knowledge the agent can predict the number of threads that can potentially increase the speed-up of an OpenMP program. Thus in the next scheduling epoch, it uses empirical observations of applications speedup to update the reword function and test the knowledge obtained so far.

Commonly reinforcement learning techniques are preferred for modeling problems that have an evolving nature. But their accuracy heavily depends on forming an appropriate value function to estimate the immediate reward. Forming a function that leads to the greatest cumulative reword when the agent is transferred from the current state to the most promising one is not a trivial task. Policy is closely related to the formation of value function, because all parameters that describe the aim of the agent must be met. Modeling such a function in complex and dynamic environment requires a sufficient amount of knowledge that has been obtained so far.

When modeling compiler optimization problems where predictions depend on various static or dynamic features and possibly the input of the program, exploration space can be gigantic. Thus the exploitation phase of the algorithm will only manage describe a rather small area of the design space of the problem. In order for RL to be used effectively, great insight of the environment and all possible actions are needed. For that reason deep learning techniques have been used in conjunct with RLs so that a value function to be learned [Mous18].

4.4 Prior Research Approaches

Despite the use of machine learning in programming languages research domain is not widely adopted, important steps have been made towards autotuning of compilers.

The effort of Cooper *et al.* [Coop99], showed that fixed lengthened optimization sequences are not

the optimal way to reduce code-size of applications. When source code needs to be run on embedded systems, it is burned in ROM of the device. In an effort to have small code-sizes, software engineers will tolerate much longer compile times in an attempt to find the best optimization sequences. The realization of that need is what drove Cooper *et al.* to the use of genetic algorithms for that problem. Standard optimization sequences potentially will reduce code-size of a program (-O2), but not in an optimal way. When a transformation is applied, it can create opportunities for other transformations. Likewise a transformation can eliminate opportunities of other optimizations. That interdependency of passes heavily depends on the structure and the semantics of the program. By giving an array representation to an initial set of different optimization sequences the three basic components of the *GAs* (selection, cross-over, mutation) are utilized. The code size after compilation of the application with a particular candidate solution (transformation sequence) has been compiled , serves as fitness function. After each epoch the population gets renewed, by merging parts of the most promising sequences and mutating others. In this way an application-specific optimization sequence is obtained after iterative compilation. This method has an advantage of 14.5% compared to fixed sized sequences.

Despite the fact that iterative compilation can be extremely gainful, in order to yield performance improvements, large optimization space should be explored. Evolutionary algorithms find efficient optimization with the trade-off time. That is because in order to exploit good transformation many epochs of population updates are needed. When the number of optimization passes that needs to be considered grows, the exploration space grows exponentially. Indenting to reduce the searches needed, Agakov *et al.* [Agak06] introduced a machine learning model that speed ups iterative optimization, by focusing the search to the most promising areas of optimization space. First, a Markov chain model is learned off-line by using training set of benchmarks. In order to gain insight to the applications static feature are used, like the number of loop and the number of instruction per block. After having acquired knowledge of the application features, the model is trained in order to predict which optimization will give the best performance. Thus, when an unseen application is encountered, its feature vector representation is extracted. After the vector has been obtained, *PCA* takes place in order to reduce its dimensions. Then nearest neighbour algorithm uses the reduced feature vector to find the closest benchmark representation that will reveal the most promising optimization which search needs to focus on. This focused search method achieves up to 86% of performance improvement compared to 36% performance improvement that is obtained by non-focused search, under the same number of evaluations. In that spirit, Knijneunberg *et al.* [Kisu00] formed a method of iterative compilation to decide optimal loop-till size parameters and loop-unroll factors. In order for the optimization space to be explored, evolutionary search was used. By focusing on just two flags (loop-unroll, loop-tiling). and their possible different parameters, Knijneunberg *et al.* where able to reduce the search space in order to gain significant speed-ups on a small number of iterations (50 iter.). Similarly Kulkarini *et al.* [Kulk04] modeled the phase ordering problem in a way to enable evolutionary search. In this work they uses an optimization set of fifteen different flags. A boosted search technique in iterative compilation was introduced by Bodini *et al.* [Bodi00]. By using static features of programs and Markov chains to focus on iterative optimization they were able to achieve up to 40% speedup. They used fourteen transformations in sequences with length of five, which consist of a relatively large exploration space (14^5). After the model has learned how to exploit that space, it is tested in a much larger space (80^{20}), in order to predict which static features of unseen programs with optimization sequences (with length of twenty) would be more beneficial. Another approach to the phase ordering problem was given by Kulkarni *et al.* [Kulk12] who used neuro evolution of augmenting topologies. Instead of predicting the complete sequence of optimizations from the initial static features extracted from code, they integrated an artificial neural network in Jikes RVM Java JIT compiler to build good optimization sequences. For that purpose, after prediction has been obtained from the neural network the optimization is applied to the source code and static features are extracted against the optimized version of the program. After that neural network is fed with the new features to predict again a possibly efficient transformation. The network is constructed with the use of *NEAT* in order to avoid modeling implications.

An additional approach to the phase ordering problem was introduced by Purini *et al.* [Puri13] .In

order to reduce the ineffectively large optimization sequences space within LLVM -O2 they defined a machine learning based downsampling technique. The authors originated a clustering algorithm to cluster sequences bound by the similarity matrix that can be used to calculate the Euclidean distance between two sequence vectors.

Another variation of phase ordering can be found in the work of Park *et al.* [Park12]. The authors proposed a new static characterization, that is based on control flow of programs. By getting a graph based representation of the programs, authors were able to capture a more expressive representation, without executing the source code. That graph structure and a series of optimization sequences are fed to an SVM model, that has learned offline to predict the speed-up gained from transformation sequences.

Correspondingly Cavazos *et al.* [Cava07] used dynamic features to predict good transformation sequences. Their method used performance counters to construct an applications feature vector. By obtaining features such as cache statistics (e.g., misses, stores) and branch instruction statistics (e.g., misspredicted branches), authors where able get a more comprehensive representation of the program. In order to make prediction of optimization sequences they trained a regression model. In their method various programs are run with different optimization sequences and their speedups are examined. Then selected effective optimizations and features of the applications are fed to the regression model. Thus when features of an unseen program are inserted to the model, it outputs the most promising sequences. In this way authors modeled the selection problem. Their technique outperforms static analysis techniques, because of system-level-characterization of programs.

In addition Ashouri *et al.* [Asho17] by using dynamic features created a framework for speedup prediction of optimization sequences. The authors clustered all the LLVM's -O3 optimization passes into optimization subsequences. They also trained a model with features of programs and their speedups for every possible optimization subsequence. Feature space was reduced with PCA in order to minimize training cost. After the regression model has been trained, an iterative process takes place. Features of a new program and an optimization subsequence are fed to the model in order to yield the speed up prediction for the given subsequence. By giving to the model every possible subsequence of the cluster the best are examined, in a dynamic manner, so that the new version of the program can be optimized again with one of all possible sub-sequences. Hence, the overall optimization sequence is built in an iterative way instead of being predicted.

