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

ΕΠΕΚΤΑΣΗ ΤΟΥ ΜΕΤΑΓΛΩΤΤΙΣΤΗ HiPE ΓΙΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ARM 64-BIT

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΘΑΝΑΣΙΟΣ Ι. ΠΑΠΟΥΤΣΙΔΑΚΗΣ

Επιβλέπων: Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Αθήνα, Οκτώβριος 2018

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

ΕΠΕΚΤΑΣΗ ΤΟΥ ΜΕΤΑΓΛΩΤΤΙΣΤΗ HiPE ΓΙΑ ΑΡΧΙΤΕΚΤΟΝΙΚΗ ARM 64-BIT

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΘΑΝΑΣΙΟΣ Ι. ΠΑΠΟΥΤΣΙΔΑΚΗΣ

Επιβλέπων: Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15/10/2018.

.....	.....	.....
Νικόλαος Παπασπύρου	Νεκτάριος Κοζύρης	Γεώργιος Γκούμας
Αναπληρωτής Καθηγητής	Καθηγητής	Επίκουρος Καθηγητής

Αθήνα, Οκτώβριος 2018

.....

Αθανάσιος Ι. Παπουτσιδάκης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικών Υπολογιστών Ε.Μ.Π.

Περίληψη

Ο σκοπός αυτής της διπλωματικής εργασίας είναι η επέκταση του μεταγλωττιστή HiPE της υλοποίησης OTP της συναρτησιακής γλώσσας προγραμματισμού Erlang, για την παραγωγή εκτελέσιμων προγραμμάτων σε κώδικα μηχανής αρχιτεκτονικής aarch64. Παράλληλα μελετήσαμε τις διαφορές ανάμεσα στην αρχιτεκτονική aarch64 και την προηγούμενή της, την αρχιτεκτονική arm, καθώς και τη διαδικασία που μπορεί ένας κατασκευαστής μεταγλωττιστών να ακολουθήσει για να επεκτείνει ένα γενικό μεταγλωττιστή αρχιτεκτονικής arm για αρχιτεκτονική aarch64. Με αυτή την εργασία οι συσκευές που είναι εξοπλισμένες με επεξεργαστή armv8 ή νεότερο θα είναι ικανές να εκτελέσουν προγράμματα Erlang μεταφρασμένα απευθείας σε κώδικα μηχανής, αυξάνοντας έτσι την ταχύτητα εκτέλεσης και μειώνοντας την κατανάλωση ενέργειας. Επιπλέον, προγραμματιστές που προσαρμόζονται στη νέα αρχιτεκτονική θα έχουν τη δυνατότητα να ανατρέξουν στην ανάλυση των διαφορών που συλλέχθηκαν κατά τη διάρκεια της εργασίας, ώστε να προλάβουν μελλοντικά σφάλματα και απρόβλεπτες δυσκολίες στις υλοποιήσεις τους.

Η aarch64 είναι αρχιτεκτονική τύπου RISC, μετεξέλιξη της υπάρχουσας αρχιτεκτονικής arm, και σταδιακά εισάγεται σε πλήθος νέων συσκευών, όσο η βιομηχανία παράγει αποδοτικότερους επεξεργαστές που την χρησιμοποιούν. Υλοποιήσεις της υπάρχουσας αρχιτεκτονικής arm υπάρχουν σε αρκετές γλώσσες προγραμματισμού, και οι διαφορές ανάμεσα στις δύο αρχιτεκτονικές είναι όχι σημαντικά πολλές, έτσι ώστε να είναι όσο το δυνατόν ευκολότερη η μετατροπή και επέκταση των υλοποιήσεων.

Η υλοποίηση OTP της γλώσσας Erlang παρέχει ένα οργανωμένο περιβάλλον εκτέλεσης, το οποίο ενεργοποιώντας τον μεταγλωττιστή HiPE είναι ικανό να παράξει και να εκτελέσει κατά παραγγελία κώδικα μηχανής για πληθώρα αρχιτεκτονικών. Τα τμήματα του HiPE που παράγουν κώδικα για κάθε αρχιτεκτονική είναι αυστηρά οργανωμένα στην υλοποίηση και κατά το πλείστον αυτόνομα, δίνοντάς μας τη δυνατότητα να κατασκευάσουμε το τμήμα μεταγλωττιστή για την αρχιτεκτονική aarch64 συμβουλευόμενοι το υπάρχον τμήμα του HiPE για την παρόμοια αρχιτεκτονική arm.

Λέξεις - κλειδιά

Μεταγλωττιστές γλωσσών προγραμματισμού, αρχιτεκτονική aarch64, επεξεργαστές RISC, γλώσσα προγραμματισμού Erlang, ανάπτυξη λογισμικού βασισμένη σε δοκιμές, παραγωγή κώδικα μηχανής, διεπαφή με περιβάλλοντα εκτέλεσης.

Abstract

The purpose of this thesis project is the adaptation of the HiPE compiler of the OTP implementation of the Erlang programming language, for the production of executables in machine code of aarch64 architecture. Concurrently we studied the differences between the aarch64 architecture and its predecessor, the arm architecture, as well as the procedure a compiler implementer may follow to adapt a generic arm compiler for the aarch64 architecture. With this effort the devices equipped with an armv8 or newer processor will become able to execute Erlang applications directly translated to machine code, thus increasing their execution speed and reducing the energy consumption. Moreover, programmers adapting to the new architecture will be able to consult the analysis of the differences collected during the project, so that they may avoid future errors and unexpected difficulties during their implementations.

Aarch64 is a RISC type architecture, a successor to the existing arm architecture, and is gradually introduced into a plethora of new devices, as the industry produces more efficient processors for it. Implementations of the existing arm architecture exist in various programming languages, and the differences between the two architectures are not too many, so that adapting and augmenting implementations would be as easy as possible.

The OTP implementation of the Erlang language provides an organized runtime system, which by activating the HiPE compiler is able to produce and execute on demand machine code for a variety of architectures. The HiPE backends that produce code for each architecture are definitively organized within the implementation and mostly autonomous, granting us the ability to construct the compiler backend for the aarch64 architecture by consulting the existing backend of HiPE for the similar arm architecture.

Keywords

Programming language compilers, aarch64 architecture, RISC processors, Erlang programming language, test-driven development, machine code production, interfacing with runtime systems.

Ευχαριστίες

Ευχαριστώ τον καθηγητή μου, κ. Κωνσταντίνο Σαγώνα, για τη συνεχή και απαραίτητη επίβλεψη και καθοδήγηση καθ' όλη τη διάρκεια της εργασίας μου.
Την ομάδα ανάπτυξης της υλοποίησης Erlang/OTP για το υπέροχο και καλά οργανωμένο αυτό πρόγραμμα ανοιχτού κώδικα.
Το εργαστήριο τεχνολογίας λογισμικού της σχολής Ηλεκτρολόγων Μηχανικών - Μηχανικών Υπολογιστών του ΕΜΠ, καθώς και το πανεπιστήμιο της Uppsala.
Τις εταιρίες ARM, Broadcom και Raspberry Pi Foundation για την αρχιτεκτονική AArch64 και τα παράγωγα νέα προϊόντα.
Τα κοντινά μου πρόσωπα για τα απαραίτητα διαλείμματα.

