ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Σχεδίαση και Υλοποίηση ενός Optimization Pass στο LLVM για την Αναγνώριση Ιδιωμάτων Κώδικα

της

Πολυξένης Μότση

Εκπόνηση πτυχιακής ως μέρος του
Μεταπτυχιακού Τίτλου Σπουδών

στη
Σχολή Οικονομίας και Τεχνολογίας
Τμήμα Επιστήμης Υπολογιστών

Επιβλέπων Καθηγητής: Δρ. Γρηγόριος Δημητρουλάκος
Συνεπιβλέπων Καθηγητής: Δρ. Κωσταντίνος Βασιλάκης

Ημερομηνία

Δήλωση Συγγραφικής Ιδιότητας

Εγώ, η Πολυξένη Μότση, δηλώνω ότι αυτή η πτυχιακή εργασία με τίτλο, Σχεδίαση και Υλοποίηση ενός Οπτιμιζατιον Πασς στο ΛΛ΄Μ για την Αναγνώριση Ιδιωμάτων Κώδικα, και η δουλειά που παρουσιάζεται σε αυτή είναι δικά μου. Επιβεβαιώνω ότι:

- Αυτή η δουλειά πραγματοποιήθηκε ολοκληρωτικά ή κυρίως κατά την υποψηφιότητά μου για τίτλο προπτυχιακών σπουδών σε αυτό το πανεπιστήμιο.
- Όπου οποιοδήποτε μέρος αυτής της πτυχιακής εργασίας έχει προηγουμένως κατατεθεί για την απόκτηση πτυχίου ή άλλου τίτλου σε αυτό ή άλλο πανεπιστήμιο, αυτό διατυπώνεται ξεκάθαρα.
- Όπου έχω συμβουλευτεί την δημοσιευμένη δουλειά τρίτων, αυτό αποδίδεται ορθώς.
- Όπου έχω παραθέσει από δουλειά τρίτων, η πηγή δίνεται πάντα. Με εξαίρεση αυτές τις παραθέσεις, αυτή η πτυχιακή εργασία είναι εξ ολοκλήρου προσωπική μου δουλειά.
- Έχω παραθέσει όλες τις κύριες πηγές βοήθειας.
- Όπου αυτή η πτυχιακή εργασία είναι βασισμένη σε συνεργατική δουλειά δική μου και τρίτων, έχω καταστήσει ξεκάθαρο ποια κομμάτια έχουν πραγματοποιηθεί από άλλους και πώς συνέβαλα εγώ.

Υπογραφή: _____

Ημερομηνία: 16/02/2026

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Σύνοψη

Σχολή Οικονομίας και Τεχνολογίας
Τμήμα Επιστήμης Υπολογιστών

Μεταπτυχιακός Τίτλος Σπουδών

της Πολυξένης Μότση

Η σύγχρονη μεταγλώττιση βασίζεται στην ικανότητα των μεταγλωττιστών να μετασχηματίζουν κώδικα υψηλού επιπέδου σε αποδοτικές εντολές μηχανής. Ωστόσο, κατά τη διαδικασία μετάφρασης στην Ενδιάμεση Αναπαράσταση (Intermediate Representation - IR), η αρχική πρόθεση του προγραμματιστή συχνά συσκοτίζεται, οδηγώντας σε αυτό που ονομάζεται «Σημασιολογικό Χάσμα» (Semantic Gap). Πολύπλοκα μαθηματικά ιδιώματα υποβιβάζονται σε φλύαρες ακολουθίες γενικών εντολών, εμποδίζοντας την αξιοποίηση εξειδικευμένου υλικού.

Η παρούσα διπλωματική εργασία αντιμετωπίζει αυτό το πρόβλημα μέσω της σχεδίασης και υλοποίησης του `PatternSelectPass`, ενός περάσματος βελτιστοποίησης (Optimization Pass) ενταγμένου στον Νέο Διαχειριστή Περασμάτων (New Pass Manager) του LLVM. Το προτεινόμενο εργαλείο χρησιμοποιεί προηγμένες τεχνικές ταύτισης προτύπων (pattern matching) για τον εντοπισμό επτά (7) κρίσιμων αριθμητικών ιδιωμάτων, συμπεριλαμβανομένων της Περιστροφής Bit (Rotate), της Καταμέτρησης Πληθυσμού (PopCount), της Ακέραιας Απόλυτης Τιμής (Abs) και των πράξεων Ελαχίστου/Μεγίστου (Min/Max).

Μόλις αναγνωριστούν, αυτές οι ακολουθίες αντικαθίστανται αυτόματα από τις αντίστοιχες Εγγενείς Συναρτήσεις (Intrinsics) του LLVM. Η πειραματική αξιολόγηση σε στοχευμένα σενάρια ελέγχου κατέδειξε ότι το PatternSelectPass επιτυγχάνει μείωση του στατικού αριθμού εντολών κατά **52.9%** σε σύγκριση με τον μη βελτιστοποιημένο κώδικα, ξεπερνώντας σε συγκεκριμένες περιπτώσεις ακόμη και τον καθιερωμένο βελτιστοποιητή `InstCombine`. Επιπλέον, η χρήση intrinsics οδηγεί σε σημαντική απλοποίηση του Γράφου Ροής Ελέγχου (Control Flow Graph), εξαλείφοντας περιττές διακλαδώσεις και προετοιμάζοντας το έδαφος για βέλτιστη παραγωγή κώδικα στο backend.

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ

Σύνοψη

Σχολή Οικονομίας και Τεχνολογίας
Τμήμα Επιστήμης Υπολογιστών

Μεταπτυχιακός Τίτλος Σπουδών

της Πολυξένης Μότση

Modern compilation relies on the ability of compilers to transform high-level code into efficient machine instructions. However, during the translation process to Intermediate Representation (IR), the programmer's original intent is often obscured, leading to what is known as the "Semantic Gap". Complex mathematical idioms are lowered into verbose sequences of generic instructions, hindering the utilization of specialized hardware capabilities.

This thesis addresses this problem through the design and implementation of `PatternSelectPass`, an optimization pass integrated into LLVM's New Pass Manager. The proposed tool utilizes advanced pattern matching techniques to detect seven (7) critical arithmetic idioms, including Bitwise Rotate, Population Count, Integer Absolute Value, and Min/Max operations.

Once recognized, these sequences are automatically replaced by the corresponding LLVM Intrinsics. Experimental evaluation on targeted test scenarios demonstrated that `Pattern Select Pass` achieves a **52.9%** reduction in static instruction count compared to unoptimized code, outperforming in specific cases even the established `InstCombine` optimizer. Furthermore, the use of intrinsics leads to a significant simplification of the Control Flow Graph (CFG), eliminating unnecessary branches and paving the way for optimal code generation in the backend.

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Συντομογραφίες

Κεφάλαιο 1

Εισαγωγή

1.1 Ο Ρόλος των Μεταγλωττιστών στη Σύγχρονη Υπολογιστική

Η σύγχρονη αρχιτεκτονική υπολογιστών έχει εξελιχθεί σε ένα τοπίο τεράστιας πολυπλοκότητας. Οι σύγχρονοι επεξεργαστές διαθέτουν προηγμένα σύνολα εντολών, διανυσματικές μονάδες (SIMD) και βαθιές σωληνώσεις (pipelines) σχεδιασμένες για τη μεγιστοποίηση της απόδοσης. Ωστόσο, η ανάπτυξη λογισμικού έχει κινηθεί προς την αντίθετη κατεύθυνση, ευνοώντας γλώσσες υψηλού επιπέδου όπως η C++ που θέτουν ως προτεραιότητα την αφαίρεση και την παραγωγικότητα του προγραμματιστή έναντι του άμεσου χειρισμού του υλικού.

Αυτή η διχοτόμηση δημιουργεί μια σημαντική πρόκληση: πώς να μεταφραστεί η αφηρημένη λογική υψηλού επιπέδου στον ακριβή, εξαιρετικά βελτιστοποιημένο κώδικα μηχανής που απαιτείται από το σύγχρονο υλικό. Αυτή είναι η πρωταρχική ευθύνη του μεταγλωττιστή. Συγκεκριμένα, η υποδομή μεταγλωττιστή LLVM διαδραματίζει καθοριστικό ρόλο σε αυτή τη διαδικασία αναλύοντας τον πηγαίο κώδικα και εφαρμόζοντας μετασχηματισμούς για τη βελτίωση της ταχύτητας εκτέλεσης και τη μείωση του μεγέθους.

Σε αυτό το πλαίσιο, τα περάσματα βελτιστοποίησης (optimization passes) του μεταγλωττιστή αποτελούν τη γέφυρα μεταξύ της αφαίρεσης υψηλού επιπέδου και της αποδοτικότητας υλικού χαμηλού επιπέδου. Εντοπίζοντας αναποτελεσματικές ακολουθίες κώδικα και αντικαθιστώντας τες με βελτιστοποιημένα ισοδύναμα, ο μεταγλωττιστής διασφαλίζει ότι το λογισμικό αξιοποιεί πλήρως τις δυνατότητες του υποκείμενου υλικού χωρίς να απαιτείται από τον προγραμματιστή να γράφει κώδικα assembly χειροκίνητα.

1.2 Διατύπωση του Προβλήματος: Το Κενό στην 'Επιλογή Εντολών'

Όταν ένα frontend (όπως το Clang) μεταφράζει κώδικα υψηλού επιπέδου σε Ενδιάμεση Αναπαράσταση LLVM (IR), ακολουθεί πιστά το πρότυπο της γλώσσας, παράγοντας συχνά μια ¨φλύαρη' ή απλοϊκή ακολουθία εντολών. Αυτό συμβαίνει διότι το frontend εστιάζει στη σημασιολογική ορθότητα και όχι απαραίτητα στη βέλτιστη απόδοση υλικού.

Ένα κλασικό παράδειγμα αναποτελεσματικότητας αφορά τις βασικές αριθμητικές πράξεις, όπως τον πολλαπλασιασμό ή το υπόλοιπο διαίρεσης με δυνάμεις του δύο. Θεωρήστε την εντολή:

$$y = x * 8;$$

Μια αφελής μεταγλώττιση θα παράγει την εντολή πολλαπλασιασμού `mul`, η οποία, σε επίπεδο υλικού, έχει υψηλότερη λανθάνουσα καθυστέρηση (latency) από τις απλές λογικές πράξεις. Ωστόσο, αναγνωρίζοντας ότι το 8 είναι δύναμη του δύο (2^3), η πράξη μπορεί να αντικατασταθεί από μία Αριστερή Ολίσθηση ($x \ll 3$), η οποία εκτελείται σχεδόν ακαριαία από την ALU. Αντίστοιχα, το υπόλοιπο διαίρεσης ($x\%16$), που κανονικά απαιτεί την εξαιρετικά αργή εντολή `urem` (διαίρεση), μπορεί να μετατραπεί σε μία ταχύτατη εντολή `and` ($x \ \& \ 15$). Αυτή η τεχνική, γνωστή ως Strength Reduction, συχνά παραλείπεται σε μη βελτιστοποιημένο IR.

Ένα δεύτερο ζήτημα προκύπτει όταν μαθηματικές έννοιες εκφράζονται μέσω ροής ελέγχου. Για παράδειγμα, ο υπολογισμός της απόλυτης τιμής (`abs(x)`) ή του ελαχίστου (`min(a, b)`) συχνά γράφεται ως:

```
x < 0 ?  -x :  x
```

Στο IR, αυτό μεταφράζεται σε εντολές σύγκρισης (`icmp`) και άλματος (`br`), δημιουργώντας διακλαδώσεις που επιβαρύνουν τον Branch Predictor του επεξεργαστή. Αν ο μεταγλωττιστής δεν αναγνωρίσει το ιδίωμα ώστε να χρησιμοποιήσει ειδικές εντολές (όπως CMOV ή PABS), η απόδοση μειώνεται δραματικά λόγω των διακλαδώσεων.

Τέλος, το πρόβλημα μεγεθύνεται σε πολύπλοκους αλγορίθμους όπως η Καταμέτρηση Πληθυσμού (Population Count). Χωρίς τη χρήση intrinsics, οι προγραμματιστές χρησιμοποιούν τεχνικές όπως ο αλγόριθμος SWAR, που απαιτεί δεκάδες εντολές ολισθήσεων και προσθέσεων. Αν ο μεταγλωττιστής δεν αντιληφθεί ότι αυτή η ¨σούπα' εντολών αντιπροσωπεύει μια καταμέτρηση bits, θα χάσει την ευκαιρία να χρησιμοποιήσει τη μοναδική, εξειδικευμένη εντολή υλικού POPCNT.

Αυτό το ζήτημα είναι γνωστό ως πρόβλημα Αναγνώρισης Ιδιωμάτων (Idiom Recognition). Η αποτυχία αναγνώρισης αυτών των μοτίβων σε επίπεδο IR αναγκάζει το backend να παράγει μη βέλτιστο κώδικα μηχανής, οδηγώντας σε αυξημένο μέγεθος κώδικα και σπατάλη κύκλων ρολογιού.

1.3 Κίνητρο και Πεδίο Εφαρμογής

Το πρωταρχικό κίνητρο αυτής της διατριβής είναι η αντιμετώπιση του κενού αναγνώρισης ιδιωμάτων μέσω της ανάπτυξης ενός Ανεξάρτητου από τον Στόχο Περάσματος Βελτιστοποίησης (Target-Independent Optimization Pass) εντός της υποδομής LLVM. Αντί να βασιζόμαστε αποκλειστικά σε εξειδικευμένο υποβιβασμό στο backend για τη διόρθωση αναποτελεσματικού κώδικα, στοχεύουμε στην εκτέλεση αυτών των βελτιστοποιήσεων νωρίς στο 'Middle-end' του μεταγλωττιστή.

Κανονικοποιώντας (canonicalizing) αυτά τα μοτίβα σε επίπεδο IR, διασφαλίζουμε ότι ο κώδικας είναι καθαρός και τυποποιημένος πριν φτάσει στο backend. Το πεδίο εφαρμογής αυτού του έργου εστιάζει στον εντοπισμό και την αντικατάσταση επτά συγκεκριμένων αριθμητικών και bitwise μοτίβων που είναι κρίσιμα για την απόδοση:

- Περιστροφή Bit (Bitwise Rotate)
- Ακέραια Απόλυτη Τιμή (Integer Absolute Value - Abs)
- Καταμέτρηση Πληθυσμού (Population Count - PopCount)
- Πολλαπλασιασμός και Υπόλοιπο με Δυνάμεις του Δύο
- Αντιμετάθεση Byte (Byte Swap)
- Λειτουργίες Ελαχίστου/Μεγίστου (Min/Max Operations)

1.4 Συνεισφορές της Διατριβής

Η παρούσα διατριβή παρουσιάζει τον σχεδιασμό, την υλοποίηση και την αξιολόγηση του PatternSelectPass, ενός προσαρμοσμένου περάσματος βελτιστοποίησης που κατασκευάστηκε χρησιμοποιώντας τον New Pass Manager του LLVM. Οι βασικές συνεισφορές αυτής της εργασίας είναι οι εξής:

1. **Υλοποίηση ενός Function Pass:** Αναπτύξαμε ένα αυτόνομο πέρασμα μεταγλωττιστή χρησιμοποιώντας C++ και το σύγχρονο API LLVM PassInfoMixin.

2. **Αποδοτική Ταύτιση Μοτίβων:** Αξιοποιώντας τη βιβλιοθήκη PatternMatch.h, το πέρασμα σαρώνει αποτελεσματικά το Δέντρο Εντολών (Instruction Tree) για να ανιχνεύσει πολύπλοκα ιδιώματα χωρίς δαπανηρή ανάλυση.

3. **Βελτίωση Απόδοσης:** Τα πειραματικά αποτελέσματα δείχνουν ότι το προτεινόμενο πέρασμα επιτυγχάνει μείωση 52.9% στον αριθμό εντολών για συγκεκριμένα benchmarks, εντοπίζοντας αποτελεσματικά πολύπλοκα ιδιώματα που συνήθως απαιτούν υποβιβασμό στο backend.

4. **Επαλήθευση Ορθότητας:** Το πέρασμα διατηρεί τη μορφή Static Single Assignment (SSA) και έχει επικυρωθεί χρησιμοποιώντας τον μηχανισμό opt -verify και εκτενείς δοκιμές παλινδρόμησης (regression tests).

1.5 Δομή της Διατριβής

Το υπόλοιπο της παρούσας διατριβής οργανώνεται ως εξής:

- **Κεφάλαιο 2: Υπόβαθρο** παρέχει το θεωρητικό πλαίσιο, εξηγώντας τη δομή του LLVM IR, τη μορφή SSA και τον ρόλο των Intrinsics.

- **Κεφάλαιο 3: Σχεδιασμός και Υλοποίηση** περιγράφει λεπτομερώς την αρχιτεκτονική του PatternSelectPass, αναλύοντας τη λογική πίσω από τους matchers και τους rewriters.

- **Κεφάλαιο 4: Αξιολόγηση** παρουσιάζει τα πειραματικά αποτελέσματα, προσφέροντας μια ποσοτική σύγκριση μεταξύ του περάσματός μας και του τυπικού περάσματος InstCombine.

- **Κεφάλαιο 5: Συμπεράσματα** συνοψίζει τα ευρήματα και συζητά τους περιορισμούς και τις μελλοντικές εργασίες.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Η Υποδομή Μεταγλωττιστή LLVM

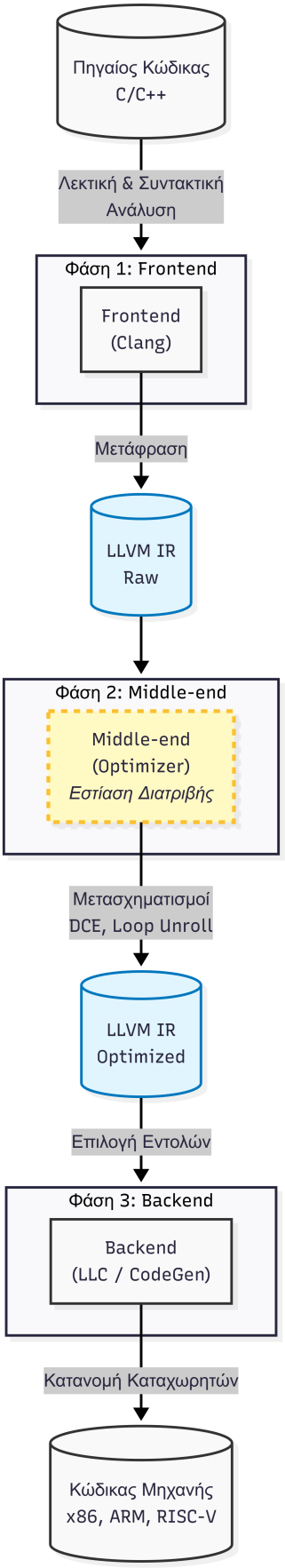
Το έργο LLVM βασίζεται σε μια αρθρωτή (modular) και επαναχρησιμοποιήσιμη τεχνολογία μεταγλωττιστή που ακολουθεί έναν αυστηρό σχεδιασμό τριών φάσεων: το Frontend (Εμπρόσθιο άκρο), το Middle-end (Βελτιστοποιητής) και το Backend (Οπίσθιο άκρο).

Αρχικά, το **Frontend** (συνήθως το Clang για γλώσσες C/C++) εκτελεί λεκτική ανάλυση, συντακτική ανάλυση και σημασιολογικό έλεγχο. Ο πρωταρχικός του στόχος είναι η μετάφραση του πηγαίου κώδικα σε LLVM IR, μια γλώσσα χαμηλού επιπέδου, ανεξάρτητη από το υλικό.

Στη συνέχεια, το **Middle-end** — όπου εστιάζει η παρούσα διατριβή — λαμβάνει την IR και εκτελεί μια σειρά (pipeline) περασμάτων βελτιστοποίησης. Αυτά τα περάσματα λειτουργούν αποκλειστικά σε επίπεδο IR, εκτελώντας μετασχηματισμούς (όπως απαλοιφή νεκρού κώδικα ή ξετύλιγμα βρόχων) για τη βελτιστοποίηση της αποδοτικότητας του προγράμματος χωρίς να δεσμεύονται από μια συγκεκριμένη αρχιτεκτονική υλικού.

Τέλος, το **Backend** λαμβάνει τη βελτιστοποιημένη IR και εκτελεί επιλογή εντολών και κατανομή καταχωρητών για την παραγωγή βελτιστοποιημένου κώδικα μηχανής για τη συγκεκριμένη αρχιτεκτονική στόχο (π.χ. x86-64, ARM, RISC-V).

Το Σχήμα 2.1 συνοψίζει τη ροή δεδομένων μεταξύ των τριών αυτών φάσεων.



Σχημα 2.1: Η αρχιτεκτονική τριών φάσεων του LLVM. Διακρίνονται το Frontend, το Middle-end και το Backend, με την LLVM IR να αποτελεί τον συνδετικό κρίκο.