An architectural-related approach was introduced by Magini *et al.* [Magn14] in order to solve the parameter selection problem. The authors proposed a machine learning model to predict the coarsening factors that achieves the greatest speed-up for different GPU architectures. Thread coarsening factor (number of threads to merge together) selection is a trade-off between execution of redundant instructions and exploitation of thread-level parallelism. Its heavy architecture-dependence adds to the programmers an overhead of manually finding the best coarsening factor. In that spirit Magiini *et al.* [Magn14] created an iterative optimization process. First, static feature from OpenCL kernels (e.g. number of instructions, number of loads etc) is extracted from a number of benchmarks. Then a neural network is fed with the features and the speed-ups gained for various different coarsening factors applied on the OpenCl kernel. Furthermore, another information that the networks needs to be supplied with is the architecture that the kernel runs on (e.g., Nvidia Kepler, AMD Tahiti). The number of static features is also reduced by PCA, without significant information loss. By training the model with these features, it is able to predict if thread coarsening should be applied to further optimize the kernels performance. In other words the model is fed with the static features of a new OpenCl kernel and outputs the coarsening factor. In a positive case coarsening is applied and static features are extracted again to feed the model with the new state of the kernel. This cascade model achieves speedups between x1.11 and x1.33 on average.

The same problem was further investigated by Cummins *et al.* [Cumm17]. They employed neural networks to extract features from source code for compiler optimization. Their system automatically abstracts and selects appropriate features from OpenCl kernels. In their work, programs are fed through a series of neural network based on language models. In this way, language models learn how to rep-

resent the kernels as a sequence of floating point vectors, acquiring knowledge of how code correlates with coarsening factor selection Their technique achieves on average 16% speedup compared to the technique by Magini *et al.* on the coarse threading problem, across four different architectures.

In that spirit Cummins *et al.* [Cumm20] introduced another model to obtain representations that accurately captures the semantics of programs. Their work is based on graph-based program representation using low level language agnostic format (LLVM-IR). In this way this representation composes "a directed attribute multi graph that captures control, data and call relations and summarizes instruction and operand types ordering" [Cumm20]. The inputs of a neural network are fed with that representation, so that whole-program classification tasks are enabled.

Despite machine learning offers major benefits for constructing optimization heuristics, many research problems remain open (phase-ordering). The gap between what state of art methods achieve and the performance of an optimal heuristic needs to be additionally reduced, due to the increasing complexity of computing systems, requires ever more accurate and aggressive optimizations.

Chapter 5

Static Analysis on Phase Ordering Problem

Compilation time can be a heavy burden in programmers' everyday life. In most cases programmers use standard optimization sequences like (e.g., -O3, -O2), rather than finding an optimal sequence matching with their application needs. Past research approaches have shown the point that "built in" optimization sequences will not gain the optimal speedup for the application being compiled. It is also noteworthy that exploring different combination of compiler flags is a tedious and time-consuming task. Thus, it is not unusual for software engineers to cling on the adequacy of common transformation flags. In that spirit we focus our work on static analysis of source code in order to predict the most rewarding optimization sequences. First, we run benchmarks in different combinations of flags and measure their runtime speedup with respect to -O3. Second, we train the network with benchmarks static features and speedups from the various flag sequences. Third, the model is trained to be able to correlate program's features and optimization flags to yield a speedup prediction. Thus, features extracted from unseen source code with a list of all possible flag combinations that have fed the trained models inputs will result in a list of predictions of speedups with respect to -O3. In that way information about what might be a "good" optimization sequence is easily obtained, and the program is compiled with the flag combination that generated the best speedup prediction. In the following sections we discuss the methods we use in order to compose such a system.

5.1 Compiler Infrastructure

In order to be able to perform various optimization sequences instead of using default transformations, we used LLVM and its toolchain. LLVM is a compiler infrastructure project, which is used to construct, optimize and produce intermediate and binary machine code. It can also be used as a compiler framework where user can provide the front-end and the back-end. In that way a high level language is translated to LLVM-IR and then to low level assembly. LLVM-IR is the intermediate level representation provided from LLVM, and can be used for optimization transformations as discussed in 2.2. Optimized IR is then passed to the back-end where machine code is generated.

5.1.1 Reducement of Exploration Space

From past research works [Kulk12], it can be understood that searching a good optimization sequence among all possible sequences would be like looking for a needle in a haystack. The exploration space of optimizations sequence grows exponentially given the ever growing number of compiler transformations. As an example GCC [GCC] has more than 200 compiler passes and and LLVM's Clang and LLVM's opt have more than 100 transformations. Searching for sequences of transformations of variable length among all passes would result in a search space of over a billion. That's why we selected to test fixed-length sequences, with the length of five.

5.1.2 LLVM Passes Selection

In order to keep the list of possible optimization sufficient enough for training, we used a small enough prediction model to avoid the much time-consuming process of compilation and execution of

benchmarks we selected five passes that transform LLVM-IR:

- **mem2reg:** Rewrites the IR by raising loads and stores to stack-allocated values to registers.
- **dce:** Removes code that is unreachable or does not affect the program result.
- **loop-unroll:** Transforms loop conditions and possibly parts of code inside the loop, so that fewer iterations be needed in order to obtain the same results.
- **instcombine:** Combines instructions in order to form fewer simple instructions with equal semantic value for the program.
- **constprop:** Merges and propagates constants, by replacing instructions involving only constant operands (e.g., add, sub) with a constant value.

We have observed that the complexity of the variation of phase ordering problem is proportional to the number of individual optimizations and the length of sequences. By defining as n the number of different passes and r the length of optimization sequences, the number of possible permutations (with repetitions) is given by the equation:

$$P_{(n,r)} = n^r$$

The reason behind the repetitions allowed in permutations is that a pass might be able to re-optimize the IR if another pass has been enabled before it. Hence the number of all possible optimizations sequences that can be formed in order to obtain a speed-up improvement is equal to 3125 (5^5).

5.2 Benchmarks

In order to measure the speedups gained against -O0 flag Collective Benchmark (cBench) is used [CToo]. cBench is a collection of open source programs with various datasets. It was introduced by Frusin in order to help research on programs and architecture optimization also contains 32 programs too. Another benefit of cBench is that is highly portable, all benchmarks imported include scripts to ease the process of iterative compilation and to perform automatic optimizations by using GCC, LLVM, PathScale and other compilers. Thus we use those scripts in order to compile the programs with all optimization sequences and measure their run-time execution with linux-perf [Perf]. Moreover, it is important to mention that we only use default datasets to run the programs and thus we don't feed the predictive model with that information (size of dataset). Some of the most useful scripts that come along with the Collective Benchmark are the following:

- **__compile:** Compiles benchmarks with a specific compiler (e.g., GCC, LLVM) indicated by the first parameter with the flags specified in the second parameter.
- **__run:** Executes the benchmark with the fist parameter indicating the input dataset that benchmark needs to run and the second parameter is the upper bound of the loop wrapper around the main procedure. We set the loop-wrapper to five in order to have more accurate time measurements.
- **all__create_work_dirs:** Creates temporal work directories for each benchmark.
- **all_compile:** Compile all benchmarks in the temporal work directories.
- **all_run:** Runs all benchmarks in temporal work directories in temporal work directories.

Benchmarks are compiled with every optimization sequence and then are executed so that information about their execution-time is obtained. When a program runs, its execution time is saved in a file that will be used to create the training dataset for the prediction model. Linux-perf is the tool we use in order to measure the execution-time. In order for accurate metrics to be, repetitions flag in linux perf is enabled and set to ten. In that way the program is executed ten times and the average execution time is what is saved in file. It is also important to note that we have cleaned the caches before we execute different benchmarks, in an effort to clear possible interferences and isolate applications data traces. Additionally we measure the execution time of all benchmarks with -O2 and -O3 flags with the intention to define speedups for every sequence against the different standard flags.

5.3 Static Analysis

In order to obtain static features from benchmarks, to feed with data the machine learning model, we use Milepost-GCC. Created by Fursin *et al.* [Furs11a], Milepost-GCC was the first attempt to make a practical on fly machine learning based compiler combined with an infrastructure targeted to autotuning. Despite the tool ceased to be used for compilation, it has been used in research as feature extractor. Its maintenance and development was discontinued in 2010 but a docker image of the tool is available [Lulo]. By using Milepost we extract static features from all benchmarks in an array form and save them in a file.