Αθανάσιος Ι. Παπουτσιδάκης
12/10/2018

Περιεχόμενα

Κατάλογος Πινάκων

Κατάλογος Σχημάτων

1. Εισαγωγή

1.1. Σκοπός

Σκοπός αυτής της εργασίας είναι η κατασκευή ενός νέου backend, τμήματος παραγωγής κώδικα μηχανής, για το μεταγλωττιστή HiPE, το οποίο θα παράγει δυαδικό κώδικα για τη νέα αρχιτεκτονική aarch64. Η αρχιτεκτονική aarch64 είναι η επόμενη της RISC αρχιτεκτονικής arm, η οποία χρησιμοποιείται ευρέως σε ενσωματωμένα συστήματα, και γενικά στις φορητές συσκευές. Ο μεταγλωττιστής HiPE αποτελεί τμήμα της υλοποίησης OTP της γλώσσας προγραμματισμού Erlang, και χρησιμεύει ώστε να είναι δυνατόν να εκτελούνται τα προγράμματα Erlang όχι μέσω διερμηνέα (interpreter) ή προσομοιωμένο δυαδικό κώδικα (bytecode) αλλά απευθείας από το hardware σε μορφή κώδικα μηχανής, βελτιώνοντας την απόδοση και την κατανάλωση ενέργειας. Στην πορεία θα καταγράψουμε τα βήματα που ακολουθήσαμε καθώς και τις ιδιαιτερότητες της αρχιτεκτονικής που εντοπίσαμε, ώστε να γίνουν εμφανή τα σημεία που πρέπει να προσέξει μία υλοποίηση κατά την ενσωμάτωση της νέας αρχιτεκτονικής. Τελικός στόχος είναι το νέο τμήμα να παράγει έγκυρο κώδικα μηχανής από προγράμματα Erlang, ο οποίος να εκτελείται με ακρίβεια και χωρίς σφάλματα από μηχανήματα αρχιτεκτονικής aarch64. Επίσης στοχεύουμε η παρούσα εργασία να αποτελέσει χρήσιμο οδηγό για προγραμματιστές που εργάζονται με ή θέλουν να ερευνήσουν τη νέα αρχιτεκτονική, ή και να κατανοήσουν τη δομή της υλοποίησης OTP για παρόμοια μελλοντική εργασία.

1.2. Ιστορία αρχιτεκτονικών RISC

Τον πρώτο καιρό της εμφάνισης των υπολογιστών, η μνήμη ήταν ένα από τα ακριβά κομμάτια του συστήματος. Αυτό σε συνδυασμό με το ότι δεν υπήρχαν γλώσσες προγραμματισμού υψηλού επιπέδου οδήγησε στη δημιουργία αρχιτεκτονικών επεξεργαστών, των οποίων οι εντολές ήταν κατασκευασμένες ώστε η καθεμία να μπορεί να πραγματοποιεί πολλές σύνθετες λειτουργίες, ώστε να μειωθεί η προσπέλαση μνήμης και έτσι να αυξηθεί η απόδοση και η ευκολία προγραμματισμού. Οι αρχιτεκτονικές αυτές επικράτησαν, και έγιναν γνωστές με το όνομα CISC (Complex Instruction Set Computer). Απόγονοι αυτών είναι οι δημοφιλείς αρχιτεκτονικές τύπου x86, οι οποίες εκμεταλλεύονται αυτή την ιδιότητα της φιλοσοφίας CISC ώστε να μεγιστοποιήσουν την ταχύτητα του συστήματος. Κατά τη δεκαετία όμως του 1970, η τιμή της προσωρινής μνήμης άρχισε να πέφτει και η ταχύτητα να αυξάνεται, και με τη βοήθεια της μνήμης cache ο επεξεργαστής έγινε ικανός να μπορεί να διαβάζει τις εντολές απευθείας από τη μνήμη χωρίς ιδιαίτερη καθυστέρηση. Έτσι, υποβοηθούμενες και από την εμφάνιση των μεταγλωττιστών, οι οποίοι ήταν ικανοί να παράξουν σύνθετες ακολουθίες εντολών από απλές γλώσσες προγραμματισμού, άρχισαν να εμφανίζονται νέου τύπου αρχιτεκτονικές, οι οποίες είχαν στόχο όχι την αύξηση των δυνατοτήτων των εντολών, αλλά την απλούστευση και έτσι βελτιστοποίηση του υλικού, με εντολές πιο εύκολα και γρήγορα επεξεργάσιμες από το λογικό κύκλωμα του επεξεργαστή. Οι αρχιτεκτονικές αυτές ονομάστηκαν RISC (Reduced Instruction Set Computer). Τα τελευταία χρόνια, με την εμφάνιση των φορητών συσκευών, όπου η ταχύτητα δεν είναι τόσο σημαντικό όφελος σε σχέση με τη χαμηλή κατανάλωση, η ηπιότερη

χρήση του υλικού από αυτές τις αρχιτεκτονικές έγινε αρκετά ελκυστικό προσόν για τους κατασκευαστές.

Στον αγώνα για τη βελτίωση της απόδοσης, τεχνάσματα που δημιουργήθηκαν κατά καιρούς άλλαξαν κατά πολύ τις συνήθειες στην κατασκευή των επεξεργαστών. Μία από τις πιο σημαντικές τεχνοτροπίες που εμφανίστηκαν είναι και η αρχιτεκτονική τύπου Harvard, η οποία αντικατέστησε κατά μεγάλο βαθμό την μέχρι τότε αρχιτεκτονική τύπου von Neumann και χρησιμοποιείται μέχρι και σήμερα. Η επιλογή αυτής της αρχιτεκτονικής είναι η οργάνωση των εντολών ενός προγράμματος σε ξεχωριστό τομέα μνήμης από αυτόν των δεδομένων, δίνοντας στον επεξεργαστή τη δυνατότητα να εκτελεί εντολές παράλληλα με την ανάγνωση των δεδομένων που αυτές χρειάζονται.