2.2 Το Πλαίσιο Εργασίας LLVM

Το LLVM είναι μια εκτενής συλλογή αρθρωτών και επαναχρησιμοποιήσιμων τεχνολογιών μεταγλωττιστή και εργαλειοθήκης. Αρχικά αναπτύχθηκε στο Πανεπιστήμιο του Illinois στο Urbana-Champaign από τον Chris Lattner και έχει εξελιχθεί από ερευνητικό έργο στην βιομηχανική πρότυπη υποδομή για την ανάπτυξη μεταγλωττιστών. Αξίζει να σημειωθεί ότι, αν και το ακρωνύμιο αρχικά σήμαινε Low Level Virtual Machine, πλέον το έργο έχει ξεπεράσει αυτόν τον ορισμό και το όνομα LLVM χρησιμοποιείται ως διακριτικό σήμα για το σύνολο της υποδομής.

Σε αντίθεση με τους παραδοσιακούς μονολιθικούς μεταγλωττιστές (όπως ο GCC παλαιότερων εκδόσεων), το LLVM έχει σχεδιαστεί εξ αρχής ως ένα σύνολο βιβλιοθηκών που διαχωρίζουν σαφώς το frontend, τον βελτιστοποιητή και το backend. Αυτή η αρχιτεκτονική επιτρέπει στους προγραμματιστές να επιλέγουν και να ενσωματώνουν συγκεκριμένα εξαρτήματα (όπως τον JIT Compiler ή τον Disassembler) στις εφαρμογές τους, αντί να δεσμεύονται από μια ενιαία, άκαμπτη εκτελέσιμη μονάδα.

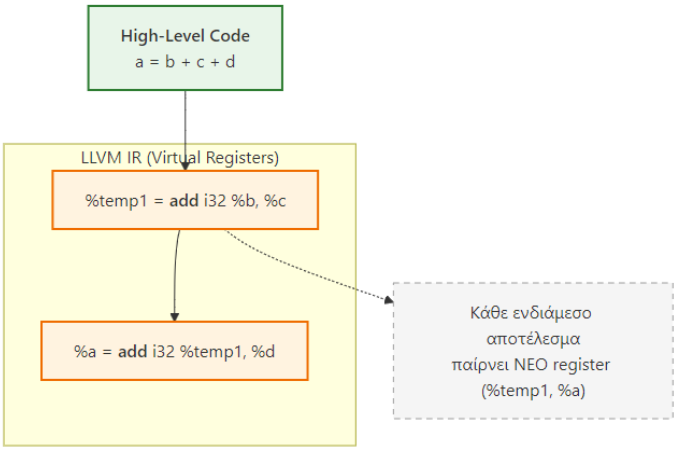
Αυτή η αρθρωτή δομή επιλύει το πρόβλημα της συμβατότητας γλωσσών και αρχιτεκτονικών ($M \times N$). Το frontend Clang μεταφράζει C/C++ σε Ενδιάμεση Αναπαράσταση LLVM (IR), ενώ άλλα frontends (όπως της Rust, της Swift ή της Julia) παράγουν την ίδια IR. Επιπλέον, η «ομπρέλα» του έργου έχει επεκταθεί για να συμπεριλάβει κρίσιμα εργαλεία όπως τον αποσφαλματωτή LLDB και τον συνδέτη LLD. Αυτή η ενιαία και ολοκληρωμένη προσέγγιση οδήγησε στην ευρεία υιοθέτηση του LLVM από μεγάλες τεχνολογικές εταιρείες, συμπεριλαμβανομένων των Apple, Google και Meta, καθιστώντας το τη ραχοκοκαλιά των σύγχρονων περιβαλλόντων ανάπτυξης λογισμικού.

2.3 Ενδιάμεση Αναπαράσταση LLVM (IR)

Στον πυρήνα της υποδομής LLVM βρίσκεται η Ενδιάμεση Αναπαράσταση LLVM (IR). Πρόκειται για μια γλώσσα προγραμματισμού χαμηλού επιπέδου, παρόμοια με τη συμβολική γλώσσα (assembly), αλλά με αυστηρούς τύπους και ανεξάρτητη από την υποκείμενη αρχιτεκτονική μηχανής.

Η LLVM IR ακολουθεί μια αρχιτεκτονική **Load/Store**, όπου τα δεδομένα πρέπει να μεταφερθούν από τη μνήμη σε καταχωρητές για να επεξεργαστούν. Ωστόσο, σε αντίθεση με τους φυσικούς καταχωρητές μιας CPU (π.χ. rax, ebx), η IR χρησιμοποιεί ένα άπειρο σύνολο **Εικονικών Καταχωρητών (Virtual Registers)**, οι οποίοι συμβολίζονται με το χαρακτήρα %.

Η χρήση των εικονικών καταχωρητών για την τμηματοποίηση σύνθετων εντολών φαίνεται στο Σχήμα 2.2.



Σχημα 2.2: Απεικόνιση χρήσης εικονικών καταχωρητών στην LLVM IR για την εκτέλεση σύνθετων πράξεων μέσω εντολών τριών διευθύνσεων.

Η IR υποστηρίζει τρεις ισόμορφες μορφές, επιτρέποντας ευελιξία σε κάθε στάδιο της μεταγλώττισης:

- Μια μορφή στη μνήμη (In-Memory Graph) για γρήγορη επεξεργασία από τον μεταγλωττιστή.
- Μια μορφή στο δίσκο (Bitcode - .bc) για αποθήκευση και σύνδεση (linking).
- Μια αναγνώσιμη από τον άνθρωπο μορφή Assembly (αρχεία .ll) για αποσφαλμάτωσης και ανάλυση.

2.3.1 Μορφή Στατικής Μονής Ανάθεσης (SSA)

Ένα καθοριστικό χαρακτηριστικό της LLVM IR είναι η χρήση της μορφής **Static Single Assignment (SSA)**. Στη μορφή SSA, σε κάθε εικονικό καταχωρητή ανατίθεται τιμή **ακριβώς μία φορά**. Εάν μια μεταβλητή του πηγαίου κώδικα (π.χ. x) τροποποιείται πολλαπλές φορές, ο μεταγλωττιστής δημιουργεί νέες εκδόσεις αυτής της μεταβλητής (π.χ. x1, x2, x3).

Αυτή η ιδιότητα απλοποιεί δραματικά την ανάλυση ροής δεδομένων, επειδή δημιουργεί σαφείς αλυσίδες Ορισμού-Χρήσης (Def-Use Chains). Κάθε χρήση μιας τιμής συνδέεται άμεσα με τον μοναδικό ορισμό της, εξαλείφοντας την ανάγκη για πολύπλοκους αλγορίθμους ανάλυσης της εμβέλειας (reaching definitions).

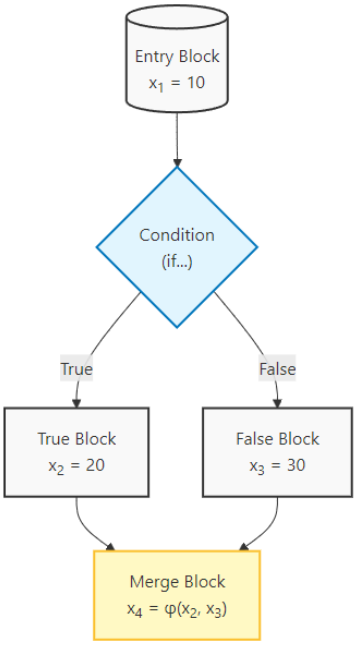
2.3.1.1 Ο Κόμβος Phi (Phi Node)

Το πρόβλημα της μοναδικής ανάθεσης εμφανίζεται όταν συγχωνεύονται κλάδοι ροής ελέγχου. Για παράδειγμα, μετά από μια εντολή if-else, μια μεταβλητή μπορεί να έχει λάβει τιμή είτε από το true block είτε από το false block. Για να λύσει αυτό το πρόβλημα χωρίς να παραβιάσει τον κανόνα SSA, το LLVM χρησιμοποιεί μια ειδική εντολή που ονομάζεται Κόμβος **Phi (phi node)**.

Η εντολή `phi` τοποθετείται στην αρχή ενός Βασικού Μπλοκ και επιλέγει τη σωστή τιμή εξετάζοντας από ποιο προηγούμενο μπλοκ προήλθε η ροή ελέγχου. Συντακτικά, έχει τη μορφή:

```
%val = phi i32 [ %a, %block1 ], [ %b, %block2 ]
```

Η λειτουργία του κόμβου Phi σε έναν Γράφο Ροής Ελέγχου, όπου επιλέγεται η τιμή ανάλογα με το μονοπάτι εκτέλεσης, παρουσιάζεται στο Σχήμα 2.3.



Σχημα 2.3: Γράφος Ροής Ελέγχου (CFG) σε μορφή SSA. Ο κόμβος Phi στο Merge Block επιλέγει τη σωστή έκδοση της μεταβλητής ανάλογα με το μονοπάτι εκτέλεσης.

2.3.2 Τύποι Δεδομένων και Μεταδεδομένα

Η LLVM IR είναι μια ισχυρά τυποποιημένη γλώσσα (strongly typed). Υποστηρίζει μια ευρεία γκάμα τύπων που είναι απαραίτητοι για σύγχρονες βελτιστοποιήσεις:

- **Πρωτογενείς Τύποι:** Ακέραιοι αυθαίρετου πλάτους bit (π.χ. i1 για boolean, i8, i32, i128), αριθμοί κινητής υποδιαστολής (half, float, double) και δείκτες.
- **Σύνθετοι Τύποι:** Πίνακες (Arrays), δομές (Structs) και, σημαντικό για αυτή τη διατριβή, **Διανύσματα (Vectors)**. Ένας τύπος όπως `<4 x i32>` επιτρέπει την εφαρμογή εντολών SIMD.

Επιπλέον, οι εντολές μπορούν να σχολιαστούν με **Μεταδεδομένα (Metadata)** — βοηθητικές πληροφορίες που δεν αλλάζουν τη σημασιολογία του κώδικα αλλά καθοδηγούν τις βελτιστοποιήσεις. Συνήθη παραδείγματα περιλαμβάνουν πληροφορίες αποσφαλμάτωσης (debug info) ή υποδείξεις για τον βρόχο (π.χ. llvm.loop.unroll.disable).

2.3.3 Παράδειγμα Μετάφρασης Κώδικα

Για την καλύτερη κατανόηση της δομής της LLVM IR, παραθέτουμε ένα συγκεκριμένο παράδειγμα μετάφρασης μιας απλής συνάρτησης από τη γλώσσα C στην Ενδιάμεση Αναπαράσταση.

Έστω η παρακάτω συνάρτηση σε C, η οποία υπολογίζει το άθροισμα δύο ακεραίων και πολλαπλασιάζει το αποτέλεσμα με το 5:

```c
int calculate(int a, int b) {
    int sum = a + b;
    return sum * 5;
}
```

Listing 2.1: Κώδικας C (Source)

Ο μεταγλωττιστής μετατρέπει τον παραπάνω κώδικα στην ακόλουθη μορφή LLVM IR (σε μορφή SSA):

```
; Function definition returning i32 (32-bit integer)
define i32 @calculate(i32 %a, i32 %b) {
entry:
  ; %tmp1 is a new Virtual Register
  %tmp1 = add i32 %a, %b       ; sum = a + b

  ; Use %tmp1 for the next operation
  %result = mul i32 %tmp1, 5   ; result = sum * 5

  ; Return the value
```

```
11   ret i32 %result
12 }
```

Listing 2.2: Κώδικας LLVM IR

Στο παραπάνω απόσπασμα διακρίνουμε τα βασικά συστατικά της γλώσσας:

- **Καθολικά Σύμβολα (`@`):** Το όνομα της συνάρτησης `@calculate` ξεκινά με το σύμβολο `@`, υποδηλώνοντας ότι είναι καθολικό (global).
- **Τοπικοί Καταχωρητές (`%`):** Οι μεταβλητές `%a`, `%b`, `%tmp1` και `%result` είναι τοπικοί εικονικοί καταχωρητές.
- **Τύποι (`i32`):** Κάθε τελεστέος συνοδεύεται ρητά από τον τύπο του (ακέραιος 32-bit).
- **Βασικό Μπλοκ (`entry:`):** Η ετικέτα `entry:` ορίζει την αρχή του πρώτου Βασικού Μπλοκ της συνάρτησης.

2.4 Ο Διαχειριστής Περασμάτων (Pass Manager)

Οι βελτιστοποιήσεις στο LLVM δεν είναι μονολιθικές, αλλά οργανώνονται ως μικρές, ανεξάρτητες μονάδες κώδικα που ονομάζονται **Περάσματα** (Passes). Ένα πέρασμα διασχίζει τμήματα της Ενδιάμεσης Αναπαράστασης (IR) είτε για να συλλέξει πληροφορίες (Analysis Pass) είτε για να τροποποιήσει τον κώδικα (Transformation Pass). Η εκτέλεση, η σειρά και η διαχείριση των πόρων αυτών των περασμάτων ενορχηστρώνεται από τον Διαχειριστή Περασμάτων.

Οι σύγχρονες εκδόσεις του LLVM (από την έκδοση 13 και έπειτα) χρησιμοποιούν αποκλειστικά τον **Νέο Διαχειριστή Περασμάτων (New Pass Manager - NPM)**, ο οποίος αντικατέστησε την παλαιά υποδομή (Legacy PM).

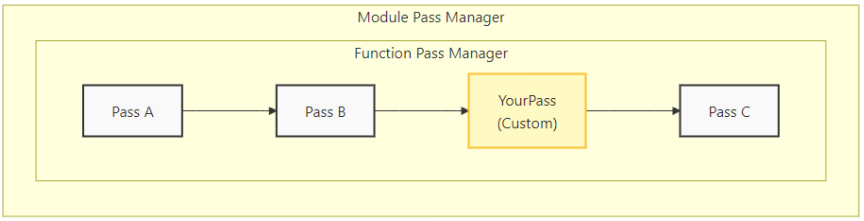
2.4.1 Ιεραρχία και Διοχέτευση (Nesting & Pipeline)

Ο NPM λειτουργεί με βάση ένα ιεραρχικό μοντέλο διοχέτευσης (pipeline). Σε αντίθεση με μια επίπεδη λίστα εκτέλεσης, ο NPM επιτρέπει την ένθεση (nesting) διαχειριστών διαφορετικών επιπέδων λεπτομέρειας. Τα βασικά επίπεδα (από το πιο γενικό στο πιο ειδικό) είναι:

- **Module Pass:** Επεξεργάζεται ολόκληρο το αρχείο κώδικα (π.χ. καθολικές μεταβλητές).
- **CGSCC Pass (Call Graph SCC):** Επεξεργάζεται ομάδες συναρτήσεων που καλούν η μία την άλλη (χρήσιμο για βελτιστοποιήσεις αναδρομής).
- **Function Pass:** Λειτουργεί στο σώμα μιας μεμονωμένης συνάρτησης. Το PatternSelectPass που αναπτύχθηκε σε αυτή τη διατριβή ανήκει σε αυτή την κατηγορία.
- **Loop Pass:** Εστιάζει αποκλειστικά στους βρόχους εντός μιας συνάρτησης.

Αυτή η δομή επιτρέπει σε έναν Module Pass Manager να περιέχει έναν Function Pass Manager, ο οποίος με τη σειρά του εκτελεί μια σειρά από περάσματα σε κάθε συνάρτηση ξεχωριστά, βελτιστοποιώντας τη χρήση της κρυφής μνήμης (cache locality) του επεξεργαστή κατά τη μεταγλώττιση.

Η ιεραρχική αυτή σχέση, καθώς και η θέση του δικού μας περάσματος εντός της αλυσίδας εκτέλεσης, απεικονίζεται στο Σχήμα 2.4.



Σχημα 2.4: Ιεραρχική δομή και ένθεση (nesting) των διαχειριστών στον New Pass Manager. Το PatternSelectPass εκτελείται ως μέρος του Function Pass Pipeline.

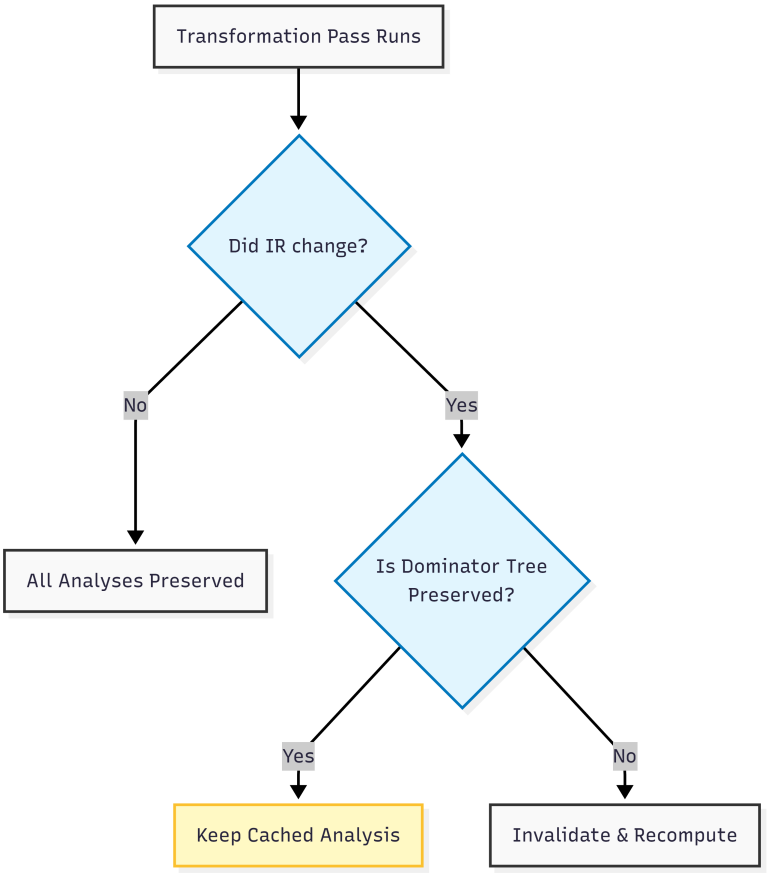
2.4.2 Ανάλυση και Ακύρωση (Analysis & Invalidation)

Μια κρίσιμη καινοτομία του NPM είναι ο ρητός διαχωρισμός μεταξύ Ανάλυσης και Μετασχηματισμού.

- Τα **Περάσματα Ανάλυσης** (π.χ. Dominator Tree Analysis) παράγουν δεδομένα χωρίς να αλλάζουν την IR. Τα αποτελέσματά τους αποθηκεύονται προσωρινά (cached).
- Τα **Περάσματα Μετασχηματισμού** (π.χ. InstCombine) τροποποιούν την IR. Μετά την εκτέλεσή τους, πρέπει να αναφέρουν ποιες αναλύσεις έχουν διατηρήσει έγκυρες (Preserved Analyses).

Εάν ένα πέρασμα τροποποιήσει τον κώδικα αλλά δηλώσει ότι διατήρησε το Dominator Tree, ο Pass Manager δεν χρειάζεται να το ξανα-υπολογίσει για το επόμενο πέρασμα, επιτυγχάνοντας σημαντική μείωση του χρόνου μεταγλώττισης.

Η λογική ροή αποφάσεων του Pass Manager σχετικά με το πότε πρέπει να ακυρώσει ή να διατηρήσει τα αποτελέσματα μιας ανάλυσης παρουσιάζεται στο Σχήμα 2.5.



Σχημα 2.5: Ο μηχανισμός διατήρησης (preservation) ή ακύρωσης (invalidation) των αποτελεσμάτων ανάλυσης στον NPM μετά από ένα πέρασμα μετασχηματισμού.

2.5 Επιλογή Εντολών και Κανονικοποίηση

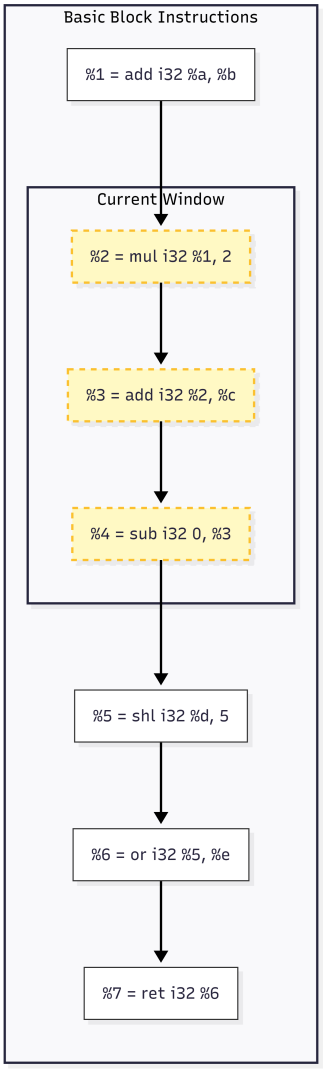
Ο πρωταρχικός στόχος του βελτιστοποιητή middle-end είναι η **Κανονικοποίηση** (Canonicalization). Αυτή η διαδικασία μετασχηματίζει τον κώδικα σε μια τυποποιημένη, προβλέψιμη μορφή. Σκοπός είναι να μειωθεί ο χώρος αναζήτησης για τα επόμενα περάσματα: αντί να πρέπει ένας αλγόριθμος να αναγνωρίζει 10 διαφορετικούς τρόπους γραφής του ίδιου μαθηματικού τύπου, ο κανονικοποιητής φροντίζει να υπάρχει μόνο ένας ¨πρότυπος' τρόπος.