5.4 Dataset Creation

After we have recorded the execution time of the benchmarks with all possible optimizations sequences and collected their static features, we construct the training dataset. The file of time measurements contains the benchmark name for every execution, optimization sequence used, the execution time and speedups with reference to O2 and O3. Defining the execution time of a benchmark compiled with -O3 flag as T_{O3} and the execution time compiled with the optimized sequence as $T_{opt.seq}$, the equation of speedup is given below:

$$SP = \frac{T_{O0}}{T_{opt,seq}}$$

The file with the static features collected from Milepost-GCC contains in every row the benchmark name and the fiftyfive features that Milepost outputs. In that way we merge the information of the two text files into a csv file in order to train a prediction model. Hence the csv file contains in every row the name of a benchmark, the flags used to compile it the execution time, the relative speedups (w.r.t. -O3, -O2) and the 55 feature values obtained from Milepost in every row . Another essential aspect is that all values need to be in a numerical form before they are fed into the machine learning model. For that reason binary encoding is used so that these categorical values are transformed. All combinations of flags are enumerated. Hence every single optimization sequence can be represented as an array of binary digits, whose legth depends on the number of unique categories of the examined feature. In our experiment we use 3125 (5^5) different optimization sequences. An array of length 12 ($\log_2 3125$) is needed so there can be a binary representation. Thus 12 additional columns are required so that the optimization sequences are modelled and fed into prediction model.

5.5 Regression

In order to make speedup predictions based on the feature values and the combination of flags, we train a linear regression model. In that way the model learns the correlation between the given data features (static features of benchmarks and flag sequences) and their output (speedups). For that purpose all static features collected form Milpost-GCC get scaled. The rescaling method we use is

min-max normalization. The rescaled static features combined with the array of binary digits that represents a compiler optimization sequence are fed into the model, with their speedup values. In that way a function f that models the relationships among all the input features (x) and speedups (y) is obtained, which means that by providing the static features of an unseen program and an optimization sequence that we want to apply to the program, a prediction about the speedup can be gained:

$$y = f(x)$$

After the model is trained we integrate it into the autotuning process. During that process a C program is compiled with the best optimization sequence predicted from the model. When this methodology is used features collected from an unseen C program get scaled, which enables the scaler to learn from the training dataset. These features merged together with every optimization sequence are fed into the trained model. Thus a list of speedups of compilers transformations is obtained. The transformation sequence that is predicted to have the best performance is the one used to compile the code via Clang compiler.

5.6 Results

In order to test our system's predictions we integrated various regression models. The way a predictive model gains knowledge from feature data can be of great significance. In other words, a model can learn in a dissimilar way from another regression model and thus the information obtained from the input given be different. In our system an unseen C program get compiled with the best predicted optimization sequence. So, it is critical to choose a prediction model which is accurate and its estimated speedup is close to the real speedup. Hence we test our system with 6 different regression models: Linear, Lasso, Ridge, SVR, Decision Trees, Random Forest. It is important to note that our regression models were trained with the exact same train set. This training set is composed by each program of the Collective Benchmark run with every optimization sequence except the ones that show the best improvement for each program. In other words the five most efficient optimization sequences for each program are left out from the training set and used as a test set for our models. Below we show the accuracy of its model with respect to mean absolute error and mean squared error.