Παρά τη συνεχή αύξηση των εντολών ανά κύκλο (instructions per cycle) και της συχνότητας των επεξεργαστών ανά τους χρόνους, τις τελευταίες δεκαετίες έγινε εμφανές ότι η συχνότητα του επεξεργαστή δε θα μπορούσε να υπερβεί τα 3-5 GHz, όπου αρχίζουν να εμφανίζονται σχετικιστικά φαινόμενα, καθώς αυτή η συχνότητα ρολογιού επιτρέπει στο φως να μετακινηθεί μόλις μερικά εκατοστά ανά κύκλο. Ακόμα, οι εντολές ανά κύκλο δε θα μπορούσαν σε ένα επεξεργαστή που εκτελεί γραμμικά εντολές να υπερβούν τη 1 ipc. Η προφανής λύση ήταν ο διαμοιρασμός των εργασιών σε περισσότερους από έναν πυρήνες επεξεργαστή μέσα στο ίδιο σύστημα. Πλέον (2018) οι επεξεργαστές που υπάρχουν στην αγορά για προσωπικούς υπολογιστές δεν περιλαμβάνουν λιγότερους από δύο πυρήνες, και έχουν εμφανιστεί επεξεργαστές για απαιτητικά συστήματα με 32 πυρήνες.

Παράλληλα με την αύξηση της απόδοσης των επεξεργαστών, το λογισμικό έγινε αρκετά απαιτητικό και όσον αφορά την κατανάλωση προσωρινής μνήμης. Με όλο και περισσότερες συσκευές να διοχετεύουν δεδομένα στον παγκόσμιο ιστό, απαιτητικές εφαρμογές εμφανίστηκαν για την επεξεργασία και αξιοποίηση αυτών των δεδομένων. Ήταν φανερό ότι τα 4 GB μνήμης που ήταν ικανός να προσπελάσει ένας επεξεργαστής 32 bit δε θα ήταν επαρκή, και η υλοποίηση τεχνασμάτων για αύξηση μνήμης όπως σελιδοποίηση θα ήταν βάρος στον επεξεργαστή σε θέμα πρόσθετων εντολών, καθώς η λέξη μνήμης είναι ίσως ένα από τα πιο χρησιμοποιούμενα στοιχεία σε μια αρχιτεκτονική. Με βάση αυτό και άλλα οφέλη όπως η επεξεργασία μεγάλων αριθμών, εισήχθηκαν στην αγορά τα συστήματα με μέγεθος λέξης 64 bit.

Πρόσφατα, ένας νέος επεξεργαστής έκανε την άνοδό του στην αγορά. Ο 32-bit RISC επεξεργαστής arm με την ομώνυμη αρχιτεκτονική του τύπου Harvard, το 2007 ήταν ο επεξεργαστής στο 95% των νέου τύπου κινητών τηλεφώνων, και ο πιο δημοφιλής στα ενσωματωμένα συστήματα. Το 2011, η ARM Holdings ανακοίνωσε τη δημιουργία του armv8, την επόμενη γενιά του επεξεργαστή arm, ο οποίος έχει μήκος λέξης 64 bit και είναι εξοπλισμένος με τη νέα αρχιτεκτονική aarch64. Μετά από λίγο καιρό εμφανίστηκαν υλοποιήσεις αυτού του επεξεργαστή σταδιακά σε φορητές συσκευές, ενσωματωμένα συστήματα αλλά και μεγαλύτερα υπολογιστικά συστήματα, οι οποίες χρησιμοποιούν πολλαπλούς πυρήνες για αύξηση της απόδοσης.

1.3. Ιστορία συναρτησιακών γλωσσών προγραμματισμού

Καθώς εμφανίστηκαν οι γλώσσες προγραμματισμού, όταν εμφανίστηκε η ανάγκη για κατασκευή σύνθετων προγραμμάτων για τα οποία η εγγραφή σε κώδικα μηχανής θα ήταν απαιτητική, κάποιες από αυτές στόχευσαν σε μία έκφραση προγραμματισμού μακριά από την θεώρησή του ως τρόπου εισαγωγής εντολών σε ένα μηχάνημα, αλλά περισσότερο ως τρόπο αυστηρά μαθηματικής έκφρασης ενός προβλήματος και της λύσης του. Αυτό επέτρεψε δυνατότητες θεωρητικής και εν μέρει αυτοματοποιημένης ανάλυσης ενός προγράμματος πολύ πιο στιβαρές από ότι θα ήταν εύκολα δυνατόν σε προστακτικές γλώσσες, αυξάνοντας την ικανότητα του μεταγλωττιστή να αναγνωρίζει ή να υποψιάζεται συντακτικά αλλά και αρκετά λογικά λάθη κατά την ώρα της μεταγλώττισης, δίνοντας καλή εγγύηση σε ένα πρόγραμμα που πέρασε τη φάση της μεταγλώττισης ότι δεν θα παρουσιάσει σφάλματα εκτέλεσης.

Ένας θεωρητικός μηχανισμός που ήρθε να ενισχύσει αυτή την ικανότητα ήταν η κατασκευαστική θεωρία τύπων του Martin-Löf (1980). Αυτή, συσχετίζοντας τα συναρτησιακά προγράμματα με κατασκευαστικές αποδείξεις μέσω της έκφρασης με εξαρτώμενους τύπους, επηρέασε αρκετές μελλοντικές συναρτησιακές γλώσσες στο να έχουν συστήματα τύπων τα οποία να μπορούν να αποδείξουν κατά ένα βαθμό την ορθότητα ενός προγράμματος.

Καθώς κατά την εποχή που εμφανίστηκαν τα συστήματα πολλαπλών πυρήνων, διαπιστώθηκε ότι η ασφαλής έλλειψη παρενεργειών (side-effects) από τις συναρτησιακές γλώσσες θα ήταν ευεργετική για την παράλληλη εκτέλεση, πιο σύγχρονες συναρτησιακές γλώσσες ενσωμάτωσαν μηχανισμούς για την αυτόματη παραλληλοποίηση εκτέλεσης διεργασιών. Μία από τις συναρτησιακές γλώσσες που έκανε αυτή τη μετάβαση ήταν και η γλώσσα προγραμματισμού Erlang. Στοχευμένη αρχικά για τηλεφωνικά δίκτυα, η φιλοσοφία της για το ότι κάθε τι που εκτελείται είναι ξεχωριστή διεργασία ήταν ευνοϊκή για την παράλληλη εκτέλεση.

Καθώς οι συναρτησιακές γλώσσες προγραμματισμού είναι αρκετά υψηλού επιπέδου, είναι άξιο παρατήρησης ότι οι πιο δημοφιλείς από αυτές ξεκίνησαν ως διερμηνευόμενες (interpreted). Για την παραγωγή άλλωστε κώδικα μηχανής μία συναρτησιακή γλώσσα έχει κάπως πιο πολύπλοκη διαδικασία στο στάδιο της μετατροπής σε ενδιάμεσο κώδικα, καθώς οι προστακτικές γλώσσες είναι αρκετά κοντά στην ενδιάμεση γλώσσα (γλώσσα μεταφοράς μεταξύ καταχωρητών, register transfer language) ως άμεσοι απόγονοι της συμβολικής γλώσσας (assembly language). Αρκετές υλοποιήσεις συναρτησιακών γλωσσών απέκτησαν μεταγλωττιστές στην πορεία (βλ. clozureCL, MLton), και κάποιες προτίμησαν το ενδιάμεσο μονοπάτι του runtime system. Η Erlang επί παραδείγματι ως προκαθορισμένη ενέργεια μεταγλωττίζει το πρόγραμμα σε δικής της μορφής bytecode, τον οποίο το περιβάλλον εκτέλεσης στη συνέχεια εκτελεί μέσω ενός προσομοιωτή (emulator). Αυτό δίνει γενικά τη δυνατότητα εκτέλεσης σε πολλά διαφορετικά συστήματα με τον ίδιο τρόπο (multi-platform).