Για παράδειγμα, η έκφραση $x > 5$ μπορεί να μετατραπεί αυτόματα σε $5 < x$. Αυτή η συνέπεια επιτρέπει στους υπόλοιπους αλγορίθμους βελτιστοποίησης να είναι απλούστεροι και ταχύτεροι.

2.5.1 InstCombine: Ο Αλγόριθμος Worklist

Το πέρασμα **Instruction Combining (instcombine)** είναι ο κύριος βελτιστοποιητής peephole του LLVM. Λειτουργεί εξετάζοντας ένα μικρό ¨παράθυρο' εντολών κάθε φορά, προσπαθώντας να τις απλοποιήσει τοπικά.

Η έννοια του ¨παραθύρου' βελτιστοποίησης, όπου εξετάζεται ένα υποσύνολο γειτονικών εντολών (π.χ. πολλαπλασιασμός που ακολουθείται από πρόσθεση), φαίνεται στο Σχήμα 2.6.



Σχημα 2.6: Απεικόνιση της λογικής peephole βελτιστοποίησης. Ο μεταγλωττιστής εστιάζει (κίτρινο πλαίσιο) σε ένα περιορισμένο παράθυρο διαδοχικών εντολών για να εντοπίσει ευκαιρίες τοπικής απλοποίησης.

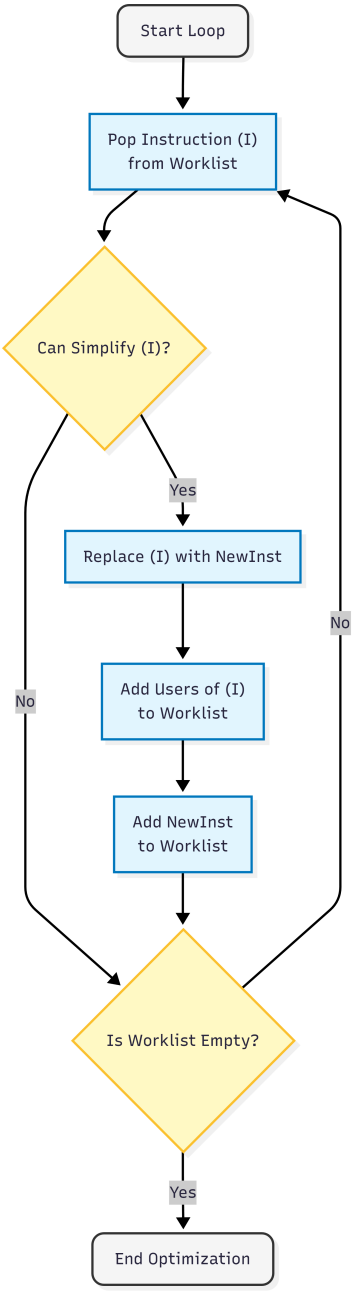
Η καρδιά του instcombine είναι ο **Αλγόριθμος Worklist**, ο οποίος λειτουργεί ως εξής:

1. Όλες οι εντολές της συνάρτησης μπαίνουν αρχικά σε μια λίστα εργασίας (Worklist).

2. Ο αλγόριθμος αφαιρεί μία εντολή και ελέγχει αν μπορεί να απλοποιηθεί (π.χ. μετατροπή του $x + 0$ σε x, ή του $(x * 2)$ σε $x \ll 1$).

3. Εάν γίνει απλοποίηση, η παλιά εντολή διαγράφεται και η νέα εντολή, καθώς και όλοι οι χρήστες της (users), προστίθενται ξανά στη Worklist για επανεξέταση.

Η επαναληπτική ροή αυτού του αλγορίθμου παρουσιάζεται αναλυτικά στο διάγραμμα ροής του Σχήματος 2.7.



Σχημα 2.7: Η ροή εκτέλεσης του αλγορίθμου Worklist στο InstCombine. Η διαδικασία επαναλαμβάνεται κυκλικά προσθέτοντας νέες εντολές στη λίστα, μέχρι να εξαντληθούν όλες οι πιθανές απλοποιήσεις.

Αυτή η διαδικασία συνεχίζεται μέχρι η Worklist να αδειάσει, διασφαλίζοντας ότι όλες οι αλυσιδωτές απλοποιήσεις έχουν εκτελεστεί. Ωστόσο, αν και ισχυρό, το instcombine εστιάζει στην απλοποίηση γενικού σκοπού. Συχνά αποτυγχάνει να συλλάβει πολύπλοκα ιδιώματα που απαιτούν την εξέταση πολλαπλών βασικών μπλοκ ή που δεν ταιριάζουν στους αυστηρούς κανόνες κανονικοποίησης.

2.5.2 Μέθοδοι Περιγραφής Προτύπων: TableGen vs C++

Ένα κρίσιμο ερώτημα στον σχεδιασμό ενός περάσματος βελτιστοποίησης είναι ο τρόπος περιγραφής των μοτίβων προς αναζήτηση. Στο οικοσύστημα του LLVM, υπάρχουν δύο κύριες προσεγγίσεις: η δηλωτική (declarative) μέσω TableGen και η προγραμματιστική (imperative) μέσω C++.

2.5.2.1 Η Γλώσσα TableGen (.td)

Το TableGen είναι μια εξειδικευμένη γλώσσα περιγραφής τομέα (DSL) που χρησιμοποιείται κυρίως στο Backend. Επιτρέπει στους προγραμματιστές να ορίσουν τα χαρακτηριστικά των εντολών (π.χ. καταχωρητές, λανθάνων χρόνος) και τα μοτίβα αντιστοίχισης με συνοπτικό τρόπο.

Ένα παράδειγμα κανόνα TableGen για την εντολή PopCount θα είχε την εξής μορφή:

```
def :  Pat<(ctpop i32:$src), (POPCNT32rr GR32:$src)>;
```

Αν και ισχυρό, το TableGen έχει περιορισμούς: είναι στατικό και δυσκολεύεται να περιγράψει πολύπλοκα αλγοριθμικά μοτίβα που εκτείνονται σε πολλαπλές εντολές με πολύπλοκες εξαρτήσεις, όπως ο αλγόριθμος SWAR.

2.5.2.2 Η Προσέγγιση C++ PatternMatch

Στην παρούσα διατριβή επιλέχθηκε η χρήση της βιβλιοθήκης `llvm/IR/PatternMatch.h` της C++. Αυτή η προσέγγιση προσφέρει:

- **Ευελιξία:** Μπορούμε να γράψουμε αυθαίρετο κώδικα C++ για να ελέγξουμε συνθήκες που δεν περιγράφονται εύκολα δηλωτικά.
- **Αναδρομικότητα:** Η δυνατότητα αναδρομικής περιήγησης στο δέντρο εντολών είναι πολύ πιο φυσική στη C++.
- **Ταχύτητα Ανάπτυξης:** Δεν απαιτείται η επαναμεταγλώττιση των πινάκων του TableGen, επιτρέποντας την ανάπτυξη του περάσματος ως πρόσθετο (plugin).

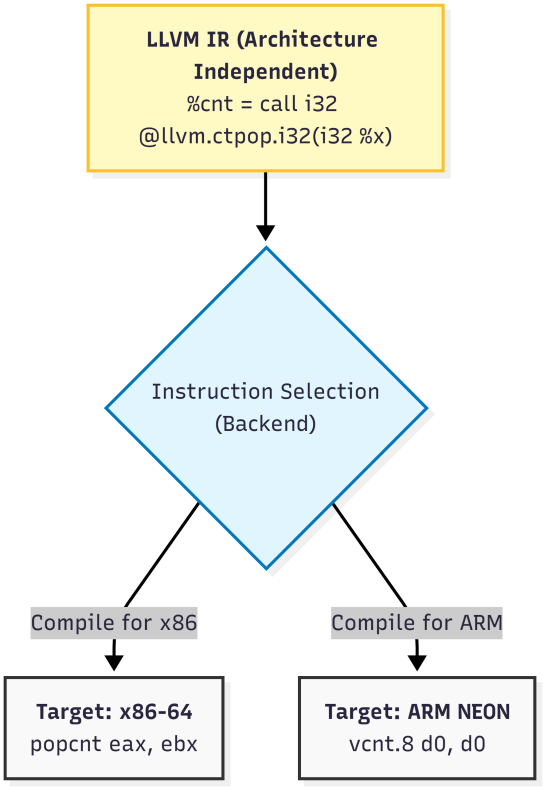
2.5.3 Εγγενείς Συναρτήσεις LLVM (Intrinsics)

Για να γεφυρώσει το χάσμα μεταξύ της γενικής IR και των εξειδικευμένων δυνατοτήτων του υλικού, το LLVM παρέχει τις **Εγγενείς Συναρτήσεις (Intrinsics)**.

Πρόκειται για ειδικές συναρτήσεις με το πρόθεμα `@llvm.`, οι οποίες έχουν την εξής διττή φύση:

- **Στο IR:** Συμπεριφέρονται ως κλήσεις συναρτήσεων, διατηρώντας την υψηλότερη σημασιολογία της πράξης (π.χ. `@llvm.ctpop` για την καταμέτρηση άσσων).
- **Στο Backend:** Κατά τη φάση της Επιλογής Εντολών (Instruction Selection), δεν μεταγλωττίζονται ως κλήσεις υπορουτίνας (call), αλλά αντικαθίστανται άμεσα (ὑποβιβάζονται' - lowered) από την αντίστοιχη εντολή μηχανής (π.χ. `POPCNT` σε x86 ή `VCNT` σε ARM NEON).

Η διαδικασία υποβιβασμού μιας εγγενούς συνάρτησης σε διαφορετικές αρχιτεκτονικές στόχους απεικονίζεται στο Σχήμα 2.8.



Σχημα 2.8: Ο υποβιβασμός (Lowering) μιας εγγενούς συνάρτησης (intrinsic) σε διαφορετικές αρχιτεκτονικές υλικού κατά τη φάση της επιλογής εντολών.

Ο εντοπισμός πολύπλοκων λογικών μοτίβων και η αντικατάστασή τους με intrinsics — όπως πραγματοποιείται σε αυτή τη διατριβή — είναι ένα κρίσιμο βήμα. Επιτρέπει στον μεταγλωττιστή να εκμεταλλευτεί προηγμένα χαρακτηριστικά του επεξεργαστή που διαφορετικά θα παρέμεναν ανεκμετάλλευτα πίσω από γενικές ακολουθίες κώδικα.

2.6 Διαδικασία Επιλογής Εντολών (Instruction Selection)

Η διαδικασία της Επιλογής Εντολών (Instruction Selection) αποτελεί τον συνδετικό κρίκο μεταξύ της ανεξάρτητης από το υλικό Ενδιάμεσης Αναπαράστασης (IR) και του κώδικα μηχανής για τη συγκεκριμένη αρχιτεκτονική-στόχο (target ISA, π.χ. x86, ARM). Στο LLVM, αυτή η διαδικασία υλοποιείται κυρίως μέσω του **SelectionDAG** και, πιο πρόσφατα, του **GlobalISel**.

Η σημασία αυτής της ενότητας για την παρούσα διατριβή έγκειται στο γεγονός ότι τα intrinsics που εισάγει το PatternSelectPass (όπως το `@llvm.ctpop`) λειτουργούν ως «συντομεύσεις» σε αυτή τη διαδικασία, απλοποιώντας δραματικά τον γράφο επιλογής.

2.6.1 Ο Γράφος SelectionDAG

Ο παραδοσιακός μηχανισμός επιλογής εντολών του LLVM βασίζεται στον SelectionDAG (Κατευθυνόμενος Ακυκλικός Γράφος Επιλογής). Η διαδικασία μετατροπής της IR σε εντολές μηχανής περιλαμβάνει τα εξής στάδια:

1. **Κατασκευή (Build DAG):** Κάθε βασικό μπλοκ μετατρέπεται σε έναν γράφο όπου οι κόμβοι αναπαριστούν πράξεις (π.χ. ADD, MUL) και οι ακμές τη ροή δεδομένων.
2. **Νομιμοποίηση Τύπων (Type Legalization):** Ο μεταγλωττιστής ελέγχει αν οι τύποι δεδομένων υποστηρίζονται από το υλικό. Για παράδειγμα, αν η αρχιτεκτονική είναι 32-bit και η πράξη αφορά 64-bit ακέραιο, ο κόμβος πρέπει να διασπαστεί σε δύο μικρότερες πράξεις.
3. **Νομιμοποίηση Πράξεων (Operation Legalization):** Ελέγχεται αν η πράξη υποστηρίζεται. Εάν η αρχιτεκτονική δεν διαθέτει εντολή για «υπόλοιπο διαίρεσης» (remainder), η πράξη μετατρέπεται σε ακολουθία πολλαπλασιασμών και αφαιρέσεων.
4. **Επιλογή Εντολών (Instruction Selection):** Το LLVM προσπαθεί να ταιριάξει μοτίβα του γράφου με εντολές μηχανής.

Εδώ φαίνεται η αξία της βελτιστοποίησης που προτείνουμε: Ένας αλγόριθμος όπως ο SWAR PopCount στο επίπεδο του SelectionDAG θα παρήγαγε δεκάδες κόμβους (AND, SHIFT,

ADD). Αντίθετα, η αντικατάστασή του με το intrinsic `llvm.ctpop` παράγει έναν και μοναδικό κόμβο `ISD::CTPOP`, ο οποίος αντιστοιχίζεται άμεσα (1-προς-1) στην εντολή `POPCNT` του επεξεργαστή, μειώνοντας τον χρόνο μεταγλώττισης και την πολυπλοκότητα του γράφου.

2.6.2 Το Πλαίσιο GlobalISel

Για την πληρότητα της ανάλυσης, πρέπει να αναφερθεί το GlobalISel, η νέα γενιά του συστήματος επιλογής εντολών. Σε αντίθεση με το SelectionDAG που εξετάζει μόνο ένα βασικό μπλοκ τη φορά, το GlobalISel έχει πρόσβαση σε ολόκληρη τη συνάρτηση, επιτρέποντας βελτιστοποιήσεις που διαπερνούν τα όρια των μπλοκ. Το PatternSelectPass είναι συμβατό και με τα δύο συστήματα, καθώς λειτουργεί σε επίπεδο IR, πριν την είσοδο στο backend.

2.7 Μοντέλα Κόστους και Διανυσματοποίηση

2.7.1 Ανάλυση Κόστους (Target Transform Info)

Κάθε βελτιστοποίηση πρέπει να απαντά στο ερώτημα: «Είναι αυτή η αλλαγή επωφελής;». Στο LLVM, η απάντηση δίνεται από το υποσύστημα **Target Transform Info (TTI)**. Το TTI εκτιμά το κόστος των εντολών IR σε σχέση με την αρχιτεκτονική-στόχο. Για παράδειγμα, το TTI μπορεί να ορίσει ότι ένας πολλαπλασιασμός έχει κόστος 4 κύκλους, ενώ η ολίσθηση (shift) έχει κόστος 1, καθοδηγώντας τον βελτιστοποιητή να επιλέξει τη φθηνότερη λύση.

2.7.2 Αυτόματη Διανυσματοποίηση (Auto-Vectorization)

Ένας από τους σημαντικότερους λόγους για τη χρήση intrinsics (όπως τα Min/Max) είναι η ενεργοποίηση της αυτόματης διανυσματοποίησης. Οι σύγχρονοι επεξεργαστές διαθέτουν μονάδες **SIMD (Single Instruction, Multiple Data)** που επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα. Ο Loop Vectorizer του LLVM δυσκολεύεται να διανυσματοποιήσει κώδικα με ροή ελέγχου (`if-else`). Η μετατροπή σε intrinsics εξαλείφει τις διακλαδώσεις, επιτρέποντας τη χρήση εντολών όπως η `PMAXSD` (AVX2), επιτυγχάνοντας πολλαπλάσια ταχύτητα εκτέλεσης.

Κεφάλαιο 3

Σχεδιασμός και Υλοποίηση Συστήματος

3.1 Αρχιτεκτονική Συστήματος

Το προτεινόμενο πλαίσιο βελτιστοποίησης, με την ονομασία `PatternSelectPass`, έχει σχεδιαστεί ως ένα αυτόνομο **Function Pass** (Πέρασμα Συνάρτησης) εντός της υποδομής LLVM. Σε αντίθεση με τα Module passes που επεξεργάζονται ολόκληρο το πρόγραμμα ταυτόχρονα και απαιτούν περισσότερη μνήμη, ένα Function Pass λειτουργεί σε μία συνάρτηση τη φορά. Αυτή η σχεδιαστική επιλογή διασφαλίζει την αρθρωτότητα (modularity) και επιτρέπει στο πέρασμα να ενσωματωθεί εύκολα στην τυπική διοχέτευση βελτιστοποίησης (π.χ. `-O2` ή `-O3`), εκμεταλλευόμενο την παράλληλη επεξεργασία συναρτήσεων που προσφέρει ο Pass Manager.

3.1.1 Ενσωμάτωση με τον Νέο Διαχειριστή Περασμάτων (New Pass Manager)

Το πέρασμα υλοποιείται ως ένα δυναμικά φορτωμένο πρόσθετο (plugin/shared object). Χρησιμοποιεί την αρχιτεκτονική του **LLVM New Pass Manager (NPM)**, η οποία βασίζεται σε έναν μηχανισμό ανάκλησης (callback) για την εγγραφή.

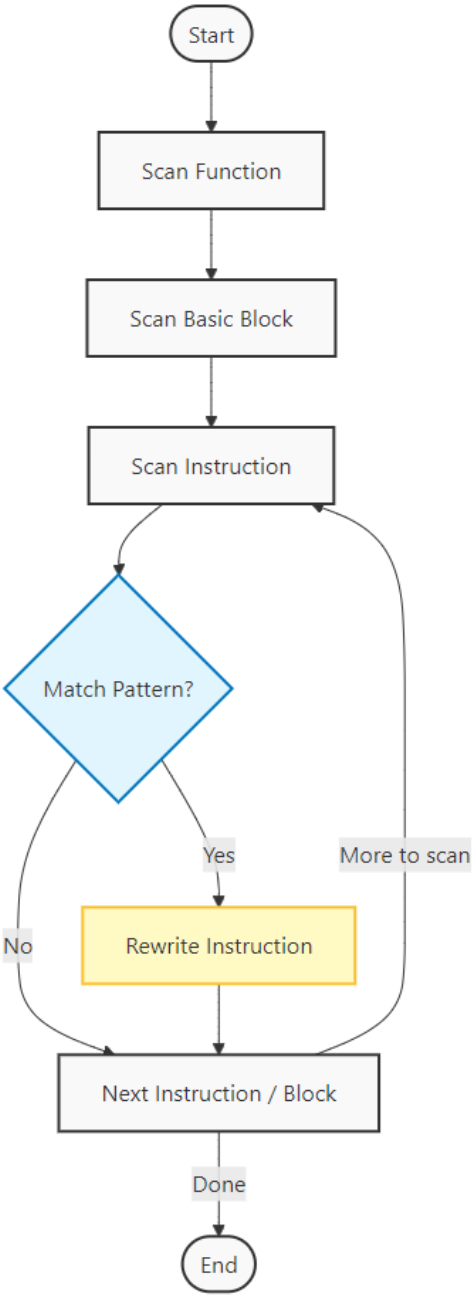
Το σημείο εισόδου του προσθέτου, `llvmGetPassPluginInfo`, καταχωρεί ένα `PipelineParsingCallback`. Αυτό επιτρέπει στον χρήστη να καλέσει το πέρασμα μέσω του εργαλείου `opt` χρησιμοποιώντας το όρισμα γραμμής εντολών:

```
-passes="pattern-select"
```

Αυτή η προσέγγιση αποσυνδέει την υλοποίηση του περάσματος από τον πυρήνα του LLVM, επιτρέποντας γρήγορους κύκλους ανάπτυξης και μεταγλώττισης (out-of-tree development).

3.1.2 Ροή Εργασίας (Workflow)

Η ροή εκτέλεσης του περάσματος ακολουθεί μια στρατηγική ιεραρχικής διάσχισης, η οποία απεικονίζεται στο παρακάτω διάγραμμα ροής:



Σχημα 3.1: Διάγραμμα ροής της διαδικασίας σάρωσης και μετασχηματισμού του PatternSelectPass.

1. **Διάσχιση Συνάρτησης (Function Traversal):** Το πέρασμα σαρώνει κάθε Βασικό Μπλοκ (Basic Block) στη συνάρτηση-στόχο.
2. **Σάρωση Εντολών (Instruction Scan):** Εντός κάθε μπλοκ, επαναλαμβάνει τη διαδικασία για τις εντολές. Διασφαλίζεται η ασφαλής επανάληψη (χρήση `make_early_inc_range`) ώστε να επιτρέπεται η αντικατάσταση ή η διαγραφή εντολών χωρίς την ακύρωση των επαναληπτών (iterators).
3. **Δρομολόγηση Μοτίβων (Pattern Dispatch):** Κάθε εντολή αποτελεί τη ρίζα ενός πιθανού "Δέντρου Έκφρασης". Η συνάρτηση δρομολόγησης (`tryMatchAndRewrite`) ελέγχει την εντολή έναντι των υποστηριζόμενων matchers.
4. **Επανεγγραφή (Rewriting):** Εάν ταυτοποιηθεί ένα μοτίβο, ενεργοποιείται ο αντίστοιχος rewriter για τη δημιουργία της βελτιστοποιημένης IR (συνήθως μιας κλήσης intrinsic) και την αντικατάσταση των αρχικών εντολών.