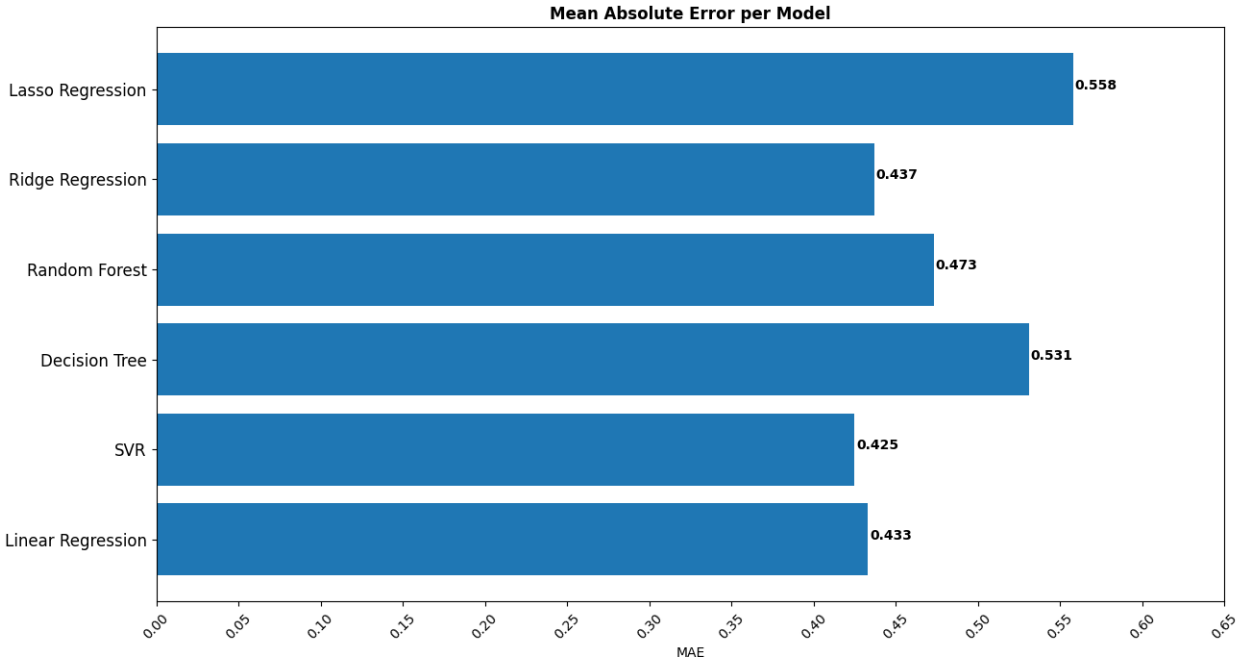


Figure 5.1: Mean Absolute Error per Model

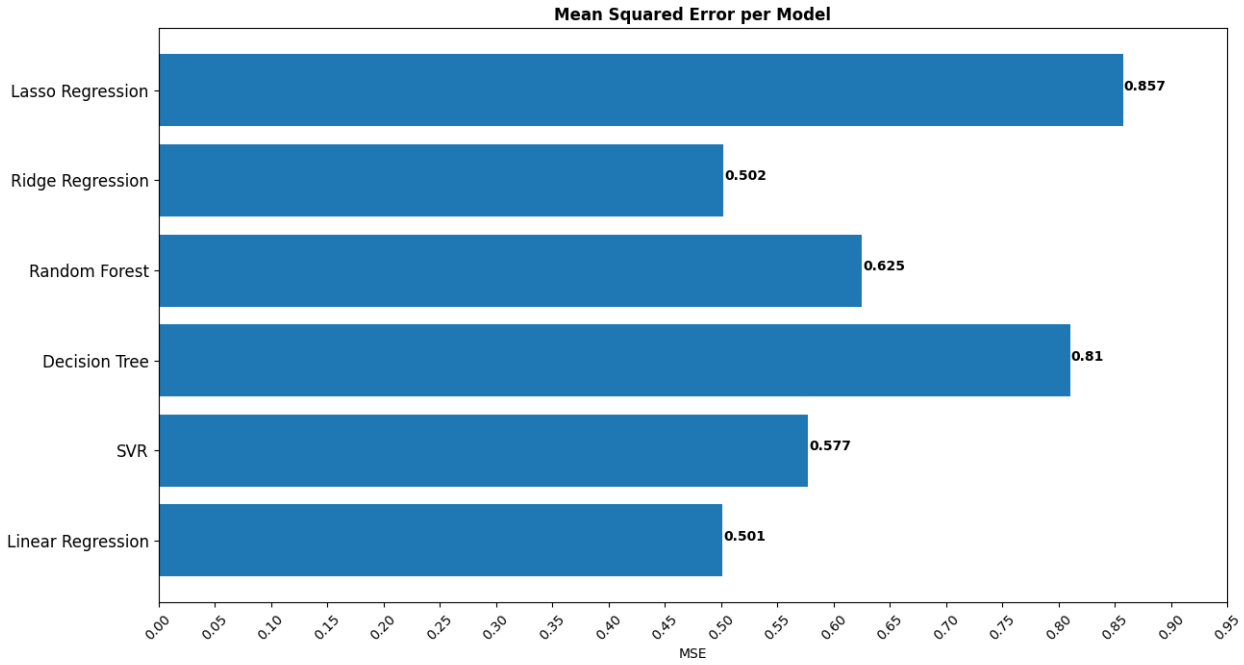
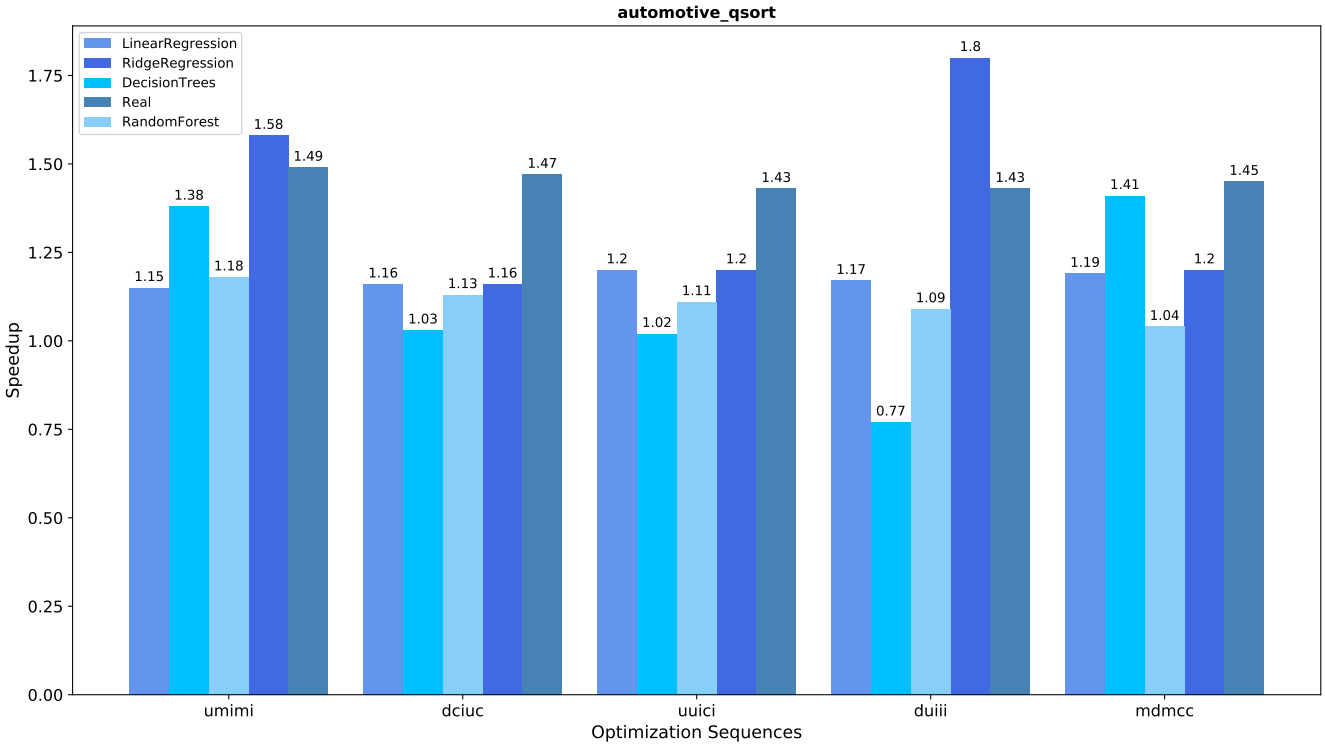


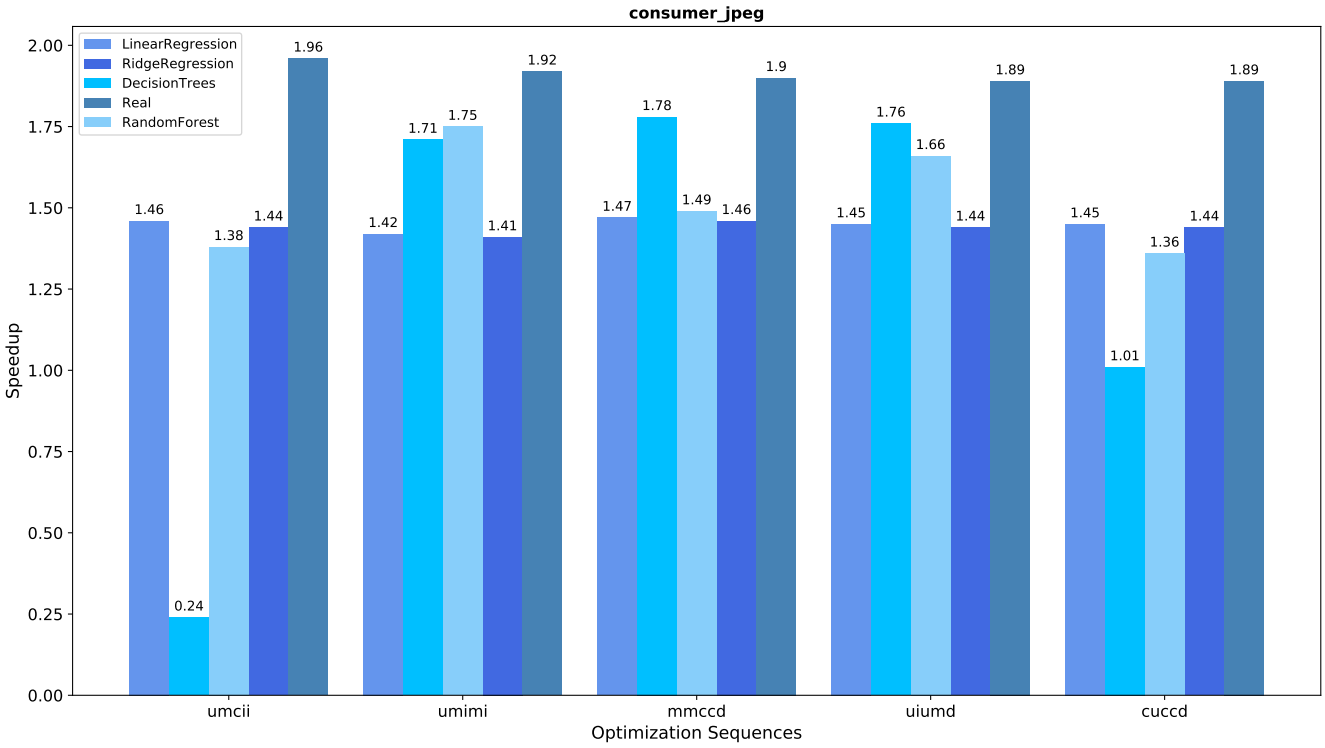
Figure 5.2: Mean Absolute Error per Model