Χρήσιμο εργαλείο σε μία συναρτησιακή γλώσσα προγραμματισμού βρέθηκαν να είναι οι garbage collectors. Καθώς η διαχείριση μνήμης θεωρείται παρενέργεια, το ιδανικό είναι αυτή να γίνεται αόρατα από τον προγραμματιστή και αυτοματοποιημένα. Ο πρώτος καθαριστής μνήμης κατασκευάστηκε ακριβώς για μια συναρτησιακή γλώσσα, τη Lisp, το 1959 από τον McCarthy. Σε

διερμηνευόμενα προγράμματα αυτό γίνεται εύκολα, καθώς ο διερμηνέας αποφασίζει κατά την εκτέλεση των εντολών πότε είναι κατάλληλη στιγμή για καθαρισμό μνήμης. Σε προγράμματα υποστηριζόμενα από περιβάλλον εκτέλεσης χρειάζεται ο μεταγλωττιστής του bytecode να τοποθετήσει κλήσεις προς το runtime μέσα στο ίδιο το πρόγραμμα, σε κατάλληλα μελετημένα σημεία, οι οποίες θα επιτρέπουν στο runtime να καθαρίσει τη μνήμη αν αυτό χρειάζεται. Σε μεταγλωττιζόμενα προγράμματα ο διαχειριστής μνήμης πρέπει να ενσωματωθεί μέσα στο πρόγραμμα, για αυτό και αρκετοί μεταγλωττιστές συναρτησιακών γλωσσών είναι γνωστό ότι παράγουν κάπως ογκώδη προγράμματα - εκτός αν υποστηρίζονται εξωτερικά από frameworks και βιβλιοθήκες συστήματος (π.χ. F#).

Η γλώσσα λοιπόν που θα μελετήσουμε, η Erlang, είναι μία παράλληλη συναρτησιακή γλώσσα γενικού σκοπού, υποστηριζόμενη από περιβάλλον εκτέλεσης με διαχειριστή μνήμης. Ο μεταγλωττιστής HiPE με τον οποίο θα εργαστούμε προστέθηκε στην Erlang/OTP το 2001, και αναλαμβάνει τη μετατροπή του bytecode της Erlang σε κώδικα μηχανής, διατηρώντας τις κλήσεις προς το περιβάλλον εκτέλεσης και το διαχειριστή μνήμης.

1.4. Σύνοψη

Στα κεφάλαια που θα ακολουθήσουν θα δούμε τα εξής:

- Στο κεφάλαιο 2 θα μελετήσουμε τη νέα αρχιτεκτονική aarch64, τα είδη των εντολών και δομών της, τις ιδιότητες αρχιτεκτονικών RISC τύπου arm, και τις διαφορές που έχει από την προηγούμενή της, την arm.

- Στο κεφάλαιο 3 θα μελετήσουμε τη γλώσσα Erlang, και θα αναλύσουμε συνοπτικά τον κώδικα της υλοποίησης OTP, τον τρόπο λειτουργίας της, και πιο αναλυτικά τα σημεία που μας ενδιαφέρουν.

- Με τις προηγούμενες γνώσεις στο κεφάλαιο 4 θα καθορίσουμε τις ενέργειες που απαιτούνταν εποπτικά για να πετύχουμε το στόχο μας, καθώς και τη μεθοδολογία που ακολουθήσαμε για την υλοποίηση.

- Στο κεφάλαιο 5 θα παρουσιάσουμε τις αλλαγές που κάναμε, τα βήματα που ακολουθήσαμε, και θα τα εξηγήσουμε και θα τα αιτιολογήσουμε.

- Στο κεφάλαιο 6 θα δούμε τα αποτελέσματα των ενεργειών μας και το κατά πόσο αυτές οδήγησαν στο στόχο μας.

- Στο κεφάλαιο 7 θα βγάλουμε συμπεράσματα για το τι επίπτωση θα έχουν τα αποτελέσματα της εργασίας μας, και θα προτείνουμε παρεμφερή θέματα για σχετική μελλοντική έρευνα.

2. Η αρχιτεκτονική aarch64

Ως αρχιτεκτονική RISC, η aarch64 έχει στόχο της την απλότητα του υλικού. Οι αρχιτεκτονικές τύπου arm περιλαμβάνουν εντολές σταθερού μεγέθους 32 bit, ώστε να απλουστευτεί το στάδιο της αποκωδικοποίησης. Αυτή η ιδιότητα παρατηρείται και σε άλλα σημεία. Ο αριθμός των καταχωρητών είναι κατά προτίμηση δύναμη του 2, ώστε αναφορά σε καταχωρητή να παίρνει πάντα σταθερό αριθμό bits. Οι καταχωρητές είναι οι ίδιοι εντελώς ισοδύναμοι μεταξύ τους, με ελάχιστες εξαιρέσεις. Στις περισσότερες περιπτώσεις αυτός που δίνει επιπλέον ιδιότητες σε κάποιους καταχωρητές είναι το λειτουργικό σύστημα, σε συμφωνία με το Procedure Call Standard της κάθε αρχιτεκτονικής.

Συγκεκριμένα, στην αρχιτεκτονική arm υπάρχουν 16 καταχωρητές, οι οποίοι μπορούν να χρησιμοποιηθούν για κάθε χρήση. Ο μόνος ο οποίος έχει ιδιαίτερη ιδιότητα είναι ο r15, ο program counter, ο οποίος αυξάνεται μετά από την εκτέλεση κάθε εντολής, και ίσως ο r14, ο link register, ο οποίος γράφεται από την εντολή bl (κλήση συνάρτησης) με τη διεύθυνση επιστροφής. Το call standard χρησιμοποιεί εκ των υστέρων τον r13 ως stack pointer (η μορφή των εντολών επιτρέπει πρακτικά σε οποιονδήποτε καταχωρητή να χρησιμοποιηθεί ως stack pointer). Από την άλλη η aarch64 αφαιρεί τον program counter εντελώς, και κάνει τον SP (x31) ιδιαίτερο καταχωρητή, ο οποίος μπορεί να γραφτεί σε ορισμένες μόνο περιπτώσεις.