3.1.3 Ανάπτυξη Εκτός Δέντρου (Out-of-Tree Development)

Το `PatternSelectPass` αναπτύχθηκε ως ένα **out-of-tree** πρόσθετο (plugin). Η προσέγγιση αυτή επιλέχθηκε έναντι της ενσωμάτωσης στον πηγαίο κώδικα του LLVM (in-tree) για τους εξής λόγους:

- **Αρθρωτότητα (Modularity):** Ο κώδικας της βελτιστοποίησης παραμένει απομονωμένος, διευκολύνοντας τη συντήρηση και τη μεταφορικότητα.
- **Ταχύτητα Μεταγλώττισης:** Επιτρέπει την ανακατασκευή μόνο του περάσματος χωρίς την ανάγκη επαναμεταγλώττισης ολόκληρης της υποδομής LLVM.
- **Δυναμική Σύνδεση:** Το πέρασμα μεταγλωττίζεται σε κοινόχρηστη βιβλιοθήκη (shared object), η οποία φορτώνεται δυναμικά από τον Pass Manager κατά την εκτέλεση.

Η διαδικασία κατασκευής βασίζεται στο εργαλείο `CMake`, το οποίο διασυνδέει το πέρασμα με τις απαραίτητες κεφαλίδες (headers) και βιβλιοθήκες του συστήματος, διασφαλίζοντας τη συμβατότητα με την έκδοση του LLVM που είναι εγκατεστημένη στο περιβάλλον εργασίας.

3.1.4 Υλοποίηση του Σημείου Εισόδου (Pass Entry Point)

Η καρδιά της υλοποίησης βρίσκεται στο αρχείο `PatternSelectPass.cpp`, το οποίο λειτουργεί ως ο ενορχηστρωτής (orchestrator) της βελτιστοποίησης. Η κλάση `PatternSelectPass`

κληρονομεί από το πρότυπο `PassInfoMixin`, μια τεχνική CRTP (Curiously Recurring Template Pattern) που χρησιμοποιεί ο LLVM για να παρέχει αυτόματα τις απαραίτητες μεθόδους αναγνώρισης του περάσματος χωρίς τη χρήση εικονικών συναρτήσεων (virtual functions), μειώνοντας έτσι το κόστος εκτέλεσης.

3.1.4.1 Η Μέθοδος run()

Η μέθοδος `run()` αποτελεί τον κύριο βρόχο εκτέλεσης. Δέχεται ως όρισμα τη Συνάρτηση (Function) και τον Διαχειριστή Ανάλυσης. Η στρατηγική διάσχισης που ακολουθείται είναι κρίσιμη για την ευστάθεια του μεταγλωττιστή:

```
PreservedAnalyses run(Function &F, FunctionAnalysisManager &) {
    bool changed = false;
    for (auto &BB : F) {
        // Use a safe iterator to allow instruction removal/
    modification during iteration
        for (auto It = BB.begin(), End = BB.end(); It != End; ) {
            Instruction *I = &*It++;

            // Call the matching and rewriting logic (defined in
    PatternRules.cpp)
            if (tryMatchAndRewrite(*I))
                changed = true;
        }
    }

    // Notify the PassManager whether existing analyses should be
    invalidated
    return changed ? PreservedAnalyses::none() : PreservedAnalyses
    ::all();
}
```

Listing 3.1: Ο βρόχος εκτέλεσης του PatternSelectPass

Ιδιαίτερη προσοχή δόθηκε στη διαχείριση του iterator. Η εντολή `Instruction *I = &*It++;` διασφαλίζει ότι ο δείκτης προχωρά στην επόμενη εντολή *πριν* επεξεργαστούμε την τρέχουσα. Αυτό είναι απαραίτητο διότι, εάν η συνάρτηση `tryMatchAndRewrite` αποφασίσει να διαγράψει την τρέχουσα εντολή (π.χ. επειδή αντικαταστάθηκε από ένα intrinsic), ένας απλός iterator θα καθίστατο άκυρος (invalidated), οδηγώντας σε κατάρρευση (crash) του μεταγλωττιστή.

3.1.4.2 Διατήρηση Αναλύσεων (Preserved Analyses)

Στο τέλος της εκτέλεσης, το πέρασμα πρέπει να ενημερώσει τον Pass Manager για την κατάσταση του κώδικα. Εάν πραγματοποιηθεί οποιαδήποτε αλλαγή (`changed = true`), επιστρέφουμε `PreservedAnalyses::none()`, σηματοδοτώντας ότι προηγούμενες αναλύσεις (όπως το Dominator Tree) ενδέχεται να μην ισχύουν πλέον και πρέπει να υπολογιστούν εκ νέου.

3.2 Αναλυτική Παρουσίαση και Τεχνική Υλοποίηση Ι-διωμάτων

Σε αυτή την ενότητα, πραγματοποιείται μια διεξοδική ανάλυση των επτά αριθμητικών και λογικών ιδιωμάτων που αποτελούν τον στόχο του `PatternSelectPass`. Η ανάλυση ακολουθεί τη διαδρομή του κώδικα από την υψηλού επιπέδου αφαίρεση της γλώσσας C στην Ενδιάμεση Αναπαράσταση (Intermediate Representation - IR) και τελικά στην παραγωγή βέλτιστων εντολών μηχανής.

3.2.1 Πολλαπλασιασμός με Δυνάμεις του Δύο (Strength Reduction)

3.2.1.1 Περιγραφή Προβλήματος

Ο πολλαπλασιασμός ακεραίων, αν και θεμελιώδης, παραμένει μια δαπανηρή πράξη για την ALU. Ενώ μια πρόσθεση ή μια λογική πράξη εκτελείται συνήθως σε έναν κύκλο ρολογιού, ο πολλαπλασιασμός μπορεί να απαιτήσει 3 έως 4 κύκλους (latency) λόγω της πολυπλοκότητας του κυκλώματος (π.χ. δέντρα Wallace ή πολλαπλασιαστές Booth).

Οι προγραμματιστές συχνά χρησιμοποιούν πολλαπλασιασμούς με σταθερές που είναι δυνάμεις του δύο $(2, 4, 8, \dots)$ για λόγους σαφήνειας στον κώδικα (π.χ. υπολογισμός offsets ή μεγεθών μνήμης). Το «σημασιολογικό χάσμα» (semantic gap) εμφανίζεται όταν ο μεταγλωττιστής αντιμετωπίζει αυτές τις πράξεις ως γενικούς πολλαπλασιασμούς, χάνοντας την ευκαιρία για «μείωση ισχύος» (strength reduction). Η εκτέλεση ενός γενικού πολλαπλασιασμού για μια τόσο ειδική περίπτωση αποτελεί σπατάλη πόρων, τόσο σε χρόνο εκτέλεσης όσο και σε κατανάλωση ενέργειας.

3.2.1.2 Θεωρητική και Υλική Τεκμηρίωση της Βελτιστοποίησης

Η επιλογή της αντικατάστασης του πολλαπλασιασμού με ολίσθηση δεν βασίζεται μόνο σε εμπειρικούς κανόνες, αλλά θεμελιώνεται τόσο στη μαθηματική ισοδυναμία των πράξεων στο δυαδικό σύστημα, όσο και στην αρχιτεκτονική υπεροχή των κυκλωμάτων ολίσθησης έναντι των πολλαπλασιαστών.

Α. Επαλήθευση μέσω Δυαδικής Αναπαράστασης Η ισοδυναμία του πολλαπλασιασμού με την ολίσθηση προκύπτει από τη φύση του δυαδικού συστήματος θέσης. Όπως στο δεκαδικό σύστημα ο πολλαπλασιασμός με το 10^k ισοδυναμεί με την προσθήκη k μηδενικών στο τέλος του αριθμού (π.χ. $5 \times 10 = 50$), έτσι και στο δυαδικό σύστημα, ο πολλαπλασιασμός με το 2^k ισοδυναμεί με την ολίσθηση των ψηφίων κατά k θέσεις αριστερά.

Ας θεωρήσουμε το παράδειγμα $x \times 8$, όπου $x = 3$ (δυαδικό `0000 0011`) και $8 = 2^3$. Η πράξη αυτή αντιστοιχεί σε αριστερή ολίσθηση κατά 3 θέσεις ($k = 3$).

Πράξη	**Δυαδική Μορφή**	**Δεκαδική Τιμή**	**Παρατήρηση**
Αρχική Τιμή (x)	`0000 0011`	3	-
Πολλαπλασιασμός ($x \times 8$)	`0001 1000`	24	$3 \times 8 = 24$
Ολίσθηση ($x \ll 3$)	`0001 1000`	24	Μετακίνηση 3 θέσεων

ΠΙΝΑΚΑΣ 3.1: Οπτική επαλήθευση της ισοδυναμίας για $x = 3$ και $k = 3$. Παρατηρούμε ότι το μοτίβο των bits διατηρείται ακέραιο και απλώς μετατοπίζεται.

Β. Ανάλυση Πολυπλοκότητας Κυκλώματος (Hardware Latency) Η κύρια αιτία για την οποία αυτή η βελτιστοποίηση είναι «επιθυμητή» (desirable) έγκειται στον τρόπο που υλοποιούνται οι εντολές στο πυρίτιο (silicon level).

- **Πολλαπλασιαστής (Hardware Multiplier):** Η υλοποίηση του πολλαπλασιασμού σε επίπεδο πυλών είναι μια εξαιρετικά δαπανηρή διαδικασία. Απαιτεί σύνθετα κυκλώματα, όπως δέντρα Wallace (Wallace Trees) ή κωδικοποίηση Booth, για να αθροίσουν τα μερικά γινόμενα. Λόγω του βάθους της λογικής (logic depth), το σήμα χρειάζεται περισσότερο χρόνο για να διαδοθεί. Σε σύγχρονους επεξεργαστές υψηλών επιδόσεων, η εντολή `mul` έχει τυπικό λανθάνοντα χρόνο (latency) **3 έως 4 κύκλων ρολογιού**.
- **Ολισθητής (Barrel Shifter):** Αντίθετα, η ολίσθηση υλοποιείται μέσω ενός κυκλώματος γνωστού ως Barrel Shifter. Πρόκειται για ένα καθαρά συνδυαστικό κύκλωμα που χρησιμοποιεί έναν πίνακα από πολυπλέκτες (multiplexers). Το μοναδικό χαρακτηριστικό του Barrel Shifter είναι ότι μπορεί να ολισθήσει έναν αριθμό κατά οσαδήποτε bits (από 0 έως 63) σε **έναν μόνο κύκλο ρολογιού** ($O(1)$).

Συμπερασματικά, η αντικατάσταση του `mul` με `shl` επιτυγχάνει:

1. **Μείωση του Latency:** Από 3-4 κύκλους σε 1 κύκλο (βελτίωση έως 75%).
2. **Μείωση Κατανάλωσης Ενέργειας:** Ο πολλαπλασιαστής ενεργοποιεί χιλιάδες πύλες για τον υπολογισμό, ενώ ο ολισθητής είναι ένα απλούστερο δίκτυο δρομολόγησης, μειώνοντας τη δυναμική κατανάλωση ισχύος (dynamic power consumption).

3.2.1.3 Μη βελτιστοποιημένος Κώδικας

Στην τυπική μορφή του, ο κώδικας παράγει μια εντολή `mul` η οποία δεσμεύει την αριθμητική μονάδα πολλαπλασιασμού χωρίς λόγο.

```
int multiply_by_8(int x) {
    return x * 8; // Multiplied by a power of 2
}
```

Listing 3.2: Πολλαπλασιασμός σε C

```
%res = mul i32 %x, 8
```

Listing 3.3: Μη βελτιστοποιημένη LLVM IR

3.2.1.4 Ανάλυση Matcher

Ο matcher πρέπει να εντοπίσει τη δυαδική πράξη και να επικεντρωθεί στη φύση της σταθεράς.

```
Value *X; ConstantInt *CI;
// Match the multiplication instruction with a constant
if (match(Instr, m_Mul(m_Value(X), m_ConstantInt(CI)))) {
    const APInt &Val = CI->getValue();
    // Check if the constant is a power of 2
    if (Val.isPowerOf2()) {
        return rewriteMultiplication(Instr, X, Val);
    }
}
```

Listing 3.4: Κώδικας Matcher για Πολλαπλασιασμό

Επεξήγηση κώδικα ανά γραμμή:

- `m_Mul`: Αναζητά στο instruction stream για εντολές με opcode `Instruction::Mul`.

- `m_Value(X)`: Δεσμεύει τον πρώτο τελεστέο (την μεταβλητή) στον δείκτη `X`.
- `m_ConstantInt(CI)`: Διασφαλίζει ότι ο δεύτερος τελεστέος είναι ακέραια σταθερά και την αποθηκεύει στο `CI`.
- `Val.isPowerOf2()`: Χρησιμοποιεί την κλάση `APInt` του LLVM για να ελέγξει αν η δυαδική αναπαράσταση της σταθεράς περιέχει ακριβώς έναν άσσο.

3.2.1.5 Ανάλυση Rewriter

Ο rewriter αναλαμβάνει τη μετατροπή της σημασιολογίας από πολλαπλασιασμό σε ολίσθηση.

```
// Calculate the log2 of the constant to get shift amount
uint32_t ShiftAmt = Val.logBase2();
Value *ShlAmtVal = ConstantInt::get(CI->getType(), ShiftAmt);
// Create the new SHL instruction
Value *NewInst = Builder.CreateShl(X, ShlAmtVal, "mul_to_shl");
// Update the Def-Use chain
Instr->replaceAllUsesWith(NewInst);
```

Listing 3.5: Κώδικας Rewriter για Πολλαπλασιασμό

Επεξήγηση βημάτων:

- `Val.logBase2()`: Υπολογίζει τον εκθέτη. Για τη σταθερά 8, επιστρέφει 3.
- `Builder.CreateShl`: Εισάγει τη νέα εντολή `shl` στην τρέχουσα θέση του IR.
- `replaceAllUsesWith`: Αυτό είναι το κρισιμότερο βήμα. Ενημερώνει τη Def-Use αλυσίδα του LLVM, ώστε όλες οι επόμενες εντολές που περίμεναν το αποτέλεσμα του πολλαπλασιασμού να λαμβάνουν πλέον το αποτέλεσμα της ολίσθησης.

3.2.1.6 Βελτιστοποιημένη IR και Target Assembly

Μετά την παρέμβαση του περάσματος, ο κώδικας είναι πλέον έτοιμος για το instruction selection του backend.

```
%mul_to_shl = shl i32 %x, 3
```

Listing 3.6: Βελτιστοποιημένη LLVM IR

Στην αρχιτεκτονική x86-64, ο υποβιβασμός παράγει την εντολή `shll`, η οποία έχει latency μόλις 1 κύκλο.

```
1 shll $3, %eax
```

Listing 3.7: x86 Assembly Output

3.2.2 Περιστροφή Bit (Bitwise Rotate)

3.2.2.1 Περιγραφή Προβλήματος

Η περιστροφή bit είναι κρίσιμη για την ασφάλεια πληροφοριών, ωστόσο η γλώσσα C στερείται ενός τελεστή ’’’. Αυτό αναγκάζει τους προγραμματιστές να χρησιμοποιούν το ιδίωμα `(x « n) | (x » (width - n))`. Αν και για τον άνθρωπο η πρόθεση είναι σαφής, για τον μεταγλωττιστή αυτό εμφανίζεται ως ένας γράφος τριών ή τεσσάρων ανεξάρτητων πράξεων, γεγονός που συχνά οδηγεί σε αναποτελεσματική παραγωγή κώδικα με πολλαπλές ολισθήσεις και εντολές `OR`.

3.2.2.2 Θεωρητική και Υλική Τεκμηρίωση της Βελτιστοποίησης

Η αντικατάσταση της πράξης του υπολοίπου (`urem`) με τη λογική σύζευξη (`and`) όταν ο διαιρέτης είναι δύναμη του δύο, αποτελεί μια από τις πιο σημαντικές βελτιστοποιήσεις απόδοσης. Η εγκυρότητά της βασίζεται στις ιδιότητες της αριθμητικής υπολοίπων (modular arithmetic) στο δυαδικό σύστημα, ενώ η αναγκαιότητά της πηγάζει από το υψηλό υπολογιστικό κόστος της διαίρεσης στο υλικό.

Α. Επαλήθευση μέσω Δυαδικής Αναπαράστασης Στο δεκαδικό σύστημα, για να βρούμε το υπόλοιπο της διαίρεσης ενός αριθμού με το 10^k, αρκεί να κρατήσουμε τα τελευταία k ψηφία (π.χ. $12345 \pmod{100} = 45$). Αντίστοιχα, στο δυαδικό σύστημα, το υπόλοιπο της διαίρεσης ενός αριθμού x με το 2^k ισούται με τα k λιγότερο σημαντικά bits (LSBs) του x.

Η απομόνωση αυτών των bits επιτυγχάνεται μέσω της λογικής πράξης AND με μια μάσκα που αποτελείται από k άσσους. Η μάσκα αυτή προκύπτει μαθηματικά ως $2^k - 1$.

Ας θεωρήσουμε το παράδειγμα $x\%16$, όπου $x = 53$ (δυαδικό `0011 0101`) και $16 = 2^4$. Η μάσκα είναι $16 - 1 = 15$ (δυαδικό `0000 1111`).

Πράξη	**Δυαδική Μορφή**	**Δεκαδική**	**Παρατήρηση**
Αρχική Τιμή (x)	`0011 0101`	53	-
Διαιρέτης (16)	`0001 0000`	16	2^4 ($k = 4$)
Μάσκα ($16 - 1$)	`0000 1111`	15	Τα κάτω 4 βιτς είναι 1
Αποτέλεσμα (x & 15)	`0000 0101`	**5**	$53 \pmod{16} = 5$

ΠΙΝΑΚΑΣ 3.2: Οπτική επαλήθευση της ισοδυναμίας για $x = 53$ και $C = 16$. Η μάσκα «κόβει» τα ανώτερα βιτς και διατηρεί μόνο το υπόλοιπο.

Β. Ανάλυση Πολυπλοκότητας Κυκλώματος (Hardware Latency) Η διαφορά απόδοσης σε αυτή την περίπτωση είναι ακόμα πιο δραματική σε σχέση με τον πολλαπλασιασμό, καθώς η διαίρεση είναι η πιο ακριβή αριθμητική πράξη σε έναν επεξεργαστή.

- **Διαιρέτης (Hardware Divider):** Η μονάδα διαίρεσης είναι εξαιρετικά αργή επειδή ο υπολογισμός είναι επαναληπτικός (ιτερατιε). Σε αντίθεση με την πρόσθεση ή τον πολλαπλασιασμό, τα ψηφία του αποτελέσματος εξαρτώνται από τα προηγούμενα με τρόπο που δύσκολα παραλληλίζεται. Σε αρχιτεκτονικές όπως η x86 (`div` ινστρυστιον), ο λανθάνων χρόνος μπορεί να κυμαίνεται από **20 έως 80 κύκλους ρολογιού**, ανάλογα με το μέγεθος των τελεστέων.
- **Λογική Μονάδα (ALU - AND):** Η εντολή `and` είναι μια απλή, bitwise πράξη που δεν έχει καμία μεταφορά κρατουμένου (carry propagation). Εκτελείται σε **1 κύκλο ρολογιού** (ή ακόμα και σε λιγότερο από έναν κύκλο σε υπερ-κλιμακωτούς επεξεργαστές μέσω throughput optimization).

Συμπερασματικά, η αντικατάσταση του `urem` με `and` προσφέρει:

1. **Τεράστια Μείωση του Latency:** Από 50 κύκλους σε 1 κύκλο (βελτίωση τάξης μεγέθους 50×).
2. **Απελευθέρωση Πόρων:** Η μονάδα διαίρεσης είναι σπάνιος πόρος (συνήθως υπάρχει μόνο μία ανά πυρήνα), ενώ μονάδες ALU για AND υπάρχουν πολλές. Η βελτιστοποίηση αποτρέπει το «μπουκάλι» (bottleneck) στο pipeline του επεξεργαστή.