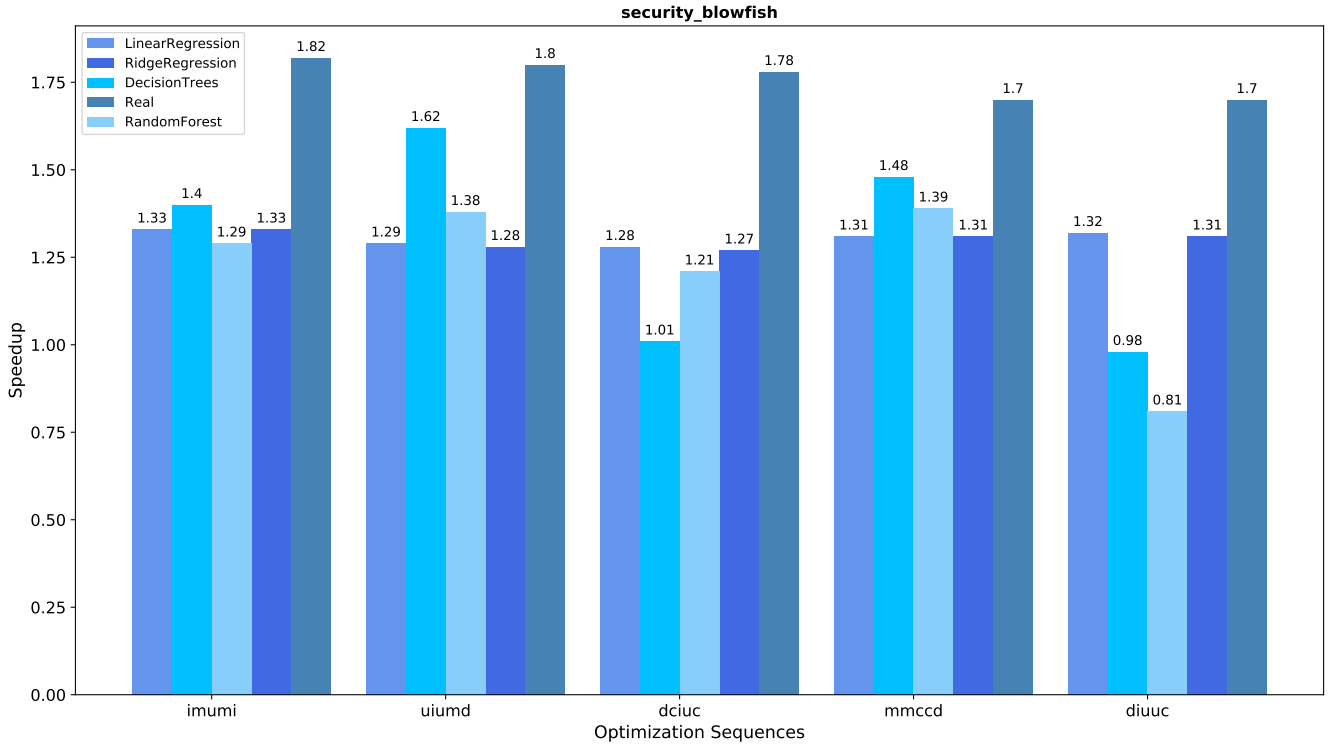
In terms of mean absolute error SVR is the most accurate model. Despite that Ridge and Linear regression seem to also have accurate predictions as well. These results show that the predicted speedups do not diverge from the real speedup obtained from the optimization sequence. When taking in consideration the mean squared error of the models, Linear regression seems to be more robust, because mean squared error penalizes large errors. Despite that predictions are relatively close in every case, meaning that models learn the correlation between training features (code static features and flags) and speedups. This reveals that regression models can provide accurate estimations, thus exercising a great impact on iterative compilation. By predicting the speedups gained from flags we are able to evaluate only the most promising optimization sequences. Considering that, by giving a list of all possible optimization sequences to the regression models we are able to inspect the ones that show the greatest potential. In order to be accurate as prediction model we use Linear Regression because of its robustness in terms of mean squared error and mean absolute error. Notably Ridge Regression also delivers similar results. Nevertheless, Decision Trees and Random Forest do not enjoy the same accuracy. In the figures below we illustrate the predictions of those models on various programs and optimization sequences from the test set.

Optimization	Name
dead code elimination	d
memory to register	m
constant propagation	c
instruction combine	i
loop unroll	u

Table 5.1: Flag's naming

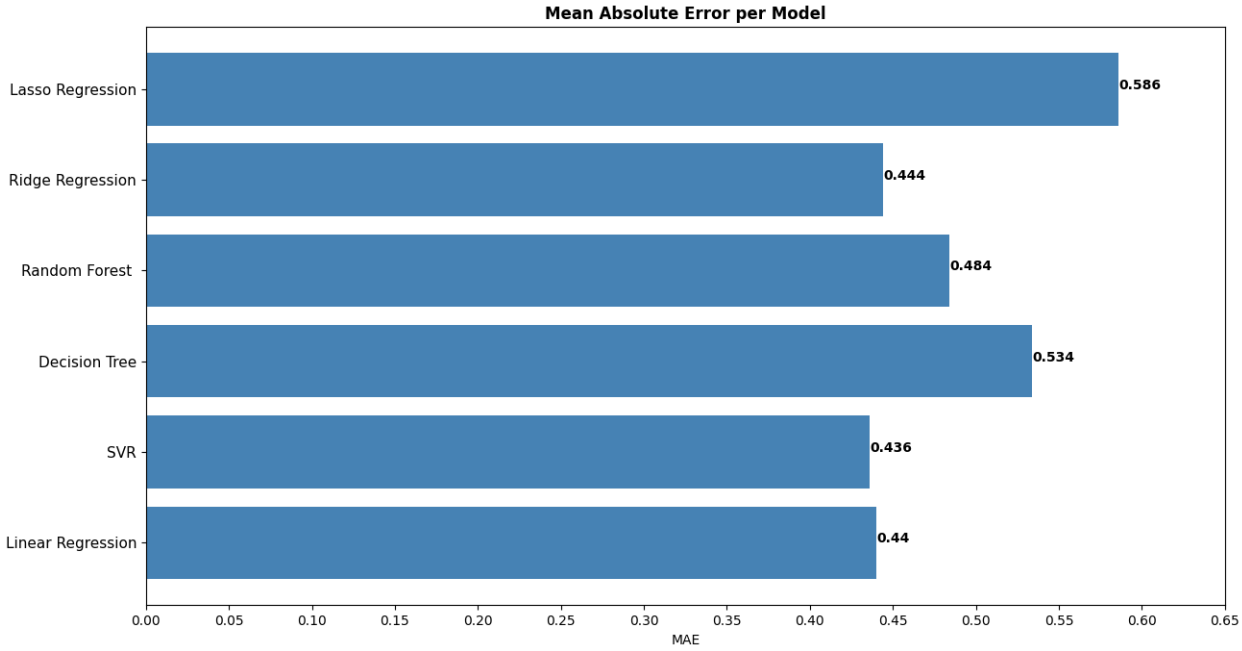

automotive_qsort
LinearRegression
RidgeRegression
DecisionTrees
Real
RandomForest
Speedup
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
1.15
1.38
1.18
1.58
1.49
1.16
1.03
1.13
1.16
1.47
1.2
1.02
1.11
1.2
1.43
1.17
0.77
1.09
1.8
1.43
1.19
1.41
1.04
1.2
1.45
umimi
dciuc
uuici
duiii
mdmcc
Optimization Sequences