Η μετατροπή αυτή του stack pointer σε ειδικό καταχωρητή του προσέδωσε στην aarch64 και άλλη μία ιδιαιτερότητα. Ενώ στην arm κανείς μπορούσε να διαχειριστεί τον r13 ως οποιοδήποτε άλλο καταχωρητή, ο SP στην aarch64 είναι πάντα στοιχισμένος (aligned) στα 4 bit. Αυτό σημαίνει ότι η κορυφή της στοίβας (stack top) επιτρέπεται να λαμβάνει μόνο τιμές πολλαπλάσιες του 16. Συνεπώς, εάν κανείς χρησιμοποιήσει τον SP της αρχιτεκτονικής για δείκτη στοίβας, οποιαδήποτε αύξηση ή μείωσή του κατά λιγότερο από 16 byte θα αποτύχει, οδηγώντας σε διαφθορά της στοίβας. Η aarch64 παρέχει τις εντολές ldp και stp (load και store pair) οι οποίες είναι ικανές να αποθηκεύσουν 2 words των 64 bit στη μνήμη και να αυξήσουν το δείκτη κατά 16. Σε περίπτωση που τα δεδομένα που πρέπει να αποθηκευτούν στη στοίβα έχουν μέγεθος όχι πολλαπλάσιο του 16, τότε γεμίζουμε τη στοίβα με κενά (padding) ώστε αυτό να γίνει ακέραιο πολλαπλάσιο, και προσέχουμε τα επόμενα access στη στοίβα να λαμβάνουν υπόψη τα κενά που υπάρχουν στην κορυφή.

Η κατάργηση του r15, program counter, άλλαξε αρκετά δεδομένα στην κλήση συναρτήσεων. Καταρχάς ο συνήθης τρόπος επιστροφής από συνάρτηση, η αντικατάσταση των δεδομένων του PC από τα περιεχόμενα του link register ή άλλον, δεν μπορεί πλέον να χρησιμοποιηθεί. Η επιστροφή είναι τώρα δυνατή με τη χρήση της νέας εντολής ret, η οποία παίρνει προαιρετικά έναν register ως όρισμα (προεπιλογή ο x30, LR) και συνεχίζει την εκτέλεση από τη διεύθυνση που είναι γραμμένη μέσα του. Επίσης για διακλάδωση της εκτέλεσης και για κλήση σε άγνωστη διεύθυνση (αποθηκευμένη σε register) υπάρχουν αντί της απευθείας εγγραφής του PC οι εντολές br και blr (branch to register και branch with link to register). Εάν επιπλέον κανείς χρειάζεται τη διεύθυνση του PC σε κάποιο σημείο, είναι διαθέσιμες οι εντολές adr και adrp, οι οποίες προσθέτουν τη διεύθυνση του PC στην οποία βρίσκονται σε ένα προαιρετικό offset και

αποθηκεύουν το αποτέλεσμα σε register. Υπόψη ότι, ενώ στην arm τα περιεχόμενα του PC ήταν η διεύθυνση της τρέχουσας εντολής + 2 λέξεις κώδικα (2x4 bytes), στην aarch64 αυτό έχει καταργηθεί, οπότε τα περιεχόμενα του PC είναι ακριβώς η διεύθυνση της εντολής που τον χρησιμοποιεί.

Επί του θέματος της κλήσης συναρτήσεων, είναι γνωστό ότι σε ορισμένες περιπτώσεις κάποιες συναρτήσεις είναι πολύ μακριά στη μνήμη για να μπορέσει να έχει πρόσβαση μία εντολή με απευθείας όρισμα διεύθυνσης (immediate offset). Η τακτική που χρησιμοποιείται σε αυτές τις περιπτώσεις είναι η δημιουργία μικρών κομματιών κώδικα, λεγόμενα veneers ή trampolines, τα οποία βρίσκονται επαρκώς κοντά στην καλούσα συνάρτηση ώστε να μπορεί να τα καλέσει με immediate offset, και έχουν λειτουργία να φορτώνουν την πλήρη διεύθυνση σε ένα register και να εκτελούν άλμα. Τα trampolines αποτελούνται από δύο βασικά τμήματα, τον κώδικα φόρτωσης και άλματος και χώρο για την πλήρη διεύθυνση, την οποία θα γράψει (patching) κατά την φόρτωση του προγράμματος ο φορτωτής (loader).
Στην arm, η διεύθυνση έπιανε 4 bytes, και η φόρτωση και το άλμα γινόταν με μία απλή ldr (pc-relative) από την επόμενη θέση μνήμης απευθείας στον καταχωρητή r15 (PC), οπότε το trampoline είχε μέγεθος 2 λέξεις κώδικα (2x4 bytes). Στην aarch64 αυτή η δυνατότητα απευθείας φόρτωσης του PC από τη μνήμη δεν υπάρχει, οπότε πρέπει να γίνει φόρτωση σε ενδιάμεσο καταχωρητή και μετά br σε αυτό τον καταχωρητή. Ακόμα, η διεύθυνση είναι 64-bit, οπότε πιάνει 8 bytes. Το trampoline τελικά αποκτά μέγεθος 4 λέξεις κώδικα (16 bytes). Ευτυχώς το Procedure Call Standard ορίζει τους καταχωρητές x16 και x17 ως προσωρινούς, και τους προτείνει ειδικά για αυτές τις περιπτώσεις. Επιπλέον, οι εντολές άλματος με immediate offset πλέον έχουν αρκετό χώρο για άλματα μήκους 128 MiB, οπότε η χρήση των trampolines σε aarch64 έχει γίνει αρκετά περιορισμένη.

Ως Harvard αρχιτεκτονικές, οι arm και aarch64 έχουν ξεχωριστή cache για τα δεδομένα και τις εντολές. Αυτό παρουσιάζει το εξής πρόβλημα στις υλοποιήσεις μεταγλωττιστών σε περιβάλλοντα εκτέλεσης: όσο το πρόγραμμα είναι σε φάση μεταγλώττισης, αποτελεί δεδομένα, και άρα αποθηκεύεται και επεξεργάζεται στην cache δεδομένων. Μόλις όμως έρθει η στιγμή της εκτέλεσης, ο επεξεργαστής για να μπορέσει να εκτελέσει το πρόγραμμα πρέπει αυτό να βρίσκεται στην cache εντολών. Η υλοποίηση θα πρέπει να προσέξει να προβεί πριν την εκτέλεση σε cache flushing, δηλαδή να μεταφέρει τον κώδικα της cache δεδομένων στην κύρια μνήμη (ή σε ανώτερη cache), και ύστερα να μεταφέρει τον κώδικα από την κύρια μνήμη στην cache εντολών, βάζοντας με προσοχή barriers ώστε να μην αρχίσει η εκτέλεση προτού τελειώσει η μεταφορά. Στην arm το cache flushing ήταν privileged operation οπότε το αναλάμβανε το λειτουργικό σύστημα, προσφέροντας μία κατάλληλη κλήση στο userspace (sys_cacheflush), ενώ στην aarch64 υπάρχουν user-mode instructions για τη διαχείριση της cache, οπότε κάθε εφαρμογή μπορεί να χειριστεί την cache όπως της χρειάζεται, αρκεί φυσικά το κομμάτι της cache που χειρίζεται να της ανήκει (write access).