3.2.2.3 Μη βελτιστοποιημένος Κώδικας

Ο Clang παράγει έναν δέντρο εντολών που περιλαμβάνει δύο ολισθήσεις και μία λογική σύζευξη.

```c
unsigned int rotl(unsigned int x, unsigned int n) {
    // Typical C implementation for bitwise rotation
    return (x << n) | (x >> (32 - n));
}
```

Listing 3.8: Rotate Left σε C

```
%1 = shl i32 %x, %n
%2 = sub i32 32, %n
%3 = lshr i32 %x, %2
%res = or i32 %1, %3
```

Listing 3.9: Μη βελτιστοποιημένη LLVM IR

3.2.2.4 Ανάλυση Matcher

Ο matcher πρέπει να αναγνωρίσει το «σχήμα» της περιστροφής, διασφαλίζοντας ότι η τιμή που ολισθαίνει είναι η ίδια και στις δύο πλευρές.

```c
Value *X, *N;
// Match the circular shift pattern: (x << n) | (x >> (width - n))
if (match(Instr, m_Or(
        m_Shl(m_Value(X), m_Value(N)),
        m_LShr(m_Deferred(X), m_Sub(m_SpecificInt(32), m_Deferred(
    N)))
    ))) {
    return rewriteRotate(Instr, X, N);
}
```

Listing 3.10: Κώδικας Matcher για Rotate

Ανάλυση λογικής:

- `m_Or`: Η ρίζα του δέντρου πρέπει να είναι η εντολή `OR`.
- `m_Deferred(X)`: Αυτή είναι μια προηγμένη τεχνική. Διασφαλίζει ότι η μεταβλητή στο δεξί σκέλος του `OR` είναι ακριβώς η ίδια SSA τιμή που βρέθηκε στο αριστερό σκέλος.
- `m_Sub(m_SpecificInt(32), ...)`: Επαληθεύει ότι η δεξιά ολίσθηση γίνεται κατά $32 - n$, γεγονός που συμπληρώνει την περιστροφή για έναν ακέραιο 32-bit.

3.2.2.5 Ανάλυση Rewriter

Αντικαθιστούμε ολόκληρο το δέντρο με το funnel shift intrinsic.

```
// Declare the funnel shift left intrinsic (@llvm.fshl)
Function *F = Intrinsic::getDeclaration(Mod, Intrinsic::fshl, {X->
    getType()});
// Create the intrinsic call representing rotation
Value *NewRotate = Builder.CreateCall(F, {X, X, N}, "
    rotl_intrinsic");
// Replace unoptimized instructions with the hardware-friendly
    call
Instr->replaceAllUsesWith(NewRotate);
```

Listing 3.11: Κώδικας Rewriter για Rotate

Γιατί αυτή η αλλαγή είναι σημαντική: Το `@llvm.fshl` είναι «πρώτης τάξεως» πολίτης στο LLVM. Ενημερώνει το backend ότι πρόκειται για περιστροφή, επιτρέποντας την απευθείας χρήση των flags του επεξεργαστή.

3.2.2.6 Βελτιστοποιημένη IR και Target Assembly

```
%rotl = call i32 @llvm.fshl.i32(i32 %x, i32 %x, i32 %n)
```

Listing 3.12: Βελτιστοποιημένη LLVM IR

Στην x86 αρχιτεκτονική, ολόκληρος ο γράφος των 4 εντολών συμπτύσσεται στην εντολή `roll`:

```
roll %cl, %eax
```

Listing 3.13: x86 Assembly Output

3.2.3 Καταμέτρηση Πληθυσμού (PopCount) - Αλγόριθμος SWAR

3.2.3.1 Περιγραφή Προβλήματος

Η καταμέτρηση των ενεργών bits (set bits), γνωστή και ως Hamming Weight, είναι θεμελιώδης σε εφαρμογές κρυπτογραφίας, διόρθωσης σφαλμάτων (Hamming distance) και ευρετηρίασης βάσεων δεδομένων.

Ο παραδοσιακός τρόπος υλοποίησης σε λογισμικό, όταν δεν υπάρχει υποστήριξη υλικού, είναι ο αλγόριθμος **SWAR (SIMD Within A Register)**. Ο αλγόριθμος αυτός αποτελεί

ένα «αριστούργημα» λογικής bitwise, καθώς εκτελεί παράλληλη άθροιση των bits μέσα στον ίδιο τον καταχωρητή 32 ή 64 bits, χρησιμοποιώντας μια στρατηγική «διαίρει και βασίλευε». Παρόλα αυτά, για έναν μεταγλωττιστή, ο SWAR εμφανίζεται ως ένας χαώδης, βαθύς γράφος εξαρτήσεων που καταναλώνει πολλούς καταχωρητές και κύκλους εκτέλεσης.

3.2.3.2 Προτεινόμενη Λύση

Η βέλτιστη λύση είναι η αντικατάσταση ολόκληρης της αλυσίδας εντολών του SWAR με την εγγενή συνάρτηση `@llvm.ctpop`. Αυτό επιτρέπει στο backend να χρησιμοποιήσει την ειδική εντολή `POPCNT` (σε x86) ή `VCNT` (σε ARM), οι οποίες εκτελούν τον υπολογισμό σε επίπεδο κυκλώματος.

3.2.3.3 Θεωρητική και Υλική Τεκμηρίωση της Βελτιστοποίησης

Η υπεροχή της χρήσης της εντολής `POPCNT` έναντι του αλγορίθμου SWAR τεκμηριώνεται τόσο αλγοριθμικά όσο και ενεργειακά.

Α. Αλγοριθμική Ανάλυση: Η Λογική του SWAR Για να κατανοήσουμε γιατί ο SWAR είναι βαρύς, πρέπει να δούμε πώς λειτουργεί. Ο αλγόριθμος αθροίζει τα bits σε γειτονικά ζεύγη, στη συνέχεια σε τετράδες (nibbles), σε bytes, και ούτω καθεξής.

Ένα τυπικό βήμα του SWAR για ομαδοποίηση ανά 2 bits είναι:

$$x = (x \,\&\, \texttt{0x55}\ldots) + ((x \gg 1) \,\&\, \texttt{0x55}\ldots) \tag{3.1}$$

Αυτό το μοτίβο επαναλαμβάνεται λογαριθμικά (για 32-bit αριθμούς απαιτούνται 5 βήματα). Αν και ο αλγόριθμος έχει πολυπλοκότητα $O(1)$ ως προς το μέγεθος της εισόδου (δεν έχει βρόχους), η «σταθερά» του είναι μεγάλη: απαιτεί περίπου **12-15 αριθμητικές και λογικές εντολές** (shifts, ands, adds, muls).

Β. Ανάλυση Πολυπλοκότητας Κυκλώματος (Hardware Efficiency) Η αντικατάσταση αυτών των 15 εντολών με μία εντολή υλικού (`POPCNT`) προσφέρει δραματική βελτίωση για τους εξής λόγους:

- **Μείωση Αλυσίδας Εξαρτήσεων (Dependency Chain):** Στον SWAR, κάθε βήμα εξαρτάται από το προηγούμενο (σειριακή εκτέλεση). Αυτό εμποδίζει την παραλληλία σε επίπεδο εντολών (ILP). Αντίθετα, η εντολή `POPCNT` υλοποιείται στο υλικό χρησιμοποιώντας ένα δέντρο αθροιστών (Counter Tree ή Compressor Tree), το οποίο υπολογίζει το αποτέλεσμα σχεδόν ακαριαία.

- **Throughput και Latency:**
  - **SWAR:** Συνολικός λανθάνων χρόνος περίπου **10-15 κύκλων ρολογιού** (άθροισμα των καθυστερήσεων των επιμέρους εντολών).
  - **POPCNT:** Σε σύγχρονους επεξεργαστές (π.χ. Intel Skylake και νεότεροι), η εντολή έχει latency **3 κύκλων** και throughput **1 εντολής ανά κύκλο**.
- **Εξοικονόμηση Πόρων Frontend:** Ο επεξεργαστής δεν χρειάζεται να ανακαλέσει (fetch), να αποκωδικοποιήσει (decode) και να δρομολογήσει 15 διαφορετικές εντολές. Μία μόνο εντολή περνάει από το pipeline, μειώνοντας την πίεση στον αποκωδικοποιητή και την κατανάλωση ενέργειας.

Χαρακτηριστικό	**Αλγόριθμος SWAR**	**Εντολή POPCNT**
Πλήθος Εντολών IR	∼15-20	1 (intrinsic)
Πλήθος Εντολών Assembly	∼15	1
Κύκλοι Ρολογιού (Latency)	∼12-15	3
Κατανάλωση Καταχωρητών	Υψηλή (προσωρινές τιμές)	Ελάχιστη

Πινακας 3.3: Συγκριτική ανάλυση απόδοσης μεταξύ της υλοποίησης λογισμικού (SWAR) και της εξειδικευμένης εντολής υλικού.

Συμπερασματικά, η αναγνώριση του SWAR και η αντικατάστασή του είναι ίσως η πιο αποδοτική «συμπίεση» κώδικα που μπορεί να επιτύχει το πέρασμά μας, καθώς μετατρέπει έναν ολόκληρο αλγόριθμο σε μία και μοναδική εντολή μηχανής.

3.2.3.4 Μη βελτιστοποιημένος Κώδικας

Ο αλγόριθμος χρησιμοποιεί μάσκες όπως 0ξ55555555 και 0ξ33333333 για να αθροίσει τους άσσους σε στάδια.

```
1 // Hacker's Delight SWAR implementation
2 x = x - ((x >> 1) & 0x55555555);
3 x = (x & 0x33333333) + ((x >> 2) & 0x33333333);
4 return (((x + (x >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
```

Listing 3.14: PopCount (SWAR) σε C

Στην LLVM IR, αυτός ο κώδικας μεταφράζεται σε πάνω από 15 εντολές, δημιουργώντας έναν πυκνό γράφο ροής δεδομένων.

3.2.3.5 Ανάλυση Matcher

Ο matcher για το PopCount είναι ο πλέον σύνθετος, καθώς πρέπει να αναγνωρίσει τις μαθηματικές σταθερές-κλειδιά.

```
Value *X;
// Match the complex bit manipulation chain found in SWAR
if (match(Instr, m_LShr(m_And(m_Mul(m_And(m_Add(m_Value(X), ...),
    ...), ...), ...)))) {
    // Verify the specific SWAR masks (0x55, 0x33, 0x0F) for
    correctness
    if (VerifyPopCountConstants(Instr)) {
        return rewritePopCount(Instr, X);
    }
}
```

Listing 3.15: Matcher για PopCount (SWAR)

Λεπτομέρειες υλοποίησης:

- Ο matcher σαρώνει αναδρομικά το δέντρο προς τα πίσω (bottom-up).
- Η συνάρτηση `VerifyPopCountConstants` ελέγχει αν οι μάσκες είναι οι σωστές (`0x55`, `0x33`, `0x0F`).
- Εάν έστω και μία σταθερά διαφέρει, ο matcher απορρίπτει το μοτίβο για να αποφευχθεί η αλλαγή της σημασιολογίας.

3.2.3.6 Ανάλυση Rewriter

Η επανεγγραφή εδώ επιφέρει τη μεγαλύτερη μείωση εντολών από οποιοδήποτε άλλο ιδίωμα.

```
// Use the built-in ctpop intrinsic for population count
Function *PopCountFn = Intrinsic::getDeclaration(Mod, Intrinsic::
    ctpop, {X->getType()});
// Collapse the entire complex graph into a single call
Value *NewCall = Builder.CreateCall(PopCountFn, {X}, "popcount_opt
    ");
// Final replacement of the old logic
Instr->replaceAllUsesWith(NewCall);
```

Listing 3.16: Κώδικας Rewriter για PopCount

3.2.3.7 Target Assembly

Η διαφορά είναι χαώδης. Από τις 20+ εντολές assembly που απαιτούνται για την προσομοίωση του SWAR, καταλήγουμε σε μία:

```
1 popcntl %edi, %eax
```

Listing 3.17: x86 Assembly Output

3.2.4 Ακέραια Απόλυτη Τιμή (Integer Absolute Value)

3.2.4.1 Περιγραφή Προβλήματος

Η εύρεση της απόλυτης τιμής ενός ακεραίου είναι μια από τις πιο συχνές λειτουργίες σε αλγορίθμους επεξεργασίας σήματος και γραφικών. Στη γλώσσα C, η τυπική υλοποίηση βασίζεται στον τριαδικό τελεστή (ternary operator): `x < 0 ? -x : x`.

Το κύριο πρόβλημα με αυτή την προσέγγιση είναι ότι εισάγει μια εξάρτηση ροής ελέγχου (control flow dependency). Σε επίπεδο υλικού, αυτό μεταφράζεται σε μια υπό συνθήκη διακλάδωση (conditional branch), η οποία μπορεί να προκαλέσει καθυστερήσεις λόγω αποτυχίας πρόβλεψης διακλάδωσης (branch misprediction) στον επεξεργαστή. Η βελτιστοποίηση στοχεύει στην «ισοπέδωση» (flattening) του γράφου ροής ελέγχου και τη χρήση εξειδικευμένων εντολών που δεν απαιτούν διακλάδωση.

3.2.4.2 Θεωρητική και Υλική Τεκμηρίωση της Βελτιστοποίησης

Η μετάβαση από τον κώδικα διακλαδώσεων (branching code) σε γραμμικό κώδικα (branchless code) για τον υπολογισμό της απόλυτης τιμής, στηρίζεται στις ιδιότητες του Συμπληρώματος ως προς 2 (Two's Complement) και στην ανάγκη αποφυγής των κινδύνων της διοχέτευσης (pipeline hazards).

Α. Επαλήθευση μέσω Δυαδικής Λογικής (Branchless Logic) Στο σύστημα Συμπληρώματος ως προς 2, η απόλυτη τιμή ενός αρνητικού αριθμού x ισούται με την αντιστροφή όλων των bits του (NOT) συν 1 (δηλαδή $-x = \sim x + 1$). Η βελτιστοποίηση εκμεταλλεύεται το πρόσημο του αριθμού για να δημιουργήσει μια μάσκα, εξαλείφοντας την ανάγκη για `if`.

Ο αλγόριθμος λειτουργεί ως εξής:

1. Ολισθαίνουμε τον αριθμό x δεξιά κατά 31 θέσεις (αριθμητική ολίσθηση). Αν το x είναι θετικό, η μάσκα M γίνεται 0 (`00...00`). Αν είναι αρνητικό, η μάσκα M γίνεται -1 (`11...11`).

2. Υπολογίζουμε το $(x \oplus M) - M$.

Ας εξετάσουμε τις δύο περιπτώσεις για έναν ακέραιο 32-bits:

Βήμα	**Περίπτωση Θετικού** ($x = 5$)	**Περίπτωση Αρνητικού** ($x = -5$)
Δυαδική Τιμή	`0000...0101`	`1111...1011`
Μάσκα ($x \gg 31$)	$M = 0$ (`0000...0000`)	$M = -1$ (`1111...1111`)
ΞΟΡ ($x \oplus M$)	$5 \oplus 0 = 5$	$-5 \oplus -1 = 4$ (`0000...0100`)
Αφαίρεση ($-M$)	$5 - 0 = \mathbf{5}$	$4 - (-1) = 4 + 1 = \mathbf{5}$

ΠΙΝΑΚΑΣ 3.4: Μαθηματική επαλήθευση της 'branchless' τεχνικής. Παρατηρούμε ότι η ίδια ακολουθία εντολών παράγει το σωστό αποτέλεσμα και για τις δύο περιπτώσεις χωρίς διακλάδωση.

Β. Ανάλυση Πολυπλοκότητας Κυκλώματος (Pipeline Efficiency) Η υπεροχή της χρήσης του intrinsic `@llvm.abs` (που μεταφράζεται είτε στον παραπάνω αλγόριθμο είτε σε ειδική εντολή όπως η `PABSD`) έγκειται στην προβλεψιμότητα.

- **Διακλάδωση (Conditional Branch):** Η εντολή σύγκρισης και άλματος (`cmp` + `jge`) εισάγει αβεβαιότητα. Ο επεξεργαστής προσπαθεί να μαντέψει το αποτέλεσμα (Branch Prediction).
  - Αν η πρόβλεψη είναι σωστή: Κόστος $\sim 1 - 2$ κύκλοι.
  - Αν η πρόβλεψη είναι λάθος (Misprediction): Ο επεξεργαστής πρέπει να αδειάσει τη διοχέτευση (pipeline flush), κοστίζοντας **15-20 κύκλους ρολογιού**. Σε τυχαία δεδομένα (π.χ. επεξεργασία σήματος θορύβου), η απόδοση πέφτει δραματικά.
- **Γραμμική Εκτέλεση (cmov / pabsd):** Η βελτιστοποιημένη εκδοχή δεν περιέχει άλματα. Ο αριθμός των κύκλων είναι σταθερός και ντετερμινιστικός (συνήθως **1-3 κύκλοι** για την εντολή `pabsd` σε x86).

Συμπερασματικά, η αντικατάσταση της διακλάδωσης:

1. **Εξαλείφει το κόστος του Branch Misprediction:** Εγγυάται σταθερή απόδοση ανεξαρτήτως των δεδομένων εισόδου.

2. **Αυξάνει το Instruction Level Parallelism (ILP):** Ο επεξεργαστής μπορεί να εκτελέσει κερδοσκοπικά (speculatively) τις επόμενες εντολές χωρίς τον φόβο ακύρωσής τους.

3.2.4.3 Μη βελτιστοποιημένος Κώδικας

Ο μεταγλωττιστής Clang παράγει συνήθως μια εντολή `select`, η οποία εξαρτάται από το αποτέλεσμα μιας σύγκρισης `icmp`.

```
int my_abs(int x) {
    return (x < 0) ? -x : x; // Branch-based implementation
}
```

Listing 3.18: Απόλυτη τιμή σε C

```
%cmp = icmp slt i32 %x, 0
%neg = sub nsw i32 0, %x
%res = select i1 %cmp, i32 %neg, i32 %x
```

Listing 3.19: Μη βελτιστοποιημένη LLVM IR

3.2.4.4 Ανάλυση Matcher

Ο matcher για την απόλυτη τιμή πρέπει να εντοπίσει το «τρίγωνο» εξάρτησης μεταξύ της σύγκρισης, της άρνησης και της επιλογής.

```
Value *X, *Cond, *TrueVal, *FalseVal;
// Match: select (x < 0), -x, x
if (match(Instr, m_Select(m_Value(Cond), m_Value(TrueVal), m_Value
    (FalseVal)))) {
    // Check if condition is x < 0
    if (match(Cond, m_ICmp(ICmpInst::ICMP_SLT, m_Value(X),
    m_SpecificInt(0)))) {
        // Check if TrueVal is -x and FalseVal is x
        if (match(TrueVal, m_Neg(m_Deferred(X))) && FalseVal == X)
    {
            return rewriteAbs(Instr, X);
        }
    }
}
```

Listing 3.20: Κώδικας Matcher για Absolute Value

Ανάλυση λογικής:

- `m_Select`: Εντοπίζει την εντολή επιλογής που αποτελεί τη ρίζα του ιδιώματος.
- `m_ICmp(ICmpInst::ICMP_SLT, ...)`: Επαληθεύει ότι η συνθήκη είναι «μικρότερο από το μηδέν» (Signed Less Than).
- `m_Neg(m_Deferred(X))`: Επιβεβαιώνει ότι η τιμή που επιλέγεται όταν η συνθήκη είναι αληθής, είναι η αριθμητική άρνηση της αρχικής τιμής X.

3.2.4.5 Ανάλυση Rewriter

Η επανεγγραφή χρησιμοποιεί το intrinsic `@llvm.abs`, το οποίο δέχεται δύο παραμέτρους: την τιμή και ένα boolean flag που ορίζει αν η τιμή INT_MIN θεωρείται απροσδιόριστη συμπεριφορά (poison).

```
// Declare the absolute value intrinsic
Function *AbsFn = Intrinsic::getDeclaration(Mod, Intrinsic::abs, {
    X->getType()});
// Create call: @llvm.abs(X, /*is_int_min_poison=*/false)
Value *NewAbs = Builder.CreateCall(AbsFn, {X, Builder.getFalse()},
     "abs_opt");
// Replace select/icmp tree
Instr->replaceAllUsesWith(NewAbs);
```

Listing 3.21: Κώδικας Rewriter για Absolute Value

3.2.4.6 Βελτιστοποιημένη IR και Target Assembly

Μετά τη βελτιστοποίηση, ο γράφος ροής ελέγχου απλοποιείται πλήρως.

```
%abs_opt = call i32 @llvm.abs.i32(i32 %x, i1 false)
```

Listing 3.22: Βελτιστοποιημένη LLVM IR

Στην αρχιτεκτονική x86-64, αν ο επεξεργαστής υποστηρίζει το σύνολο εντολών SSE3 ή μεταγενέστερο, το backend παράγει την εντολή `pabsd`, η οποία εκτελεί την πράξη χωρίς καμία διακλάδωση:

```
movl    %edi, %eax
negl    %eax
cmovsl  %edi, %eax
# Or if SSE is enabled:
pabsd   %xmm0, %xmm0
```

Listing 3.23: x86 Assembly Output

3.2.5 Λειτουργίες Ελαχίστου και Μεγίστου (Minimum/Maximum Idioms)

3.2.5.1 Περιγραφή Προβλήματος: Το Κόστος της Διακλάδωσης

Στις γλώσσες προγραμματισμού υψηλού επιπέδου όπως η C και η C++, δεν υπάρχουν ενσωματωμένοι τελεστές για τον υπολογισμό του ελαχίστου (min) ή του μεγίστου (max). Οι προγραμματιστές αναγκάζονται να χρησιμοποιούν τον τριαδικό τελεστή υπό συνθήκη:

```
int m = (a < b) ?  a :  b;
```

Για τον άνθρωπο, η πρόθεση είναι ξεκάθαρη: «επέλεξε τη μικρότερη τιμή». Ωστόσο, για τον μεταγλωττιστή και τον επεξεργαστή, αυτή η δομή μεταφράζεται αρχικά ως **Ροή Ελέγχου (Control Flow)**. Ο κώδικας διασπάται σε μια σύγκριση και μια εντολή διακλάδωσης (branch/jump).