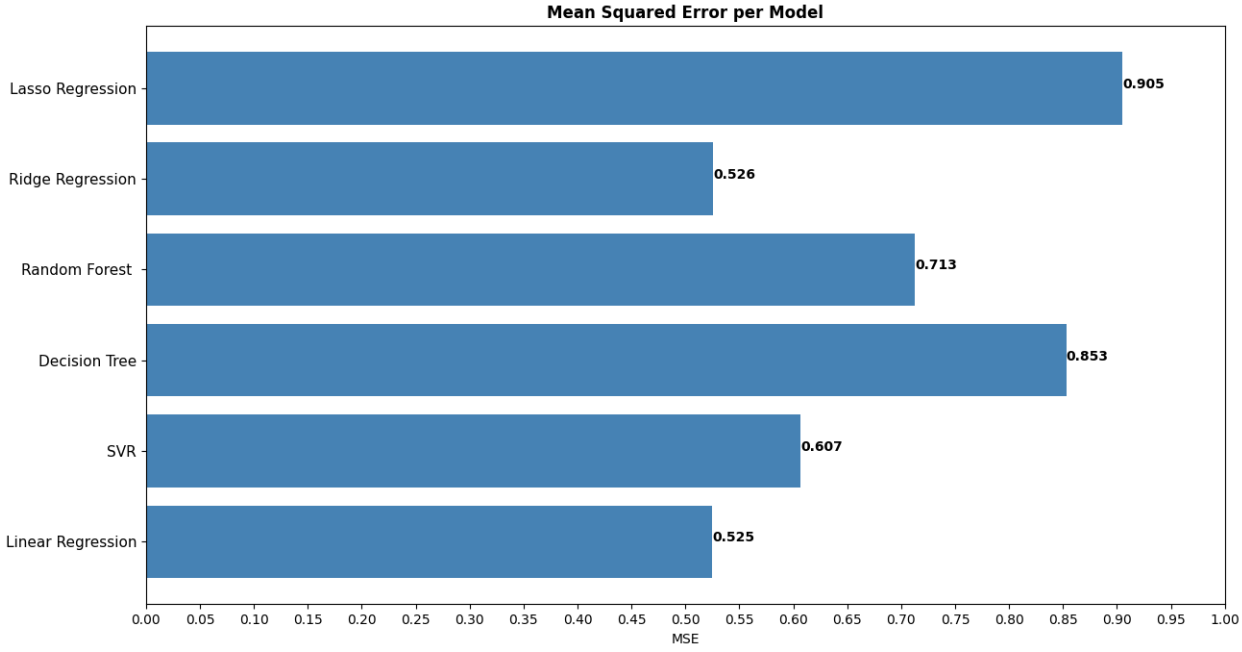

consumer_jpeg
LinearRegression
RidgeRegression
DecisionTrees
Real
RandomForest
Speedup
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
1.46
0.24
1.38
1.44
1.96
1.42
1.71
1.75
1.41
1.92
1.47
1.78
1.49
1.46
1.9
1.45
1.76
1.66
1.44
1.89
1.45
1.01
1.36
1.44
1.89
umcii
umimi
mmccd
uiumd
cuccd
Optimization Sequences



It is worth mentioning that for each program only the five best optimization sequences are taken in consideration for evaluation. From the above figures it seems that Random Forest and Decision Tree have better predictions than Ridge Regression and Linear Regression in some cases. But also they tend to have very high value errors in other cases, which is the reason they perform so poorly in terms of mean absolute error and mean squared error 5.1. Also for each program Ridge and Linear regression seem to be stuck in some cases and in that way not follow the gains and losses of each optimization sequence. That fact implies that more data is needed in order for the models to form a function f that follows these changes of those models.

In a second experiment we tried to test the above models with a different setup. As a test set we used a small part of the whole dataset. In that way we do not hide the best flags of the programs. In this manner regression models have already seen the features of a program and the impact of a combination of flags. The metrics observed, when that technique is used, are shown below:





It is observed that there in not huge difference in terms of mean squared error and mean absolute error, meaning that regardless the method used to train the models, the knowledge gained to predict the speedup does not differ much. Despite that again linear regression seems to be the model that performs best.

Chapter 6

Conclusion

6.1 Concluding Remarks

Speedup prediction has been an active scientific field in the past couple of years [Cumm17]. Despite that no static analysis techniques have been evaluated, most feature extraction approaches use dynamic characterization which requires the execution of the program. This approach can be very time consuming and tedious and that why a static analysis technique comes in handy. In the present thesis we have implemented a static approach for the phase ordering problem. First, in order to train a regression model we have formed a dataset which consists of static features of C programs collected with Milepost-GCC. Another part of the dataset is the speedups of each program with a combination of flags. In order to keep the optimization's search space feasible we have considered only five Clang optimizations (dce, constantprop, mem2reg, loop-unroll, instcombine). All permutations of length five of those optimizations compose our search space. We have trained a linear regression model in order to be able to make predictions on the speedup of an unseen C program only by obtaining its static features and feeding them into the regression model with all possible optimization sequences. In that way linear regression yields a list of predicted speedups, one for every sequence. The optimization sequence that corresponds to the highest predicted speedup is the one that the input unseen program gets compiled with. As shown in Figures 5.1 regression models can relate static features with the predicted speedup. In this way, by training a regression model we can have an estimation to which combination of passes a better speedup could be led. Thus, we can focus our search and iterative compilation process only in the most promising ones. That's because higher predicted speedups of optimization sequences have higher real speedups. This is quite useful when an exponential search space needs to be examined. This kind of exploration can become more effective when it is known where the optimal solution lies. And this is what our system predicts. The shortage in accuracy of our system is due to the lack of diversity in our dataset. By having more than 20 programs being compiled and run with 3125 different optimization flags the static features do not have much variation. That's what the second experiment shows us 5.1. But surprisingly there still seems to be a correlation between static features and flags that regression models are able to exploit. The complexity of acquiring knowledge of that interdependencies grow exponentially with respect to the length of the examined sequences. So, it is clear that in order to predict more accurately and to have larger search space more training samples are required. Regardless, we have shown that even with a small dataset prediction of the speedup of a combination of flags can be considered as valid technique so we can have an insight in which optimization sequences there can be the most impact on performance. This can be extremely useful when the compilation of a program is heavy, and in order to find the best optimization sequence iterative compilation is needed. In our method only by obtaining the features of the code, we can estimate the most gainful optimization sequences. Because of this feature, the draining and laborious iterative compilation process can turn into a quick check of the most promising optimization sequences.

6.2 Future Work

In this thesis, we have focused our interest in gaining insight into how the interchangeability of optimization flags can affect a program and if that kind of relation and interdependency can be modeled so to be able to predict the speedup of an optimization sequence given the static features of the program. However autotuning in compilers is research field that constantly advances with the use of and the advances in natural language processing models [Cumm17]. Yet, many issues remain open and thus more challenges arise:

- From our results it is clear that in order to have a more accurate prediction model more data is needed. In other words more diverse features and more programs are required in order to compose a dataset. By having a dataset large enough regression models will be able to converge more precisely. Thus, the higher estimated speedups will be the higher actual speedups. The construction of such dataset must have diversity in static features so that regression model is able to generalize. Another factor that must be taken in consideration is the time that is needed to compile programs, which must be kept short enough so that iterative optimizations won't take too long. Moreover, in order to make easier and more portable the collection of data from the various flags selected as search space, scripts which provide that workflow would be developed. In such a way, data collection will become an automated process.