Οι εντολές της aarch64 χωρίζονται στις εξής μεγάλες κατηγορίες: Αποθήκευση και φόρτωση από τη μνήμη (loads and stores), εντολές ελέγχου εκτέλεσης και συστήματος (branches, exceptions, system), επεξεργασία δεδομένων με απευθείας ορίσματα (data processing - immediate), επεξεργασία δεδομένων με ορίσματα σε καταχωρητές (data processing - register),

και επεξεργασία αριθμών κινητής υποδιαστολής και διανυσμάτων (NEON, SIMD). Ένα σύστημα για να είναι λειτουργικό δεν έχει άμεση ανάγκη της τελευταίας κατηγορίας, όμως μπορεί με τη χρήση της να αυξήσει κατά μεγάλο βαθμό την απόδοσή του.

Μία σημαντική διαφορά ανάμεσα σε arm και aarch64 όσον αφορά τη μορφή των εντολών είναι ότι ενώ στην arm όλες οι εντολές έπαιρναν ένα προαιρετικό όρισμα - συνθήκη εκτέλεσης, που όριζε υπό ποιες συνθήκες (flags) η εντολή θα εκτελεστεί, στην aarch64 αυτή η δυνατότητα έχει μείνει μόνο στις εντολές άλματος, ώστε στις υπόλοιπες εντολές τα bits που χρησιμοποιούνταν να γίνουν διαθέσιμα για σημαντικότερες λειτουργίες. Παρόλα αυτά, κάποιες εντολές που βασίζονται στις συνθήκες έχουν προστεθεί, όπως η conditional swap, η conditional select και η conditional compare, οι οποίες με κατάλληλη χρήση μπορούν να επιταχύνουν μερικές λειτουργίες.

Οι εντολές φόρτωσης από τη μνήμη και αποθήκευσης σε αυτή έχουν ένα σύνολο από αρκετά παρόμοιες εκδόσεις. Καταρχάς όσον αφορά το μέγεθος του δεδομένου που προσπαθούμε να φορτώσουμε ή να αποθηκεύσουμε, αυτό μπορεί να είναι ολόκληρη λέξη (64 bit), λέξη 32-bit, μισή λέξη (16-bit, halfword), ή byte. Οι εκδόσεις 64 και 32 bit αναλαμβάνονται από την ldr και str, καταλαβαίνοντας το μέγεθος από το είδος του καταχωρητή που μεταφέρεται (όλοι οι 64-bit καταχωρητές x0-x31 έχουν 32-bit εκδόσεις w0-w31, το λιγότερο σημαντικό μισό τους). Τα 16-bit δεδομένα από τις ldrh και strh, και τα bytes από τις ldrb και strb. Το ενδιαφέρον είναι σε κώδικα μηχανής ότι αυτές οι τέσσερις εκδόσεις είναι πανομοιότυπες, με μόνο δύο bit να ορίζουν το μέγεθος, κάτι αρκετά βολικό για υλοποιήσεις κωδικοποιητών (encoders).
Όσον αφορά το πρόσημο του δεδομένου, οι 8, 16 και 32 bit εκδόσεις έχουν τις εντολές επέκτασης προσήμου ldrsb, ldrsh και ldrsw αντίστοιχα, οι οποίες κάνουν τις κατάλληλες ενέργειες ώστε κατά τη μεταφορά του αριθμού σε μεγαλύτερο καταχωρητή ο αριθμός και το πρόσημο να μένουν ίδια χωρίς αλλοίωση.
Όσον αφορά τη διεύθυνση προέλευσης και προορισμού, υπάρχουν οι εκδόσεις όπου η διεύθυνση λαμβάνεται σε σχέση με την τιμή του program counter στη θέση της εντολής, για φόρτωση δεδομένων που βρίσκονται κοντά στον κώδικα του προγράμματος, με ακτίνα +/- 1MiB μακριά από την εντολή. Υπάρχουν οι εντολές με unscaled offset, ldur και stur (ldurb, ldursb, sturh κλπ.), όπου η διεύθυνση υπολογίζεται από το άθροισμα της τιμής ενός καταχωρητή - ‘βάσης’ και ενός άμεσου αριθμού (immediate) ο οποίος δηλώνει απόσταση σε αριθμό bytes, με ακτίνα 256 bytes. Και υπάρχουν και οι συνηθισμένες ldr και str, οι οποίες λειτουργούν όπως και αυτές του unscaled offset, όμως ανάλογα με το εάν το μέγεθος είναι byte, halfword, 32-bit ή word, ο αριθμός - απόσταση πολλαπλασιάζεται με x1, x2, x4 και x8 αντίστοιχα. Έτσι, απόσταση 1 σε μία ldr που φορτώνει λέξη 64-bit σημαίνει απόσταση 1 λέξη (8 bytes), ενώ σε μία ldr που φορτώνει halfword σημαίνει απόσταση 1 halfword (2 bytes). Αυτό φαίνεται μόνο στην κωδικοποίηση και όχι σε συμβολική γλώσσα, όπου φαίνεται η καθαρή απόσταση, και είναι σημαντικό για τις υλοποιήσεις διότι για τον υπολογισμό του offset θα πρέπει να πάρουν περιπτώσεις ανάλογα με το μέγεθος του δεδομένου που πρέπει να φορτωθεί.
Τέλος, όσον αφορά τις επιπλέον λειτουργίες, η τελευταία, πιο συνήθης μορφή offset, υποστηρίζει τρεις εκδοχές, οι οποίες επηρεάζουν τον καταχωρητή ο οποίος περιέχει τη διεύθυνση ‘βάσης’. Η εκδοχή pre-index αυξάνει τη διεύθυνση βάσης κατά offset, προτού γίνει η

μεταφορά, και έχει εύρος offset 256 bytes. Η παρόμοια εκδοχή post-index αυξάνει τη διεύθυνση βάσης κατά offset μετά τη μεταφορά. Αυτές οι δύο εκδοχές χρησιμεύουν για την αποδοτική επεξεργασία μεγάλων πινάκων δεδομένων, και μπορούν να χρησιμεύσουν σε βελτιστοποιήσεις σε ένα μεταγλωττιστή, όπου σε άλλη περίπτωση θα έπρεπε η λειτουργία να γίνει σε δύο βήματα, πρώτα φόρτωση / αποθήκευση και μετά αύξηση του δείκτη πίνακα. Τέλος υπάρχει και η απλή εκδοχή unsigned offset, όπου χωρίς να επηρεάσει τη διεύθυνση βάσης πραγματοποιεί κανονική πρόσβαση στη μνήμη με θετικό μόνο εύρος άμεσου offset 4KiB, 8KiB, 16KiB και 32KiB για μεταφορές byte, halfword, 32-bit και word αντίστοιχα. Εάν σε οποιαδήποτε περίπτωση το εύρος δεν επαρκεί χρησιμοποιούμε την έκδοση με register offset.
Στις εντολές φόρτωσης και αποθήκευσης με register offset μας δίνεται μία δυνατότητα βελτιστοποίησης με στόχο τη μείωση εντολών, σε περίπτωση που ο καταχωρητής που συγκρατεί το offset μας περιέχει όχι αριθμό bytes, αλλά αριθμό λέξεων, συχνό φαινόμενο σε πρόσβαση πινάκων. Με την ενεργοποίηση ενός bit, το offset πολλαπλασιάζεται κατά 2, 4 ή 8, ανάλογα αν αυτό που φορτώνουμε είναι halfword, 32-bit ή word αντίστοιχα, γλιτώνοντάς μας μία εντολή ολίσθησης.