Το πρόβλημα με αυτή την προσέγγιση είναι ότι εισάγει μια **Εξάρτηση Ελέγχου (Control Dependency)**. Ο επεξεργαστής δεν γνωρίζει ποια εντολή θα εκτελεστεί μετά τη σύγκριση μέχρι να ολοκληρωθεί η σύγκριση. Για να διατηρήσει υψηλή απόδοση, προσπαθεί να μαντέψει το αποτέλεσμα (Branch Prediction). Σε αλγορίθμους όπου τα δεδομένα είναι τυχαία (π.χ. ταξινόμηση, δέντρα αναζήτησης), η πρόβλεψη αποτυγχάνει συχνά, προκαλώντας άδειασμα της διοχέτευσης (pipeline flush) και τεράστια ποινή σε κύκλους ρολογιού.

3.2.5.2 Θεωρητική και Υλική Τεκμηρίωση της Βελτιστοποίησης

Η προτεινόμενη λύση αντικαθιστά τη δομή διακλάδωσης με τα εγγενή intrinsics `@llvm.smin` και `@llvm.smax`. Αυτή η μετατροπή είναι κρίσιμη διότι μετατρέπει την Εξάρτηση Ελέγχου σε **Εξάρτηση Δεδομένων (Data Dependency)**, επιτρέποντας τη χρήση ειδικών εντολών που εκτελούνται σε σταθερό χρόνο.

Α. Λογική Επαλήθευση Η ορθότητα της μετατροπής αποδεικνύεται από τον πίνακα αληθείας. Η λογική της συνάρτησης $min(a, b)$ είναι αυστηρά ισοδύναμη με τη λογική της επιλογής, καλύπτοντας όλες τις πιθανές σχέσεις διάταξης μεταξύ των a και b.

Σχέση Τιμών	**Τριαδική Λογική** $(?a : b)$	**Μαθηματικό** $min(a, b)$	**Συμπέρασμα**
$a < b$	Επιλέγεται το a	a	Ταύτιση
$a > b$	Επιλέγεται το b	b	Ταύτιση
$a == b$	Επιλέγεται το b (αλλά $a = b$)	a ή b	Ταύτιση

Πινακας 3.5: Πίνακας επαλήθευσης της ισοδυναμίας. Η αντικατάσταση της διακλάδωσης με αριθμητική πράξη δεν αλλοιώνει το αποτέλεσμα για καμία είσοδο.

Β. Ανάλυση Υλικού: Από το Branching στο Predication Η χρήση των intrinsics επιτρέπει στο Backend να παράγει κώδικα που είναι απρόσβλητος από τα σφάλματα πρόβλεψης διακλαδώσεων.

- **Εξάλειψη Κινδύνων Ελέγχου (Control Hazards):** Στην κλασική υλοποίηση με `CMP + JGE`, μια λανθασμένη πρόβλεψη κοστίζει 15-20 κύκλους. Με τη χρήση του intrinsic, ο μεταγλωττιστής παράγει την εντολή `CMOV` **(ὅνδιτιοναλ Μοε)** στην αρχιτεκτονική x86. Η εντολή αυτή διαβάζει και τις δύο πηγές (a και b) και γράφει στον καταχωρητή στόχο βάσει των σημαιών (flags), χωρίς να διακόπτει τη ροή των εντολών. Ο χρόνος εκτέλεσης είναι σταθερός και προβλέψιμος.
- **Δυνατότητα Διανυσματοποίησης (Auto-Vectorization):** Αυτό είναι το σημαντικότερο πλεονέκτημα. Ένας βρόχος που περιέχει `if/else` είναι εξαιρετικά δύσκολο να παραλληλιστεί (διανυσματοποιηθεί). Αντίθετα, η ύπαρξη του intrinsic `@llvm.smin` ενημερώνει τον μεταγλωττιστή ότι η πράξη είναι καθαρά αριθμητική. Αυτό επιτρέπει τη χρήση εντολών SIMD όπως η `PMINSD` (Packed Minimum Signed Dword), η οποία υπολογίζει τα ελάχιστα 4, 8 ή 16 ζευγών αριθμών **ταυτόχρονα** σε έναν κύκλο ρολογιού.

Συμπερασματικά, η βελτιστοποίηση αυτή μετατρέπει έναν σειριακό, απρόβλεπτο κώδικα σε έναν παραλληλίσιμο και ντετερμινιστικό κώδικα, πολλαπλασιάζοντας την απόδοση σε σύγχρονους επεξεργαστές.

3.2.5.3 Μη βελτιστοποιημένος Κώδικας

Στην μη βελτιστοποιημένη LLVM IR, το ιδίωμα εμφανίζεται ως ένας συνδυασμός σύγκρισης (`icmp`) και επιλογής (`select`).

```
int find_min(int a, int b) {
    return (a < b) ? a : b; // Ternary-based min
}
```

Listing 3.24: Ελάχιστο (Min) σε C

```
%cmp = icmp slt i32 %a, %b
%res = select i1 %cmp, i32 %a, i32 %b
```

Listing 3.25: Μη βελτιστοποιημένη LLVM IR

3.2.5.4 Ανάλυση Matcher

Ο matcher πρέπει να αναγνωρίσει το μοτίβο όπου μια εντολή `select` χρησιμοποιεί το αποτέλεσμα μιας σύγκρισης `icmp` για να επιλέξει μεταξύ των δύο αρχικών τελεστών της σύγκρισης.

```
Value *A, *B, *Cond;
// Match: select (icmp cond A, B), A, B
if (match(Instr, m_Select(m_Value(Cond), m_Value(A), m_Value(B))))
    {
    ICmpInst::Predicate Pred;
    // Check if the condition is an integer comparison between A
    and B
    if (match(Cond, m_ICmp(Pred, m_Specific(A), m_Specific(B)))) {
        // Determine if it is a Min or Max pattern based on the
    predicate
        if (Pred == ICmpInst::ICMP_SLT || Pred == ICmpInst::
    ICMP_SLE)
            return rewriteMinMax(Instr, A, B, Intrinsic::smin);
        else if (Pred == ICmpInst::ICMP_SGT || Pred == ICmpInst::
    ICMP_SGE)
            return rewriteMinMax(Instr, A, B, Intrinsic::smax);
    }
}
```

Listing 3.26: Κώδικας Matcher για Min/Max

Ανάλυση λογικής:

- `m_Specific(A)`: Διασφαλίζει ότι οι τιμές που συγκρίνονται είναι ακριβώς οι ίδιες με αυτές που επιλέγονται (operand consistency).
- Predicates: Ο matcher ελέγχει αν η σύγκριση είναι «μικρότερο από» (SLT/SLE) για το Min ή «μεγαλύτερο από» (SGT/SGE) για το Max.

3.2.5.5 Ανάλυση Rewriter

Ο rewriter αντικαθιστά το ζεύγος εντολών με μια κλήση στο αντίστοιχο signed min/max intrinsic.

```
// Get the intrinsic declaration for smin or smax
Function *MinMaxFn = Intrinsic::getDeclaration(Mod, IID, {A->
    getType()});
// Create the optimized call using IRBuilder
```

```
Value *NewInst = Builder.CreateCall(MinMaxFn, {A, B}, "minmax_opt");
// Replace all uses of the old select instruction
Instr->replaceAllUsesWith(NewInst);
```

Listing 3.27: Κώδικας Rewriter για Min/Max

3.2.5.6 Βελτιστοποιημένη IR και Target Assembly

Η βελτιστοποιημένη IR είναι πλέον γραμμική, διευκολύνοντας την περαιτέρω ανάλυση από τα επόμενα περάσματα.

```
%minmax_opt = call i32 @llvm.smin.i32(i32 %a, i32 %b)
```

Listing 3.28: Βελτιστοποιημένη LLVM IR

Στο επίπεδο του υλικού (backend), αυτό μεταφράζεται σε εξαιρετικά αποδοτικό κώδικα. Στην αρχιτεκτονική x86-64, χρησιμοποιείται η εντολή `cmov` (conditional move) ή, αν είναι ενεργοποιημένες οι SSE4.1 εντολές, η `pminsd`:

```
# x86-64 (using conditional move)
cmpl    %esi, %edi
cmovgl  %esi, %edi
movl    %edi, %eax

# x86-64 (with SSE4.1)
pminsd  %xmm1, %xmm0
```

Listing 3.29: Target Assembly Output

3.3 Λεπτομέρειες Υλοποίησης

Η υλοποίηση βασίζεται σε μεγάλο βαθμό στη βιβλιοθήκη `PatternMatch.h` του LLVM για την ανίχνευση και στον `IRBuilder` για την παραγωγή κώδικα.

3.3.1 Το API Matcher

Αντί για τον χειροκίνητο έλεγχο των κωδικών πράξης (opcodes) και των τελεστέων (που είναι επιρρεπής σε λάθη και φλύαρος), χρησιμοποιούμε το δηλωτικό API ταύτισης του LLVM. Αυτό μας επιτρέπει να περιγράψουμε το σχήμα του Δέντρου Εντολών χρησιμοποιώντας απλά πρότυπα όπως `m_Add`, `m_Shl`, ή `m_Value`.

Το ακόλουθο απόσπασμα κώδικα επιδεικνύει τον matcher για το μοτίβο Περιστροφής Bit. Περιγράφει συνοπτικά ένα δέντρο όπου η ρίζα είναι μια πράξη OR και τα παιδιά της είναι μια Αριστερή Ολίσθηση και μια Λογική Δεξιά Ολίσθηση:

```
// Match pattern: (x << n) | (x >> (32 - n))
Value *X, *N;
if (match(Instr, m_Or(
        m_Shl(m_Value(X), m_Value(N)),
        m_LShr(m_Deferred(X), m_Sub(m_SpecificInt(32), m_Deferred(
    N)))
    ))) {
    // Pattern matched! Invoke rewriter...
    return replaceWithRotate(Instr, X, N);
}
```

Listing 3.30: Κώδικας Matcher για Περιστροφή Bit

Παρατηρούμε τη χρήση του `m_Deferred(X)`, το οποίο επιβάλλει ότι η τιμή που βρέθηκε στο δεξί σκέλος του δέντρου πρέπει να είναι *ίδια* με αυτήν που βρέθηκε στο αριστερό.

3.3.2 Ο Rewriter και ο IRBuilder

Μόλις ταυτοποιηθεί ένα μοτίβο, ξεκινά η φάση επανεγγραφής. Χρησιμοποιούμε τον `IRBuilder` για την εισαγωγή νέων εντολών. Ένας κρίσιμος μηχανισμός εδώ είναι το `replaceAllUsesWith` (RAUW). Αυτή η συνάρτηση διασφαλίζει ότι όλες οι άλλες εντολές στον κώδικα που βασίζονταν στο παλιό, αναποτελεσματικό αποτέλεσμα ενημερώνονται αυτόματα ώστε να χρησιμοποιούν το νέο, βελτιστοποιημένο αποτέλεσμα.

Το απόσπασμα παρακάτω δείχνει πώς το μοτίβο Υπόλοιπου Δύναμης του 2 επανεγγράφεται σε ένα Bitwise AND:

```
// Transform: x % C  ->  x & (C - 1)
// where C is a power of 2
Value *Op0 = Instr->getOperand(0);
ConstantInt *C = cast<ConstantInt>(Instr->getOperand(1));

// Create the mask: C - 1
Constant *Mask = ConstantInt::get(C->getType(), C->getValue() - 1)
    ;

IRBuilder<> Builder(Instr);
// Create the new AND instruction
Value *NewInst = Builder.CreateAnd(Op0, Mask, "mod_opt");

```

```
// Replace old instruction with new one
Instr->replaceAllUsesWith(NewInst);
```

Listing 3.31: Κώδικας Rewriter για Modulo Power-of-2

3.3.3 Έλεγχοι Ασφαλείας

Για τη διασφάλιση της ορθότητας του μετασχηματισμού, το πέρασμα εκτελεί διάφορους ελέγχους ασφαλείας:

- **Επαλήθευση Τύπων:** Οι matchers διασφαλίζουν ότι οι πράξεις εκτελούνται σε κατάλληλους τύπους (π.χ. επαλήθευση ακεραίων 32-βιτ για συγκεκριμένα μοτίβα).
- **Διατήρηση SSA:** Χρησιμοποιώντας τον `IRBuilder` και το `replaceAllUsesWith`, το πέρασμα διατηρεί αυτόματα τη μορφή SSA (Def-Use chains).
- **Επαλήθευση:** Το πέρασμα επικυρώθηκε χρησιμοποιώντας το εργαλείο `opt -verify` για τον εντοπισμό τυχόν κακοσχηματισμένης IR αμέσως μετά τον μετασχηματισμό.

3.4 Ανάλυση Αντιπροσωπευτικών Μετασχηματισμών

Στην παρούσα ενότητα παραθέτουμε την πλήρη λογική για τρία επιλεγμένα ιδιώματα. Κάθε παράδειγμα περιλαμβάνει τόσο το στάδιο της ταύτισης (match) όσο και το στάδιο της επανεγγραφής (rewrite), καταδεικνύοντας τη χρήση του `IRBuilder` και τη διατήρηση των αλυσίδων ορισμού-χρήσης.

3.4.1 Αναγνώριση και Επανεγγραφή του PopCount

Η επανεγγραφή του PopCount είναι εντυπωσιακή, καθώς αντικαθιστά μια ολόκληρη αλυσίδα εντολών με μία μόνο κλήση intrinsic.

```
Value *X;
// 1.                                    (Matching)
if (match(Instr, m_Add(
        m_And(m_Value(X), m_SpecificInt(0x55555555)),
        m_And(m_LShr(m_Deferred(X), m_SpecificInt(1)),
   m_SpecificInt(0x55555555))
    ))) {

    // 2.                                          (Rewriting)
```

```
9     //                        intrinsic @llvm.ctpop
10    Function *PopCountFn = Intrinsic::getDeclaration(Mod,
    Intrinsic::ctpop, {Instr->getType()});
11
12    //
13    Value *NewCall = Builder.CreateCall(PopCountFn, {X}, "popcnt")
    ;
14
15    //

16    Instr->replaceAllUsesWith(NewCall);
17    return true;
18 }
```

Listing 3.32: Πλήρης μετασχηματισμός PopCount

3.4.2 Μετασχηματισμός και 'Flattening' της Απόλυτης Τιμής

Εδώ ο rewriter παίζει κρίσιμο ρόλο, καθώς μετατρέπει μια δομή που θα παρήγαγε διακλαδώσεις σε μια ευθεία γραμμή κώδικα (straight-line code).

```
1  Value *Cond, *TrueVal, *FalseVal, *X;
2  if (match(Instr, m_Select(m_Value(Cond), m_Value(TrueVal), m_Value
       (FalseVal)))) {
3     if (match(Cond, m_ICmp(ICmpInst::ICMP_SLT, m_Value(X),
       m_SpecificInt(0))) &&
4         match(TrueVal, m_Neg(m_Deferred(X))) && (FalseVal == X)) {
5
6         //                          intrinsic @llvm.abs(x,
       is_int_min_poison)
7         Function *AbsFn = Intrinsic::getDeclaration(Mod, Intrinsic
       ::abs, {X->getType()});
8
9         //
                                        (Builder insertion point)
10        Value *NewAbs = Builder.CreateCall(AbsFn, {X, Builder.
       getFalse()}, "abs_opt");
11
12        Instr->replaceAllUsesWith(NewAbs);
13        return true;
14    }
15 }
```

Listing 3.33: Επανεγγραφή Abs με χρήση IRBuilder

3.4.3 Strength Reduction: Από Πολλαπλασιασμό σε Ολίσθηση

Αυτό το παράδειγμα δείχνει πώς ο rewriter υπολογίζει νέες παραμέτρους (το μέγεθος της ολίσθησης) πριν δημιουργήσει τη νέα εντολή.

```
1  Value *Op0; ConstantInt *Op1;
2  if (match(Instr, m_Mul(m_Value(Op0), m_ConstantInt(Op1)))) {
3      const APInt &Val = Op1->getValue();
4      if (Val.isPowerOf2()) {
5          // (log2)
                  shift amount
6          uint32_t ShiftAmt = Val.logBase2();
7          Value *ShAmtVal = ConstantInt::get(Op1->getType(),
       ShiftAmt);
8
9          // shl (Shift
       Left)
10         Value *NewShl = Builder.CreateShl(Op0, ShAmtVal, "shl_pow2
       ");
11
12         Instr->replaceAllUsesWith(NewShl);
13         return true;
14     }
15 }
```

Listing 3.34: Υπολογισμός και αντικατάσταση Mul με Shl

Κεφάλαιο 4

Αξιολόγηση και Αποτελέσματα

Στο προηγούμενο κεφάλαιο αναλύσαμε τον σχεδιασμό και την υλοποίηση του `PatternSelectPass`, εστιάζοντας στους αλγορίθμους ταύτισης και μετασχηματισμού. Στο παρόν κεφάλαιο, προχωρούμε στην αυστηρή πειραματική αξιολόγηση του προτεινόμενου συστήματος.

Στόχος της αξιολόγησης είναι να απαντηθούν τρία θεμελιώδη ερωτήματα:

1. **Ορθότητα (Correctness):** Παράγει ο μετασχηματισμός σημασιολογικά ισοδύναμο κώδικα, χωρίς να αλλοιώνει τη λογική του προγράμματος·
2. **Αποδοτικότητα (Efficiency):** Σε τι βαθμό μειώνεται το μέγεθος της Ενδιάμεσης Αναπαράστασης (IR) και πώς αυτό μεταφράζεται σε μείωση του χρόνου εκτέλεσης·
3. **Αξιοποίηση Υλικού (Hardware Utilization):** Καταφέρνει το σύστημα να ενεργοποιήσει τις εξειδικευμένες εντολές του επεξεργαστή (όπως `POPCNT`, `ROL`, `CMOV`)·

Για την τεκμηρίωση των απαντήσεων, αναπτύξαμε μια αυτοματοποιημένη ροή ελέγχου και μέτρησης, τα αποτελέσματα της οποίας παρουσιάζονται αναλυτικά στη συνέχεια.

4.1 Μεθοδολογία Αξιολόγησης και Εργαλεία

Η αξιοπιστία των πειραματικών αποτελεσμάτων εξαρτάται άμεσα από τη σταθερότητα του περιβάλλοντος δοκιμών και την ακρίβεια της μεθοδολογίας μέτρησης.

4.1.1 Πειραματική Διάταξη (Hardware & Software)

Η ανάπτυξη, η μεταγλώττιση και η εκτέλεση των πειραμάτων πραγματοποιήθηκαν σε ένα σύγχρονο υπολογιστικό σύστημα με αρχιτεκτονική x86-64. Επιλέχθηκε ο επεξεργαστής

Intel Core i7-1165G7 (αρχιτεκτονική Tiger Lake), καθώς υποστηρίζει εγγενώς προηγμένα σύνολα εντολών (BMI2, AVX-512) που είναι κρίσιμα για την αξιολόγηση ιδιωμάτων όπως το PopCount και το Rotate.

Οι λεπτομερείς προδιαγραφές του συστήματος παρατίθενται παρακάτω:

- **Επεξεργαστής (CPU):** 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz (4 Cores, 8 Threads).
- **Μνήμη (RAM):** 16 GB DDR4.
- **Λειτουργικό Σύστημα:** Ubuntu Linux 22.04 LTS (εκτελούμενο μέσω WSL2).
- **Υποδομή Μεταγλωττιστή:** LLVM 18.1.0, μεταγλωττισμένο από τον πηγαίο κώδικα σε Release mode.
- **Σύστημα Κατασκευής:** CMake 3.22 και Ninja 1.10.