- Despite the fact that static features are easily collected, they do not reflect how the program functions. By using dynamic features this kind of utility can be observed and thus a comprehensive picture can be obtained. The reason for their use is because they provide us with more accurate perception of the resources used from the code ran. It is important to note that by having a comprehensive characterization of the code estimations the actual speedup can be more precise and closer. In addition, by merging static features with dynamic features of the code, we can get the best of both worlds. In this way feature vector of a program holds information about both structure and utility of the code. Similarly fashion natural language processing models could output feature vectors from code. Their usage can point out the semantic value of programs. By establishing factual and precise representations, less noise is inserted to machine learning models.

- By examining the phase ordering and the optimal selection it is understood that there is an evolutionary nature in these problems. This kind of environment fits perfectly to genetic algorithms. By building the sequence step by step instead of predicting it, interdependencies of flags can be examined and thus a set of best combination can be kept and reevaluated in the next generation Neuroevolution of augmenting topologies). An important aspect of genetic algorithms is the initial generation. Starting with an inauspicious optimization sequence generated by the algorithm might get stuck in a local minima. Despite that the iterative process of genetic algorithms, compiling and running with the best selected sequences can concluded in the best optimization sequence for the program.

- Finally, when taking in consideration compiler optimizations we must think of the architecture the code will be run on. That is because the performance of optimizations may have different effect depending on the architecture. This is quite common when parallelism of optimizations are taken in consideration. The integration of a system that can predict in which architecture the code should be deployed could have enormous effect on terms of energy consumed or speedup.

Bibliography

[Agak06] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint and C. K. I. Williams, "Using machine learning to focus iterative optimization", in *International Symposium on Code Generation and Optimization (CGO'06)*, pp. 11 pp.–305, 2006.

[Alla15] Miltiadis Allamanis, Earl T. Barr, Christian Bird and Charles Sutton, "Suggesting Accurate Method and Class Names", in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, p. 38–49, New York, NY, USA, 2015, Association for Computing Machinery.

[Alla18] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu and Charles Sutton, "A Survey of Machine Learning for Big Code and Naturalness", *ACM Comput. Surv.*, vol. 51, no. 4, July 2018.

[Alle70] Frances E. Allen, "Control Flow Analysis", in *Proceedings of a Symposium on Compiler Optimization*, p. 1–19, New York, NY, USA, 1970, Association for Computing Machinery.

[Alon18] Uri Alon, Meital Zilberstein, Omer Levy and Eran Yahav, "code2vec: Learning Distributed Representations of Code", 2018.

[Asho14] A. H. Ashouri, G. Mariani, G. Palermo and C. Silvano, "A Bayesian network approach for compiler auto-tuning for embedded processors", in *2014 IEEE 12th Symposium on Embedded Systems for Real-time Multimedia (ESTIMedia)*, pp. 90–97, 2014.

[Asho16a] Amir Hossein Ashouri, Andrea Bignoli, Gianluca Palermo and Cristina Silvano, "Predictive Modeling Methodology for Compiler Phase-Ordering", in *Proceedings of the 7th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and the 5th Workshop on Design Tools and Architectures For Multicore Embedded Computing Platforms*, PARMA-DITAM '16, p. 7–12, New York, NY, USA, 2016, Association for Computing Machinery.

[Asho16b] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos and Cristina Silvano, "COBAYN: Compiler Autotuning Framework Using Bayesian Networks", *ACM Trans. Archit. Code Optim.*, vol. 13, no. 2, June 2016.

[Asho17] Amir H. Ashouri, Andrea Bignoli, Gianluca Palermo, Cristina Silvano, Sameer Kulkarni and John Cavazos, "MiCOMP: Mitigating the Compiler Phase-Ordering Problem Using Optimization Sub-Sequences and Machine Learning", *ACM Transactions on Architecture and Code Optimization*, vol. 14, pp. 1–28, 09 2017.

[Asho18] Amir H. Ashouri, Gianluca Palermo, John Cavazos and Cristina Silvano, "The Phase-Ordering Problem: A Complete Sequence Prediction Approach", 01 2018.

[Asho19] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo and Cristina Silvano, "A Survey on Compiler Autotuning using Machine Learning", *ACM Computing Surveys*, vol. 51, no. 5, p. 1–42, Jan 2019.

[Asse11] "MINimal IR space", http://www.assembla.com/wiki/show/minir-dev, 2011.

[BenN18] Tal Ben-Nun, Alice Shoshana Jakobovits and Torsten Hoefler, "Neural Code Comprehension: A Learnable Representation of Code Semantics", in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, p. 3589–3601, Red Hook, NY, USA, 2018, Curran Associates Inc.

[Beye97] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan and Uri Shaft, "When Is "Nearest Neighbor" Meaningful?", *ICDT 1999. LNCS*, vol. 1540, 12 1997.

[Bodi00] François Bodin, Toru Kisuki, Peter Knijnenburg, Mike Boyle and Erven Rohou, "Iterative compilation in a non-linear optimisation space", *Workshop on Profile and Feedback-Directed Compilation*, 03 2000.

[Cast11] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra and J. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications", in *2011 18th International Conference on High Performance Computing*, pp. 1–10, 2011.

[Cava06] J. Cavazos, C. Dubach, F. Agakov, Edwin V. Bonilla, M. O'Boyle, G. Fursin and O. Temam, "Automatic performance model construction for the fast software exploration of new hardware designs", in *CASES '06*, 2006.

[Cava07] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael O'Boyle and Olivier Temam, "Rapidly Selecting Good Compiler Optimizations using Performance Counters", 03 2007.

[Coop99] Keith D. Cooper, Philip J. Schielke and Devika Subramanian, "Optimizing for Reduced Code Space Using Genetic Algorithms", *SIGPLAN Not.*, vol. 34, no. 7, p. 1–9, May 1999.

[CToo] "Cbench", https://ctuning.org/wiki/index.php/CTools:CBench.

[Cumm17] C. Cummins, P. Petoumenos, Z. Wang and H. Leather, "End-to-End Deep Learning of Optimization Heuristics", in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 219–232, 2017.

[Cumm20] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler and H. Leather, "ProGraML: Graph-based Deep Learning for Program Optimization and Analysis", *ArXiv*, vol. abs/2003.10536, 2020.

[Darw] Charles Robert Darwin, "On the Origin of Species".

[Eman14] Murali Emani and Michael O'Boyle, "Change Detection Based Parallelism Mapping: Exploiting Offline Models and Online Adaptation", 09 2014.

[Esma11] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling", in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376, 2011.

[Ferr87] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren, "The Program Dependence Graph and Its Use in Optimization", *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, p. 319–349, July 1987.

[Furs08] Grigori Fursin and Olivier Temam, "Collective Optimization", in *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, p. 34–49, Berlin, Heidelberg, 2008, Springer-Verlag.

[Furs11a] Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Bilha Mendelson, Ayal Zaks, Eric Courtois, François Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher Williams and Michael O'Boyle, "Milepost GCC: Machine Learning Enabled Self-tuning Compiler", *International Journal of Parallel Programming*, vol. 39, pp. 296–327, 06 2011.

[Furs11b] Grigori Fursin and Olivier Temam, "Collective Optimization: A Practical Collaborative Approach", *ACM Trans. Archit. Code Optim.*, vol. 7, no. 4, December 2011.

[Furs16] G. Fursin, Anton Lokhomotov and Ed Plowman, "Collective Knowledge", in *Design,Automation and Test in Europe Conference and Exibition (Date)*, pp. 864–869, 2016.

[GCC] "GCC passes", https://gcc.gnu.org/onlinedocs/gccint/Passes.html.

[Goog] "The Size and Quality of A Data Set", `https://developers.google.com/machine-learning/data-prep/construct/collect/data-size-quality`.