Οι εντολές επεξεργασίας δεδομένων χωρίζονται σε υποκατηγορίες όπως μετακίνηση δεδομένων, αριθμητικές και λογικές. Ο τρόπος με τον οποίο η aarch64 έχει οργανώσει τις εντολές είναι ιδιαίτερος ως προς το ότι αρκετές από τις λογικές εντολές, καθώς και οι εντολές ολίσθησης, είναι πλασματικές (aliases) και μεταφράζονται σε μία κατηγορία εντολών στην οποία η αρχιτεκτονική έχει μεγάλη ευχέρεια, την bitfield move και extract. Για λόγους απόδοσης και ελαχιστοποίησης του κώδικα, εντολές επεξεργασίας δεδομένων οι οποίες δέχονται απευθείας όρισμα έχουν ιδιαίτερο τρόπο μετάφρασης του ορίσματος, ανάλογα με τη χρήση του, ώστε να μπορούν να κωδικοποιούνται μεγαλύτεροι αριθμοί σε μικρότερες εντολές, πράγμα χρήσιμο, αφού πλέον τα δεδομένα έχουν μέγεθος 64 bit ενώ οι εντολές έχουν παραμείνει στα 32.
Σε σχέση με απλούστερες αρχιτεκτονικές, θα μελετήσουμε δύο ιδιοτροπίες κωδικοποίησης. Η μία είναι η κωδικοποίηση άμεσων τιμών σε εντολές μετακίνησης και αριθμητικές, και η άλλη είναι η κωδικοποίηση δυαδικών μασκών (bitmask).
Οι εντολές μετακίνησης με άμεσο όρισμα δέχονται μία άμεση τιμή μεγέθους 16 bit, με την δυνατότητα αυτή κατά τη μεταφορά να υποστεί ολίσθηση κατά 0, 16, 32, ή 48 bit προς τα αριστερά. Υλοποιήσεις μπορούν κατά τη μετακίνηση σταθερών αριθμών με τιμές σε περισσότερα από 16 συνεχόμενα bit να επιλέξουν εάν θα τους κωδικοποιήσουν με φόρτωση από τη μνήμη, το οποίο σημαίνει πιθανές καθυστερήσεις λόγω αστοχίας της ενδιάμεσης ταχείας μνήμης (stalls, cache misses), ή με ακολουθία εντολών μετακίνησης movz (μετακίνηση με μηδενισμό) και movk (μετακίνηση με συγκράτηση) ώστε ο μεγάλος σταθερός αριθμός να φορτωθεί σε 2, 3 ή και 4 τμήματα των 16 bit, με κόστος την παραγωγή περισσότερων εντολών.
Παρόμοια δυνατότητα βλέπουμε και στις αριθμητικές εντολές, όπου το άμεσο όρισμα των 12 bit μπορεί να ολισθήσει μία φορά κατά 12 bit. Έτσι, μία πρόσθεση με σταθερά μεγέθους 24 bit μπορεί θεωρητικά να χωριστεί σε δύο προσθέσεις των 12 bit αντί για πρόσθεση με αριθμό φορτωμένο από τη μνήμη. Πρακτικά, αυτές οι βελτιστοποιήσεις ίσως είναι αρκετά ‘ακριβές’, με την έννοια της αύξησης της απόδοσης σε σχέση με την αύξηση του χρόνου μεταγλώττισης, και το αν θα είναι επιτυχημένες εξαρτάται από πολλούς παράγοντες, όχι μόνο ορατούς σε γενικές μετρήσεις (benchmarks) αλλά ίσως και από το μέγεθος της εκάστοτε cache line. Το καλό σε

υλοποιήσεις με ahead-of-time μεταγλωττιστές όπως ο HiPE είναι ότι αφού η μεταγλώττιση και η εκτέλεση γίνονται στο ίδιο μηχάνημα, είναι δυνατόν να είναι γνωστές τέτοιες πληροφορίες συστήματος όπως το μέγεθος της cache την ώρα της μεταγλώττισης, οπότε ο μεταγλωττιστής να προβεί σε βελτιστοποιήσεις κατάλληλες για το κάθε μέγεθος, ανάλογα αν το δεδομένο που θα φορτωθεί θα έχει πιθανότητα να είναι μέσα ή έξω από την cache την ώρα της εκτέλεσης. Βέβαια, τόσο λεπτομερής βελτιστοποίηση θα ήταν κάπως έξω από τα όρια αυτής της εργασίας, οπότε την αφήνουμε εδώ ως θεωρητικό υπόβαθρο.

Παρόλα αυτά, η αρχιτεκτονική μας δίνει μία ακόμα δυνατότητα βελτιστοποίησης, πολύ πιο εύκολη, που μπορούμε να λάβουμε υπόψη. Όλες οι αριθμητικές και λογικές εντολές που δέχονται όρισμα από καταχωρητή έχουν προαιρετική δυνατότητα αριστερής, δεξιάς και δεξιάς με διατήρηση προσήμου ολίσθησης κατά οποιοδήποτε αριθμό bits, κάνοντας δυνατή την σύμπτυξη δύο εντολών, ολίσθησης και αριθμητικής, σε μία, με μία απλή βελτιστοποίηση τύπου peephole.