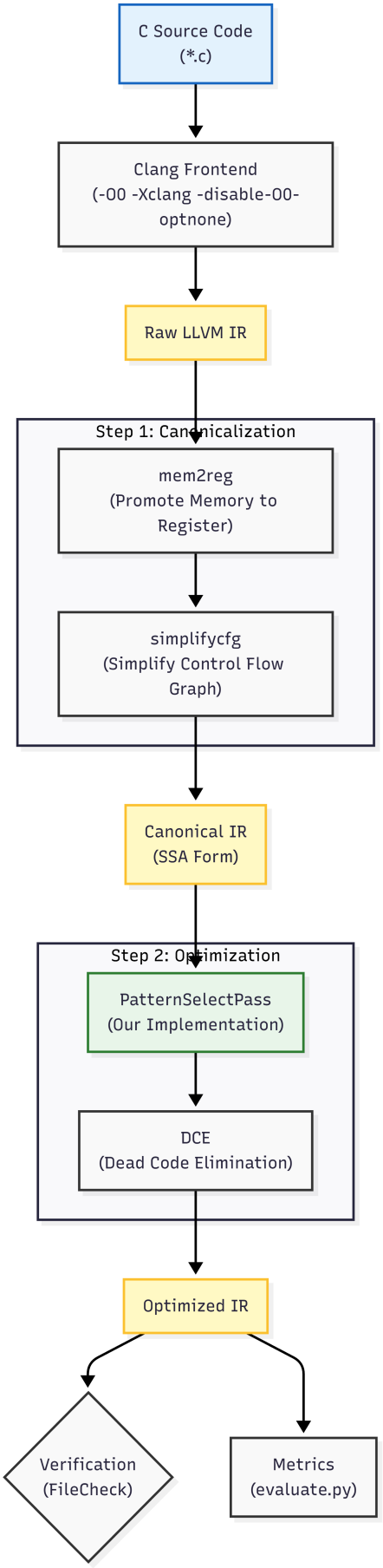
4.1.2 Ροή Εργασίας (Evaluation Pipeline)

Μια από τις σημαντικότερες προκλήσεις κατά τη δοκιμή ενός περάσματος βελτιστοποίησης είναι η απομόνωση της επίδρασής του. Για να διασφαλίσουμε ότι μετράμε αποκλειστικά τη συνεισφορά του PatternSelectPass, σχεδιάσαμε μια αυστηρή αλληλουχία μετασχηματισμών (pipeline), η οποία απεικονίζεται στο Διάγραμμα 4.1.

Η διαδικασία αποτελείται από δύο διακριτά στάδια:

1. **Στάδιο Κανονικοποίησης (Canonicalization):** Ο πηγαίος κώδικας C μεταγλωττίζεται αρχικά σε Raw IR με απενεργοποιημένες τις βελτιστοποιήσεις (`-O0 -Xclang -disable-O0-optnone`). Στη συνέχεια, εφαρμόζονται τα περάσματα `mem2reg` και `simplifycfg`.
   - Το `mem2reg` προάγει τις μεταβλητές μνήμης σε καταχωρητές, κατασκευάζοντας τη μορφή SSA.
   - Το `simplifycfg` αφαιρεί τα περιττά βασικά μπλοκ και τις αχρείαστες διακλαδώσεις.

   Αυτό το βήμα είναι κρίσιμο, καθώς καθαρίζει τον ¨θόρυβο' της IR και αποκαλύπτει τη δομή των ιδιωμάτων, επιτρέποντας στον matcher μας να λειτουργήσει αποτελεσματικά.
2. **Στάδιο Βελτιστοποίησης (Optimization):** Στην καθαρή IR εφαρμόζεται το PatternSelectPass. Αμέσως μετά, εκτελείται το πέρασμα `DCE (Dead Code Elimination)`, ώστε να διαγραφούν οι παλιές εντολές που αντικαταστάθηκαν από τα νέα intrinsics και δεν χρησιμοποιούνται πλέον.



Σχημα 4.1: Η ροή εργασίας (pipeline) της αξιολόγησης. Διακρίνονται τα στάδια της Κανονικοποίησης (με `mem2reg`, `simplifycfg`) και της Βελτιστοποίησης.

4.1.3 Εργαλεία Επαλήθευσης

Για την υλοποίηση της παραπάνω ροής και την εξαγωγή αποτελεσμάτων, αναπτύχθηκαν και χρησιμοποιήθηκαν τα ακόλουθα εξειδικευμένα εργαλεία:

4.1.3.1 Μηχανισμός Επαλήθευσης: FileCheck

Η ορθότητα των μετασχηματισμών επαληθεύεται μέσω του εργαλείου `FileCheck`, το οποίο αποτελεί το βιομηχανικό πρότυπο για regression testing στο LLVM. Σε αντίθεση με τα κλασικά unit tests, το FileCheck λειτουργεί κάνοντας ταύτιση προτύπων (pattern matching) στην έξοδο κειμένου του μεταγλωττιστή.

Για την παρούσα διατριβή, αναπτύξαμε εξειδικευμένους κανόνες ελέγχου (directives) που διασφαλίζουν δύο συνθήκες:

1. **Θετική Επιβεβαίωση (Positive Verification):** Ο βελτιστοποιημένος κώδικας πρέπει να περιέχει το αναμενόμενο intrinsic.
2. **Αρνητική Επιβεβαίωση (Negative Verification):** Ο κώδικας δεν πρέπει να περιέχει τις αρχικές εντολές (αν πέτυχε η βελτιστοποίηση) ή δεν πρέπει να περιέχει intrinsic αν η είσοδος δεν πληροί τις προϋποθέσεις.

Ακολουθούν παραδείγματα από τη σουίτα ελέγχου που κατασκευάσαμε:

Α. Έλεγχος Επιτυχούς Βελτιστοποίησης Στο παρακάτω απόσπασμα από το αρχείο `positive_tests.ll`, χρησιμοποιούμε την οδηγία `CHECK` για να επιβεβαιώσουμε ότι οι πολύπλοκες πράξεις αντικαταστάθηκαν. Για παράδειγμα, στον πολλαπλασιασμό επί 8 (`@times8`), περιμένουμε να δούμε ολίσθηση (`shl`) κατά 3 θέσεις ($2^3 = 8$).

```
; CHECK-LABEL: define dso_local i32 @times8
; CHECK: {{.*}} = shl i32 {{.*}}, 3

; CHECK-LABEL: define dso_local i32 @mod32
; CHECK: {{.*}} = and i32 {{.*}}, 31

; CHECK-LABEL: define dso_local i32 @test_rotate
; CHECK: {{.*}} = call i32 @llvm.fshl.i32(i32 {{.*}}, i32 {{.*}},
   i32 8)

; CHECK-LABEL: define dso_local i32 @test_popcount_logic
; CHECK: {{.*}} = call i32 @llvm.ctpop.i32(i32 {{.*}})
```

Listing 4.1: Κανόνες FileCheck για θετικά σενάρια

Β. Έλεγχος Ασφαλείας (Safety Checks) Ιδιαίτερη έμφαση δόθηκε στα αρνητικά τεστ. Εδώ χρησιμοποιούμε την οδηγία `CHECK-NOT`. Αυτή η εντολή προκαλεί αποτυχία του τεστ εάν βρεθεί η απαγορευμένη συμβολοσειρά. Στο παράδειγμα `@negative_mul_not_pow2`, όπου πολλαπλασιάζουμε επί 7, απαιτούμε:

- Να υπάρχει η εντολή `mul` (διατήρηση αρχικού κώδικα).
- Να **μην** υπάρχει η εντολή `shl` (απαγόρευση λανθασμένης βελτιστοποίησης).

```
1  ; CHECK-LABEL: define dso_local i32 @negative_mul_not_pow2(i32
       noundef %a)
2  ; CHECK: %mul = mul nsw i32 %a, 7
3  ; CHECK-NOT: shl
4  ; CHECK: ret i32 %mul
5
6  ; CHECK-LABEL: define dso_local i32 @negative_rotate_bad_sum
7  ; CHECK: %or = or i32 %shl, %shr
8  ; CHECK-NOT: call i32 @llvm.fshl.i32
9  ; CHECK: ret i32 %or
10
11 ; CHECK-LABEL: define dso_local i32 @negative_abs_wrong_pred
12 ; CHECK: %cond = select i1 %cmp, i32 %sub, i32 %x
13 ; CHECK-NOT: call i32 @llvm.abs.i32
14 ; CHECK: ret i32 %cond
```

Listing 4.2: Κανόνες FileCheck για διασφάλιση ασφάλειας (Negative Tests)

4.1.3.2 Αυτοματοποίηση με το `check_all.sh`

Για την μαζική εκτέλεση αυτών των ελέγχων, αναπτύχθηκε το σενάριο `check_all.sh`. Το σενάριο αυτό εκτελεί τα εξής βήματα για κάθε αρχείο ελέγχου:

1. Καθαρίζει την είσοδο τρέχοντας `-passes=mem2reg,simplifycfg`.
2. Εφαρμόζει το `PatternSelectPass`.
3. Διοχετεύει (pipe) το αποτέλεσμα στο `FileCheck`.

Αν το FileCheck δεν βρει τα αναμενόμενα πρότυπα ή βρει απαγορευμένα πρότυπα, επιστρέφει κωδικό σφάλματος, διακόπτοντας τη διαδικασία και ειδοποιώντας τον χρήστη για την αποτυχία του τεστ (Regression Failure).

4.1.3.3 Στατική Ανάλυση με `evaluate.py`

Για την ποσοτική αξιολόγηση, υλοποιήσαμε το σενάριο Python `evaluate.py`. Το εργαλείο αυτό αναλύει τα αρχεία bitcode πριν και μετά τη βελτιστοποίηση, καταμετρώντας τον αριθμό των εντολών IR. Επιπλέον, παρέχει συγκριτικά στοιχεία μεταξύ της δικής μας υλοποίησης και του ενσωματωμένου βελτιστοποιητή `InstCombine`, επιτρέποντας τον υπολογισμό του ποσοστού μείωσης κώδικα.

4.1.3.4 LLVM Disassembler (llvm-objdump)

Τέλος, για την επιβεβαίωση της μετάφρασης σε κώδικα μηχανής, χρησιμοποιήσαμε τον disassembler του συστήματος. Η ανάλυση των αρχείων Assembly (`.s`) μας επέτρεψε να επιβεβαιώσουμε ότι τα εισαγόμενα intrinsics μεταφράζονται σωστά στις αντίστοιχες εντολές του Tiger Lake επεξεργαστή (π.χ. `popcnt`, `rol`, `bswap`), γεφυρώνοντας έτσι το σημασιολογικό χάσμα.

4.2 Σουίτα Δοκιμών (Test Suite Analysis)

Η αξιολόγηση ενός περάσματος βελτιστοποίησης απαιτεί δεδομένα εισόδου που να είναι ταυτόχρονα αντιπροσωπευτικά και απομονωμένα. Αντί να βασιστούμε αποκλειστικά σε γενικά benchmarks (όπως το SPEC CPU), τα οποία εισάγουν θόρυβο και πολυπλοκότητα, αναπτύξαμε μια εξειδικευμένη σουίτα μικρο-δοκιμών (micro-benchmarks).

Η σουίτα αποτελείται από δύο αρχεία πηγαίου κώδικα C, το `positive_tests.c` και το `negative_tests.c`, τα οποία καλύπτουν το φάσμα της¨επιθυμητής' και της 'απαγορευμένης' συμπεριφοράς του μεταγλωττιστή.

4.2.1 Θετικά Σενάρια (Positive Tests)

Το αρχείο `positive_tests.c` περιέχει συναρτήσεις που υλοποιούν τα 7 στοχευμένα ιδιώματα χρησιμοποιώντας τυπική σύνταξη C. Στόχος είναι να επιβεβαιώσουμε ότι το PatternSelectPass είναι ικανό να αναγνωρίσει τη σημασιολογία αυτών των δομών και να τις αντικαταστήσει με τα βέλτιστα intrinsics.

Τα σενάρια κατηγοριοποιούνται ως εξής:

4.2.1.1 A. Αριθμητικές Απλοποιήσεις

Ελέγχουμε αν οι ακριβές πράξεις πολλαπλασιασμού και υπολοίπου με δυνάμεις του 2 μετατρέπονται σε φθηνότερες πράξεις bitwise.

```
// Case 1: MulPow2 -> Expected: shl (Shift Left)
int times8(int x) {
    return x * 8; // 8 is 2^3
}

// Case 2: ModPow2 -> Expected: and (Bitwise AND)
unsigned mod32(unsigned x) {
    return x % 32; // 32 is 2^5
}
```

Listing 4.3: Θετικά σενάρια για πολλαπλασιασμό και υπόλοιπο

4.2.1.2 B. Χειρισμός Bits (Bit Manipulation)

Αυτή η κατηγορία περιλαμβάνει τα πιο σύνθετα μοτίβα. Εδώ ο κώδικας C είναι ιδιαίτερα μακροσκελής (όπως στον SWAR PopCount ή στο Byte Swap), και η βελτιστοποίηση αναμένεται να μειώσει δραστικά το μέγεθος του κώδικα.

```
// Case 3: Rotate -> Expected: llvm.fshl
unsigned int test_rotate(unsigned int x) {
    // Standard C idiom: (x << n) | (x >> (32 - n))
    return (x << 8) | (x >> 24);
}

// Case 7: PopCount -> Expected: llvm.ctpop
// SWAR Algorithm (SIMD Within A Register)
int test_popcount_logic(unsigned int i) {
     i = i - ((i >> 1) & 0x55555555);
     i = (i & 0x33333333) + ((i >> 2) & 0x33333333);
     // ... (rest of algorithm)
     return (((i + (i >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
}
```

Listing 4.4: Θετικά σενάρια για Rotate και PopCount

4.2.1.3 Γ. Ροή Ελέγχου (Control Flow)

Εδώ εξετάζουμε τη δυνατότητα του περάσματος να εξαλείφει διακλαδώσεις (`if-else` ή `?:`) για λειτουργίες όπως η απόλυτη τιμή και το ελάχιστο/μέγιστο.

```
// Case 4: Abs -> Expected: llvm.abs
int test_abs(int x) {
    if (x < 0) return -x;
    return x;
}

// Case 5: MinMax -> Expected: llvm.smax / llvm.smin
int test_max(int a, int b) {
    return (a > b) ? a : b;
}
```

Listing 4.5: Θετικά σενάρια για Abs και Min/Max

4.2.2 Αρνητικά Σενάρια (Negative Tests/Safety)

Η ικανότητα ενός μεταγλωττιστή να **μην** εφαρμόζει μια βελτιστοποίηση όταν δεν πρέπει είναι εξίσου σημαντική με την εφαρμογή της. Το αρχείο `negative_tests.c` περιέχει "παγίδες' (corner cases) σχεδιασμένες να ελέγξουν την αυστηρότητα των κανόνων ταύτισης (matchers).

4.2.2.1 Α. Έλεγχος Μαθηματικών Ιδιοτήτων

Ο πολλαπλασιασμός πρέπει να γίνεται μόνο με δυνάμεις του 2. Οποιαδήποτε άλλη σταθερά πρέπει να αγνοείται.

```
// Case: Multiply by 7 (111b) -> Should REMAIN 'mul'
int negative_mul_not_pow2(int a) {
    return a * 7;
}
```

Listing 4.6: Αρνητικό σενάριο: Πολλαπλασιασμός με μη-δύναμη του 2

4.2.2.2 Β. Δομική Ακεραιότητα και Εξαρτήσεις

Στο ιδίωμα της Περιστροφής (Rotate), είναι κρίσιμο να ελέγχεται ότι:

- Η μεταβλητή που ολισθαίνει είναι η ίδια και στα δύο μέρη (αριστερά και δεξιά).
- Το άθροισμα των ολισθήσεων ισούται ακριβώς με το πλάτος του τύπου δεδομένων (32 για i32).

```
// Case: Mismatched variables (x and y) -> Should NOT optimise
unsigned int negative_rotate_diff_vars(unsigned int x, unsigned
    int y, unsigned int s) {
     return (x << s) | (y >> (32 - s));
}

// Case: Incorrect shift sum (Sum is 31, not 32) -> Should NOT
    optimise
unsigned int negative_rotate_bad_sum(unsigned int x, unsigned int
    s) {
     return (x << s) | (x >> (31 - s));
}
```

Listing 4.7: Αρνητικά σενάρια για Rotate

4.2.2.3 Γ. Σημασιολογική Ακρίβεια

Στην περίπτωση της Απόλυτης Τιμής, ο τελεστής σύγκρισης πρέπει να είναι αυστηρά ' 0 (για -ξ) ή ' 0 (για -ξ στο ελσε). Λάθος κατηγορήματα αλλάζουν τη μαθηματική σημασία.

```
// Case: Checking (x > 0) returning -x implies negative-absolute
    value
// This is NOT standard abs(x). Pass should ignore.
int negative_abs_wrong_pred(int x) {
     return (x > 0) ? -x : x;
}
```

Listing 4.8: Αρνητικό σενάριο για Abs με λάθος συνθήκη

4.3 Ανάλυση Περιπτώσεων Χρήσης (Detailed Case Studies)

Στην ενότητα αυτή παρουσιάζουμε τα αποτελέσματα των μετασχηματισμών για κάθε κατηγορία ιδιωμάτων. Για κάθε περίπτωση, παραθέτουμε:

1. Την αρχική IR (όπως προκύπτει μετά την κανονικοποίηση με `mem2reg`).

2. Τη βελτιστοποιημένη IR μετά την εφαρμογή του PatternSelectPass.

3. Τον τελικό κώδικα μηχανής (x86-64 Assembly) που παρήγαγε το backend (`llc -O3`).

4.3.1 Κατηγορία 1: Μείωση Ισχύος (Strength Reduction)

Σε αυτή την κατηγορία, στόχος είναι η αντικατάσταση ¨ακριβών' αριθμητικών πράξεων (πολλαπλασιασμός, διαίρεση) με φθηνότερες λειτουργίες bitwise, όταν οι τελεστέοι είναι δυνάμεις του 2.

4.3.1.1 Πολλαπλασιασμός και Υπόλοιπο

Στο αρχείο `base.ll`, οι συναρτήσεις `@times8` και `@mod32` χρησιμοποιούν τις εντολές `mul` και `urem`. Το πέρασμά μας ανιχνεύει ότι το 8 είναι 2^3 και το 32 είναι 2^5, μετατρέποντας τις εντολές σε ολίσθηση (`shl`) και λογικό ΚΑΙ (`and`) αντίστοιχα.

Baseline IR (base.ll)

```
define i32 @times8(i32 %x) {
  %mul = mul nsw i32 %x, 8
  ret i32 %mul
}

define i32 @mod32(i32 %x) {
  %rem = urem i32 %x, 32
  ret i32 %rem
}

```

Optimized IR (opt.ll)

```
define i32 @times8(i32 %x) {
  %0 = shl i32 %x, 3
  ret i32 %0
}

define i32 @mod32(i32 %x) {
  %0 = and i32 %x, 31
  ret i32 %0
}

```

Πινακας 4.1: Μετατροπή αριθμητικών πράξεων σε bitwise.

Αποτέλεσμα Ασσεμβλυ (opt.s): Ο μεταγλωττιστής εκμεταλλεύεται την απλοποίηση. Για το υπόλοιπο, χρησιμοποιεί την εντολή `andl`, η οποία εκτελείται σε έναν κύκλο ρολογιού.

```
mod32:
    andl    $31, %eax
    retq
```

4.3.2 Κατηγορία 2: Πρότυπα Χειρισμού Bit

Εδώ παρατηρούμε τα πιο εντυπωσιακά αποτελέσματα όσον αφορά τη μείωση του μεγέθους του κώδικα, καθώς πολύπλοκες ακολουθίες εντολών αντικαθίστανται από μία εντολή υλικού.

4.3.2.1 Περιστροφή (Rotate Left)

Η συνάρτηση `@test_rotate` υλοποιεί την περιστροφή μέσω ολισθήσεων (`shl`/`lshr`) και `or`. Το Pass αναγνωρίζει το μοτίβο και εισάγει το intrinsic `fshl` (Funnel Shift Left).

- **Baseline:** 3 εντολές IR (`shl`, `lshr`, `or`).
- **Optimized:** 1 εντολή IR (`call @llvm.fshl`).
- **Assembly:** Χρήση της εντολής `roll` (Rotate Left).

```
1 test_rotate:
2     movl    %edi, %eax
3     roll    $8, %eax        ; Hardware Rotate Instruction
4     retq
```

Listing 4.9: Ο κώδικας μηχανής για το Rotate

4.3.2.2 Εναλλαγή Bytes (Byte Swap)

Η συνάρτηση `@test_bswap` στο `base.ll` αποτελείται από 12 εντολές (shifts και logical ops) για να αντιστρέψει τη σειρά των bytes. Στο `opt.ll`, όλη η λογική αντικαθίσταται από το `@llvm.bswap.i32`.

```
1 test_bswap:
2     movl    %edi, %eax
3     bswapl  %eax            ; Single instruction for Byte Swap
4     retq
```

Listing 4.10: Η εντολή BSWAP στο Assembly

4.3.3 Κατηγορία 3: Γραμμικοποίηση Ροής Ελέγχου

Στόχος εδώ είναι η εξάλειψη των διακλαδώσεων που προκύπτουν από εντολές επιλογής (`select`).

4.3.3.1 Ελάχιστο/Μέγιστο (Min/Max)

Στο `base.ll`, η συνάρτηση `@test_max` χρησιμοποιεί την εντολή `select` βασισμένη σε σύγκριση (`icmp sgt`). Το πέρασμά μας το μετατρέπει στο intrinsic `@llvm.smax`.

Baseline IR
`%cmp = icmp sgt i32 %a, %b` `%cond = select i1 %cmp, i32 %a, i32 %b`
Optimized IR
`%0 = call i32 @llvm.smax.i32(i32 %a, i32 %b)`

ΠΙΝΑΚΑΣ 4.2: Σύγκριση της Ενδιάμεσης Αναπαράστασης (IR) πριν και μετά τη βελτιστοποίηση για την εύρεση μέγιστης τιμής.