[Hint12] Geoffrey Hinton, li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Phuongtrang Nguyen, Tara Sainath and Brian Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups", *Signal Processing Magazine, IEEE*, vol. 29, pp. 82–97, 11 2012.

[Ipek08] E. Ipek, O. Mutlu, J. F. Martínez and R. Caruana, "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach", in *2008 International Symposium on Computer Architecture*, pp. 39–50, 2008.

[Jeff12] Brian Jeff, "Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration", in Patrick Groeneveld, Donatella Sciuto and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pp. 1143–1146, ACM, 2012.

[Jian10] Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen and Yaoqing Gao, "Exploiting Statistical Correlations for Proactive Prediction of Program Behaviors", in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, p. 248–256, New York, NY, USA, 2010, Association for Computing Machinery.

[Jose06] P. J. Joseph, Kapil Vaswani and Matthew J. Thazhuthaveetil, "A Predictive Performance Model for Superscalar Processors", in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, p. 161–170, USA, 2006, IEEE Computer Society.

[Kisu00] T. Kisuki, P. M. W. Knijnenburg and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation", in *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pp. 237–246, 2000.

[Kose98] Akira Koseki, H. Komastu and Yoshiaki Fukazawa, "A method for estimating optimal unrolling times for nested loops", 01 1998.

[Kulk04] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson and Douglas Jones, "Fast Searches for Effective Optimization Phase Sequences", *SIGPLAN Not.*, vol. 39, no. 6, p. 171–182, June 2004.

[Kulk12] Sameer Kulkarni and John Cavazos, "Mitigating the Compiler Optimization Phase-Ordering Problem Using Machine Learning", in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, p. 147–162, New York, NY, USA, 2012, Association for Computing Machinery.

[Leve80] Leverett, Cattell, Hobbs, Newcomer, Reiner, Schatz and Wulf, "An Overview of the Production-Quality Compiler-Compiler Project", *Computer*, vol. 13, no. 8, pp. 38–49, 1980.

[Liu19] Xiao Liu, Xiaoting Li, Rupesh Prajapati and Dinghao Wu, "DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing", 08 2019.

[Lulo] "Docker Milepost", https://github.com/lulogit/milepost-feature-extractor-docker.

[Magn14] Alberto Magni, Christophe Dubach and Michael O'Boyle, "Automatic Optimization of Thread-Coarsening for Graphics Processors", in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, p. 455–466, New York, NY, USA, 2014, Association for Computing Machinery.

[Miko13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean, "Distributed Representations of Words and Phrases and Their Compositionality", in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, p. 3111–3119, Red Hook, NY, USA, 2013, Curran Associates Inc.

[Mohr12] Mehryar Mohri, Afshin Rostamizadeh and Ameet Talwalkar, "Foundations of machine learning", 2012.

[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, 2006.

[Mou16] Lili Mou, Ge Li, Lu Zhang, Tao Wang and Zhi Jin, "Convolutional Neural Networks over Tree Structures for Programming Language Processing", in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, p. 1287–1293, AAAI Press, 2016.

[Mous18] Sajad Mousavi, Michael Schukat and Enda Howley, "Deep Reinforcement Learning: An Overview", 06 2018.

[Nobr16] Ricardo Nobre, Luiz G.A. Martins and João M.P. Cardoso, "A Graph-Based Iterative Compiler Pass Selection and Phase Ordering Approach", 2016.

[Park11a] Eunjung Park, Sameer Kulkarni and John Cavazos, "An Evaluation of Different Modeling Techniques for Iterative Compilation", in *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, p. 65–74, New York, NY, USA, 2011, Association for Computing Machinery.

[Park11b] Eunjung Park, Louis-Noel Pouche, John Cavazos, Albert Cohen and P. Sadayappan, "Predictive Modeling in a Polyhedral Optimization Space", in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, p. 119–129, USA, 2011, IEEE Computer Society.

[Park12] Eunjung Park, John Cavazos and Marco A. Alvarez, "Using Graph-Based Program Characterization for Predictive Modeling", in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, p. 196–206, New York, NY, USA, 2012, Association for Computing Machinery.

[Park14] EunJung (EJ) Park, Christos Kartsaklis and John Cavazos, “HERCULES: Strong patterns towards more intelligent predictive modeling”, 09 2014.

[Pere03] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood and Brad Calder, “Using SimPoint for Accurate and Efficient Simulation”, in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS ’03, p. 318–319, New York, NY, USA, 2003, Association for Computing Machinery.

[Perf] “Linux perf”, https://perf.wiki.kernel.org/index.php/MainPage.

[Puri13] Suresh Purini and Lakshya Jain, “Finding Good Optimization Sequences Covering Program Space”, *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, January 2013.

[Rao09] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang and George Yin, “VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-Configuration”, in *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC ’09, p. 137–146, New York, NY, USA, 2009, Association for Computing Machinery.

[Reji17] R. S. Rejitha, Shajulin Benedict, Suja A. Alex and Shany Infanto, “Energy Prediction of CUDA Application Instances Using Dynamic Regression Models”, *Computing*, vol. 99, no. 8, p. 765–790, August 2017.

[Sher02] Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder, “Automatically Characterizing Large Scale Program Behavior”, in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, p. 45–57, New York, NY, USA, 2002, Association for Computing Machinery.

[Simo13] Douglas Simon, John Cavazos, Christian Wimmer and Sameer Kulkarni, “Automatic Construction of Inlining Heuristics Using Machine Learning”, in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO ’13, p. 1–12, USA, 2013, IEEE Computer Society.

[Step05] M. Stephenson and Saman P. Amarasinghe, “Predicting unroll factors using supervised classification”, *International Symposium on Code Generation and Optimization*, pp. 123–134, 2005.

[Steu08] R. Steuer, “Multiple Criteria Optimization: Theory, Computation, And Application (Wiley Series In Probability And Statistics) By Ralph E. Steuer”, 2008.

[Tayl17] Ben Taylor, Vicent Sanz Marco and Zheng Wang, “Adaptive Optimization for OpenCL Programs on Embedded Heterogeneous Systems”, in *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2017, p. 11–20, New York, NY, USA, 2017, Association for Computing Machinery.

[Tria03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani and David I. August, “Compiler Optimization-Space Exploration”, in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO ’03, p. 204–215, USA, 2003, IEEE Computer Society.

[Vegd82] Steven R. Vegdahl, “Phase Coupling and Constant Generation in an Optimizing Microcode Compiler”, *SIGMICRO Newsl.*, vol. 13, no. 4, p. 125–133, October 1982.

[Wang09] Zheng Wang and Michael F.P. O'Boyle, "Mapping Parallelism to Multi-Cores: A Machine Learning Based Approach", in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, p. 75–84, New York, NY, USA, 2009, Association for Computing Machinery.

[Wang10] Zheng Wang and Michael F.P. O'Boyle, "Partitioning Streaming Parallelism for Multi-Cores: A Machine Learning Based Approach", in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, p. 307–318, New York, NY, USA, 2010, Association for Computing Machinery.

[Wang14] Zheng Wang, Dominik Grewe and Michael F. P. O'boyle, "Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-Based Heterogeneous Systems", *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, December 2014.

[Wen14] Yuan Wen, Zheng Wang and Michael O'Boyle, "Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms", 12 2014.

[Whit19] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus and Denys Poshyvanyk, "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities", in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 479–490, 2019.

[Wils94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam and John L. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers", *SIGPLAN Not.*, vol. 29, no. 12, p. 31–37, December 1994.

[Zahr16] Mohamed Zahran, "Heterogeneous Computing: Here to Stay: Hardware and Software Perspectives", *Queue*, vol. 14, no. 6, p. 31–42, December 2016.