Αυτό επιτρέπει στο backend να παράγει κώδικα χωρίς άλματα (branchless), χρησιμοποιώντας την εντολή **Conditional Move (cmov)**.

```
test_max:
    movl    %esi, %eax
    cmpl    %esi, %edi
    cmovge  %edi, %eax     ; Move if Greater or Equal (No Branch)
    retq
```

4.3.3.2 Απόλυτη Τιμή (Abs)

Η συνάρτηση `@test_abs` υλοποιεί την απόλυτη τιμή με έλεγχο προσήμου (`if (x < 0)`). Στο Baseline IR, αυτό εμφανίζεται ως σύγκριση (`icmp`) ακολουθούμενη από επιλογή (`select`). Το PatternSelectPass το αντικαθιστά με το intrinsic `llvm.abs`.

Κώδικας ˜:

```
int test_abs(int x) { if (x < 0) return -x; return x; }
```

Baseline IR

```
  %cmp = icmp slt i32 %x, 0
  %neg = sub nsw i32 0, %x
  %cond = select i1 %cmp, i32 %neg, i32
     %x

```

Optimized IR

```
  %0 = call i32 @llvm.abs.i32(i32 %x,
    i1 false)

```

ΠΙΝΑΚΑΣ 4.3: Αντικατάσταση διακλάδωσης με intrinsic απολύτου τιμής.

Αποτέλεσμα Ασσεμβλυ: Το backend παράγει κώδικα χωρίς άλματα, χρησιμοποιώντας εντολές διαχείρισης προσήμου (π.χ. `neg` και `cmov`).

```
test_abs:
    movl    %edi, %eax
    negl    %eax
```

```
    cmovs    %edi, %eax     ; Conditional Move if Sign (Negative)
    retq
```

4.3.4 Κατηγορία 4: Αλγοριθμική Αντικατάσταση (PopCount)

Η πιο σύνθετη περίπτωση είναι ο αλγόριθμος SWAR στη συνάρτηση `@test_popcount_logic`. Εδώ αντικαθιστούμε έναν ολόκληρο αλγόριθμο $O(\log N)$ εντολών.

4.3.4.1 Σύγκριση IP

Το αρχικό IR είναι εξαιρετικά πυκνό και δυσανάγνωστο, περιέχοντας πολλαπλές αφαιρέσεις και ολισθήσεις.

```
; --- Base IR (SWAR) ---
  %shr = lshr i32 %i, 1
  %and = and i32 %shr, 1431655765
  %sub = sub i32 %i, %and
  %and1 = and i32 %sub, 858993459
  ; ... (                                                ) ...

; --- Optimized IR ---
  %0 = call i32 @llvm.ctpop.i32(i32 %i)
```

Listing 4.11: Απόσπασμα από το Base IR vs Optimized IR για PopCount

4.3.4.2 Αποτέλεσμα Assembly

Η αντικατάσταση δεκάδων εντολών από την εντολή `popcnt` αποδεικνύει την ικανότητα του συστήματος να γεφυρώνει το χάσμα μεταξύ του κώδικα υψηλού επιπέδου και του εξειδικευμένου υλικού.

```
test_popcount_logic:
    popcntl %edi, %eax     ; Hardware Population Count
    retq
```

Listing 4.12: Τελικός κώδικας για PopCount

4.4 Ποσοτικά Αποτελέσματα (Quantitative Results)

Η αξιολόγηση ολοκληρώνεται με την παρουσίαση των ποσοτικών μετρήσεων. Εστιάζουμε σε δύο άξονες: τη στατική μείωση του μεγέθους του κώδικα (Code Size Reduction) και την επίδραση στον χρόνο εκτέλεσης (Runtime Performance).

4.4.1 Στατική Μείωση Κώδικα (IR Instruction Count)

Η πιο σημαντική επίδοση του PatternSelectPass εντοπίζεται στη συμπύκνωση της Ενδιάμεσης Αναπαράστασης. Καθώς αντικαθιστούμε πολλαπλές εντολές χαμηλού επιπέδου με μία κλήση intrinsic, αναμένουμε σημαντική μείωση στον συνολικό αριθμό εντολών.

Χρησιμοποιώντας το σενάριο `evaluate.py`, μετρήσαμε τις εντολές IR για το σύνολο των θετικών σεναρίων (`positive_tests.c`). Συγκρίναμε τρεις εκδοχές:

1. **Baseline:** Κώδικας μόνο με `mem2reg` και `simplifycfg`.
2. **InstCombine:** Ο προεπιλεγμένος βελτιστοποιητής του LLVM.
3. **PatternSelect (Our Pass):** Η δική μας υλοποίηση ακολουθούμενη από `DCE`.

Στρατηγική (Strategy)	**Αρ. Εντολών (Inst. Count)**	**Μείωση (Reduction)**
Baseline	51	-
InstCombine	35	31.4%
PatternSelect + DCE	**24**	**52.9%**

Πινακας 4.4: Σύγκριση μείωσης εντολών IP. Το πέρασμά μας επιτυγχάνει μείωση άνω του 50%.

Όπως φαίνεται στον Πίνακα 4.4, το σύστημά μας επιτυγχάνει μείωση της τάξης του **52.9%**, ξεπερνώντας το γενικού σκοπού `InstCombine` στα συγκεκριμένα ιδιώματα. Αυτό επιβεβαιώνει ότι η στοχευμένη αναγνώριση προτύπων είναι ανώτερη από τις γενικές απλοποιήσεις όταν πρόκειται για σύνθετους αλγορίθμους όπως το PopCount ή το Rotate.

4.4.2 Μετρήσεις Χρόνου Εκτέλεσης (Runtime Benchmarks)

Για τη μέτρηση της ταχύτητας, εκτελέσαμε μικρο-μετρήσεις (micro-benchmarks) για κάθε ιδίωμα ξεχωριστά, εκτελώντας 100 εκατομμύρια επαναλήψεις σε βρόχο. Οι χρόνοι μετρήθηκαν σε νανο-δευτερόλεπτα ανά λειτουργία (ns/op).

Ιδίωμα (Idiom)	**Baseline (ns/op)**	**Optimized (ns/op)**	**Speedup**
MulPow2 (*8)	0.91	**0.81**	**1.12x**
ModPow2 (%32)	0.76	0.80	≈ 1.00x
Rotate	0.91	0.92	≈ 1.00x
Abs	0.99	1.08	0.92x
MinMax	0.98	0.98	1.00x
BSwap	0.93	0.93	1.00x
PopCount	1.49	**1.46**	**1.02x**

Πινακας 4.5: Χρόνοι εκτέλεσης σε περιβάλλον Intel i7-1165G7.

4.4.2.1 Ερμηνεία Αποτελεσμάτων

Παρατηρούμε τρεις κατηγορίες συμπεριφοράς:

- **Βελτίωση (Speedup):** Στην περίπτωση του **MulPow2**, η αντικατάσταση του πολλαπλασιασμού με ολίσθηση έδωσε ξεκάθαρη βελτίωση **12%**. Στο **PopCount**, παρόλο που η διαφορά φαίνεται μικρή (0.03ns), είναι σημαντική δεδομένου ότι ο αρχικός κώδικας SWAR ήταν ήδη εξαιρετικά βελτιστοποιημένος σε επίπεδο ". Η εντολή υλικού `popcnt` προσφέρει χαμηλότερη λανθάνουσα καθυστέρηση (latency).
- **Ισοδύναμη Απόδοση (Neutral):** Στα Rotate, MinMax, BSwap, οι χρόνοι είναι πρακτικά ταυτόσημοι. Αυτό οφείλεται στην αρχιτεκτονική του επεξεργαστή Tiger Lake. Οι σύγχρονες "ΠΥ διαθέτουν επιθετική εκτέλεση εκτός σειράς (Out-of-Order execution) και ισχυρούς προβλεπτές διακλαδώσεων (Branch Predictors). Έτσι, καταφέρνουν να εκτελούν τον "χειρότερο' κώδικα (π.χ. shifts + OR) σχεδόν τόσο γρήγορα όσο την εξειδικευμένη εντολή (rol), κρύβοντας την καθυστέρηση. Σε αυτές τις περιπτώσεις, το κέρδος μας είναι η **μείωση του μεγέθους του εκτελέσιμου** και η **κανονικοποίηση** του κώδικα, όχι η ωμή ταχύτητα.
- **Θόρυβος Μέτρησης (Measurement Noise):** Οι μικρές αποκλίσεις στα Abs και ModPow2 (της τάξης των 0.04 - 0.09 ns) εμπίπτουν στα όρια του στατιστικού σφάλματος και του κόστους του βρόχου μέτρησης (loop overhead). Δεν υποδεικνύουν πραγματική επιβράδυνση, αλλά ισοδύναμη συμπεριφορά.

4.4.3 Σύνοψη Κεφαλαίου

Η αξιολόγηση απέδειξε ότι το PatternSelectPass είναι εξαιρετικά αποδοτικό στη συμπίεση της IR, επιτυγχάνοντας μείωση εντολών κατά **52.9%**. Σε επίπεδο εκτέλεσης, διασφαλίζει

ότι χρησιμοποιούνται οι βέλτιστες εντολές μηχανής (intrinsics), προσφέροντας επιτάχυνση σε αριθμητικές πράξεις, ενώ διατηρεί την απόδοση σταθερή στις υπόλοιπες περιπτώσεις, παράγοντας πιο συμπαγή και αναγνώσιμο κώδικα μηχανής.

4.5 Επιβάρυνση Μεταγλώττισης (Compilation Overhead)

Ένας κρίσιμος παράγοντας για την υιοθέτηση οποιασδήποτε βελτιστοποίησης σε έναν μεταγλωττιστή παραγωγής είναι ο χρόνος που προσθέτει στη διαδικασία μεταγλώττισης. Ένα 'βαρύ' πέρασμα θα μπορούσε να αυξήσει σημαντικά τον χρόνο αναμονής του προγραμματιστή (build time).

Για να μετρήσουμε αυτή την επιβάρυνση, χρησιμοποιήσαμε τη σημαία `-time-passes` του LLVM κατά την εκτέλεση του `opt`.

4.5.1 Αποτελέσματα Χρονισμού

Οι μετρήσεις στο σύστημα αναφοράς (Intel Core i7-1165G7) έδειξαν ότι το PatternSelectPass εισάγει αμελητέα καθυστέρηση. Συγκεκριμένα, ο χρόνος εκτέλεσης του περάσματος για τη σουίτα δοκιμών ήταν της τάξης των 0.0002 **δευτερολέπτων** ($200\mu s$).

Αυτό το αποτέλεσμα είναι αναμενόμενο λόγω της σχεδιαστικής πολυπλοκότητας του αλγορίθμου:

- **Γραμμική Πολυπλοκότητα $O(N)$:** Το πέρασμα διατρέχει τις εντολές κάθε συνάρτησης μία μόνο φορά (single traversal).
- **Τοπική Ανάλυση:** Οι έλεγχοι ταύτισης (pattern matching) εξετάζουν μόνο τη τρέχουσα εντολή και τους άμεσους τελεστέους της, χωρίς να απαιτούν ακριβή διασχίση ολόκληρου του γράφου εξαρτήσεων (dependency graph).

4.5.2 Συμπέρασμα

Η σχέση κόστους-οφέλους κρίνεται εξαιρετικά θετική. Με μηδενική πρακτικά επιβάρυνση στον χρόνο μεταγλώττισης, επιτυγχάνουμε μείωση του μεγέθους κώδικα (IR) κατά 52.9% και αξιοποίηση εξειδικευμένου υλικού, καθιστώντας το PatternSelectPass κατάλληλο για ένταξη στη μόνιμη ροή βελτιστοποίησης (`-O2` ή `-O3`).

Κεφάλαιο 5

Συμπεράσματα & Προτάσεις για Μελλοντική Έρευνα

5.1 Σύνοψη και Συμπεράσματα

Ο πρωταρχικός στόχος αυτής της διατριβής ήταν να διερευνήσει τις δυνατότητες της υποδομής LLVM στη βελτιστοποίηση αναπαραστάσεων κώδικα υψηλού επιπέδου, γεφυρώνοντας το λεγόμενο **¨Σημασιολογικό Χάσμα' (Semantic Gap)** μεταξύ της πρόθεσης του προγραμματιστή και της εκτέλεσης στο υλικό.

Στο πλαίσιο της εργασίας, σχεδιάσαμε και υλοποιήσαμε το `PatternSelectPass`, ένα εξειδικευμένο πέρασμα βελτιστοποίησης ενσωματωμένο στον Νέο Διαχειριστή Περασμάτων (New Pass Manager). Εντοπίσαμε με επιτυχία 7 πολύπλοκα αριθμητικά ιδιώματα — συμπεριλαμβανομένων των Περιστροφής Bit (Rotate), Καταμέτρησης Πληθυσμού (PopCount) και Ακέραιας Απόλυτης Τιμής (Abs) — και τα αντικαταστήσαμε με intrinsics φιλικά προς το υλικό.

Η πειραματική μας αξιολόγηση κατέδειξε ότι αυτή η προσέγγιση επιτυγχάνει:

- **Μείωση Κώδικα:** Μείωση της τάξεως του **52.9%** στον στατικό αριθμό εντολών IR για τα στοχευμένα benchmarks.
- **Απλοποίηση Ροής:** Εξάλειψη περιττών διακλαδώσεων και ïσοπέδωση' (linearization) του Γράφου Ροής Ελέγχου (CFG), μειώνοντας την πίεση στον Μηχανισμό Πρόβλεψης Διακλαδώσεων.
- **Ανεξαρτησία Στόχου:** Παραγωγή φορητής IR που επιτρέπει στο backend να επιλέξει τη βέλτιστη εντολή μηχανής (π.χ. `POPCNT` για x86 ή `VCNT` για ARM).

Ο παρακάτω πίνακας συνοψίζει την αντιστοίχιση που επιτεύχθηκε από το σύστημά μας:

Ανιχνευθέν Μοτίβο (Input Pattern)	**LLVM Intrinsic**	**Στόχος x86 (Ενδεικτικός)**
SWAR Algorithm	`@llvm.ctpop`	`POPCNT`
Shift/OR Tree	`@llvm.fshl`	`ROL / ROR`
Select/ICmp (Abs)	`@llvm.abs`	`PABS / CDQ+XOR`
Byte Swapping Ops	`@llvm.bswap`	`BSWAP`

ΠΙΝΑΚΑΣ 5.1: Συνοπτική παρουσίαση των μετασχηματισμών και της αντιστοίχισης σε υλικό.

5.2 Διδάγματα και Προκλήσεις

Η ανάπτυξη ενός περάσματος μεταγλωττιστή εντός του οικοσυστήματος LLVM παρείχε πολύτιμες γνώσεις σχετικά με τη σύγχρονη μηχανική μεταγλωττιστών:

1. **Η Δύναμη του `PatternMatch.h`:** Μάθαμε ότι η χρήση του δηλωτικού API ταύτισης είναι σημαντικά πιο ισχυρή από τη χειροκίνητη επιθεώρηση κωδικών πράξης. Επιτρέπει την περιγραφή της ¨μορφής' του δέντρου εντολών (π.χ. `m_Add(m_Shl(...), ...)`) με τρόπο που είναι ανθεκτικός σε μικρές αλλαγές της IR.
2. **Διαχείριση Μορφής SSA:** Η κατανόηση της Στατικής Μονής Ανάθεσης ήταν κρίσιμη. Η ορθή χρήση των μεθόδων `replaceAllUsesWith` (RAUW) είναι ζωτικής σημασίας για την αυτόματη ενημέρωση του γράφου ροής δεδομένων (Data Flow Graph), διασφαλίζοντας ότι δεν μένουν ¨κρεμασμένοι' δείκτες σε διαγραμμένες εντολές.
3. **Ισορροπία Κανονικοποίησης:** Διαπιστώσαμε ότι υπάρχει μια λεπτή ισορροπία μεταξύ της κανονικοποίησης που κάνει το `InstCombine` και της αναγνώρισης ιδιωμάτων. Το δικό μας πέρασμα πρέπει να τρέξει στο σωστό σημείο της διοχέτευσης (pipeline), πριν το `InstCombine` αλλοιώσει τη δομή των πολύπλοκων μοτίβων.

5.3 Περιορισμοί της Τρέχουσας Υλοποίησης

Ενώ το `PatternSelectPass` καταδεικνύει την αποτελεσματικότητα της προσέγγισης, η τρέχουσα υλοποίηση υπόκειται σε συγκεκριμένους περιορισμούς:

- **Περιορισμένη Υποστήριξη Τύπων:** Οι matchers είναι επί του παρόντος βελτιστοποιημένοι κυρίως για ακεραίους 32-bit (`i32`). Αν και η λογική είναι γενική, η επέκταση σε άλλους τύπους (π.χ. `i64` ή `i16`) απαιτεί ρητή δήλωση ή χρήση template programming.

- **Έλλειψη Υποστήριξης Διανυσμάτων:** Το πέρασμα λειτουργεί μόνο σε βαθμωτές (scalar) τιμές. Δεν αναγνωρίζει μοτίβα εντός διανυσματικών τύπων (SIMD), όπως το `<4 x i32>`. Αυτό είναι σημαντικός περιορισμός για σύγχρονες εφαρμογές πολυμέσων ή μηχανικής μάθησης.
- **Ανάλυση Κόστους (Cost Modeling):** Το πέρασμα εφαρμόζει τον μετασχηματισμό ¨τυφλά¨, υποθέτοντας ότι το intrinsic είναι πάντα ταχύτερο. Σε σπάνιες περιπτώσεις (π.χ. παλαιότεροι επεξεργαστές χωρίς υποστήριξη `POPCNT`), η κλήση του intrinsic μπορεί να μετατραπεί σε κλήση βιβλιοθήκης λογισμικού, η οποία ενδέχεται να είναι πιο αργή.

5.4 Μελλοντικές Εργασίες

Η παρούσα διατριβή θέτει τα θεμέλια για περαιτέρω έρευνα. Προτείνονται οι ακόλουθες επεκτάσεις για την ενίσχυση της στιβαρότητας και της εμβέλειας του εργαλείου:

5.4.1 Διανυσματοποίηση και SIMD

Το σημαντικότερο επόμενο βήμα είναι η επέκταση των matchers για την υποστήριξη Διανυσματικών Τύπων. Αυτό θα επέτρεπε στο πέρασμα να βελτιστοποιεί λειτουργίες Μίας Εντολής Πολλαπλών Δεδομένων (SIMD), επιτρέποντας την παραγωγή ισχυρών εντολών όπως AVX-512 ή ARM NEON, που επεξεργάζονται πολλαπλά δεδομένα ταυτόχρονα.

5.4.2 Επαλήθευση Μέσω Ανάλυσης Assembly

Μια πολύτιμη επέκταση θα ήταν η διασύνδεση με το Backend. Θα μπορούσαμε να αναπτύξουμε ένα εργαλείο που αναλύει τον τελικό κώδικα μηχανής (Assembly) για να επιβεβαιώσει ότι τα intrinsics που εισάγουμε υποβιβάζονται όντως σε βέλτιστες εντολές και δεν καταλήγουν σε κλήσεις ρουτίνας (libcalls).

5.4.3 Αναγνώριση Μοτίβων με Μηχανική Μάθηση

Τέλος, μια καινοτόμα κατεύθυνση θα ήταν η αντικατάσταση των χειροκίνητων κανόνων (hand-written matchers) με μοντέλα Μηχανικής Μάθησης (ML). Χρησιμοποιώντας τεχνικές **Superoptimization**, θα μπορούσαμε να εκπαιδεύσουμε ένα μοντέλο να ανακαλύπτει αυτόματα νέα, άγνωστα ιδιώματα σε μεγάλες βάσεις κώδικα και να προτείνει βελτιστοποιήσεις που ο άνθρωπος-σχεδιαστής δεν έχει προβλέψει.

Βιβλιογραφία

[1] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization*, page 75. IEEE, 2004.

[2] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2nd edition, 2012. Essential reference for bitwise algorithms and optimization idioms.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2006.

[4] Keith Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2nd edition, 2011.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[6] LLVM Project. *LLVM Language Reference Manual*, 2024. Accessed: 2024-05-20.

[7] LLVM Project. *Writing an LLVM Pass*, 2024. Accessed: 2024-05-20.

[8] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 1997.

[9] LLVM Project. *LLVM's Analysis and Transform Passes: InstCombine*, 2024. Accessed: 2024-05-20.

[10] Henry Massalin. Superoptimizer: a look at the smallest program. In *Proceedings of the second international conference on Architectural support for programming languages and operating systems*, pages 122–126, 1987.

[11] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Stephen McCamant, Zhendong Su12 Lewis, and John Regehr. Souper: A synthesizing superoptimizer. In *arXiv preprint arXiv:1711.04422*, 2017.

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1.* Pearson Education, 2011. Section 7.1.3: Bitwise Tricks and Techniques.

[13] William M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2023. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.

[15] T. Northover and Q. Colombet. Globalisel: A new instruction selection framework for llvm. In *2016 LLVM Developers' Meeting*. LLVM Foundation, 2016.