Για τις λογικές εντολές οι οποίες δέχονται άμεσο όρισμα, η αρχιτεκτονική προσφέρει έναν ιδιαίτερο τρόπο για την κωδικοποίηση των άμεσων τιμών, όπου μία μάσκα-κατασκευαστής των 13 bit χρησιμοποιείται για να δημιουργήσει την πραγματική τιμή του ορίσματος κατά την αποκωδικοποίηση. Ο μηχανισμός δημιουργίας πραγματεύεται την επανάληψη και συνένωση μιας μικρότερης μάσκας - μήτρας, η οποία δημιουργείται γεμίζοντας μέρος της με δυαδικά 1 και μετά περιστρέφοντάς την κατά ένα αριθμό θέσεων. Τα μεγέθη σε αυτή τη διαδικασία είναι το E, το μέγεθος της μάσκας - μήτρας που θα επαναληφθεί (δύναμη του 2, μέχρι 64), το S, ο αριθμός των δυαδικών 1 μέσα σε αυτή (μείον 1), και το R, ο αριθμός των θέσεων που θα περιστραφεί. Ο τρόπος με τον οποίο αυτά τα μεγέθη κωδικοποιούνται στη μάσκα - κατασκευαστή των 13 bit της αρχικής εντολής είναι: N:immr:imms, όπου R = immr, S = imms όπου τα μη χρησιμοποιημένα πιο σημαντικά ψηφία (εξαιτίας του μεγέθους της μήτρας) έχουν την τιμή 1, και το N σημαίνει μήτρα μεγέθους 64 bit (καμία επανάληψη).

Απαραίτητα παραδείγματα:

Bitmask σε λογική εντολή: ‘1000110001010’ - N=1, immr=000110, imms=001010. Παραγόμενη άμεση τιμή (64 bit): 000…0000000000000000111111111110000000.

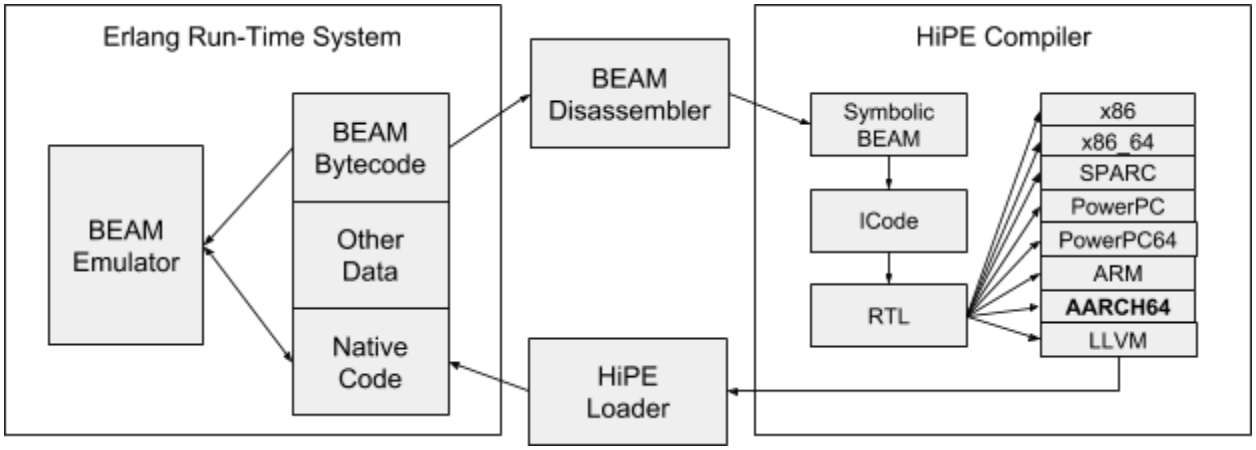
Bitmask σε λογική εντολή: ‘0000101110100’ - N=0, immr=0000101, imms=110100 ~ $S=100_2=4_{10}$, E=8. Παραγόμενη άμεση τιμή: 1100001111000011110000111100001 1… 11000011.

Γνωρίζοντας αυτά, ο αλγόριθμος κωδικοποίησης που θα πρέπει να χρησιμοποιήσουν οι υλοποιήσεις μπορεί να σχεδιαστεί με απλό και ευθύ τρόπο. Έστω σταθερά A η οποία πρέπει να διαπιστωθεί αν και πώς μπορεί να κωδικοποιηθεί σε άμεσο όρισμα. Καταρχάς βρίσκεται η μικρότερη επαναλαμβανόμενη υπακολουθία. Αρχίζοντας από μέγεθος E=2 και διπλασιάζοντας σε κάθε σάρωση, η σταθερά τεμαχίζεται σε τμήματα μεγέθους E τα οποία συγκρίνονται μεταξύ τους, ή επαναλαμβάνονται προσπαθώντας να επανακατασκευάσουν τη σταθερά. Αν η σταθερά δεν περιέχει τέτοια επανάληψη, θέτουμε N=1. Ύστερα μετράμε τον αριθμό των δυαδικών 1 από την αρχή της ακολουθίας, και ύστερα τον αριθμό των δυαδικών 0 από το σημείο που τα 1 τελειώνουν, μέχρι να ξαναβρεθεί 1 ή να τελειώσει η υπακολουθία. Με αυτές τις μετρήσεις, εκτιμούμε τα S και R (αριθμός άσσων και αριθμός περιστροφών - 1). Με τα imms και immr που προκύπτουν, επιχειρούμε να ανακατασκευάσουμε τη σταθερά ή την υπακολουθία με τον αλγόριθμο αποκωδικοποίησης της αρχιτεκτονικής. Εάν η σταθερά που προκύπτει είναι ίση με

την αρχική, έχουμε πετύχει, και κωδικοποιούμε τη σταθερά με τα N, imms και immr που βρήκαμε. Αλλιώς, χρησιμοποιούμε φόρτωση της σταθεράς από τη μνήμη και την έκδοση της λογικής εντολής που παίρνει όρισμα από καταχωρητή. Όποιος ενδιαφέρεται, μπορεί να μελετήσει τη δική μας υλοποίηση από τον πηγαίο κώδικα της εργασίας (σε Erlang).

3. Η γλώσσα προγραμματισμού Erlang

Ως γλώσσα προγραμματισμού, η Erlang προσφέρει μερικές αξιοζήλευτες δυνατότητες. Οργάνωση του κώδικα σε λειτουργικά τμήματα (modules) εξασφαλίζει την απομόνωση των λειτουργιών του κάθε τμήματος, προσφέροντας απλότητα στη μετάφραση και χρήση συμβόλων. Αποστολή μηνυμάτων μεταξύ διεργασιών για υποβοήθηση της παράλληλης εκτέλεσης, και αυτόματη διαχείριση διεργασιών σε απομακρυσμένα συστήματα. Ευρετήριο μεταβλητών (process dictionary) για κάθε διεργασία, δίνοντας ευκολίες που κανονικά δεν υπάρχουν σε αγνές συναρτησιακές γλώσσες, και φυσικά μία αρκετά μεγάλη και βολική βιβλιοθήκη με χρήσιμες κλήσεις.
Όμως το σημείο στο οποίο θα εστιάσουμε στη γλώσσα, είναι η ροή της εκτέλεσης και της μεταγλώττισης, καθώς και η δομή του κώδικα της υλοποίησης OTP. Διότι γνωρίζοντας αυτά, θα βοηθηθούμε στο να σχεδιάσουμε τον τρόπο με τον οποίο θα εργαστούμε.

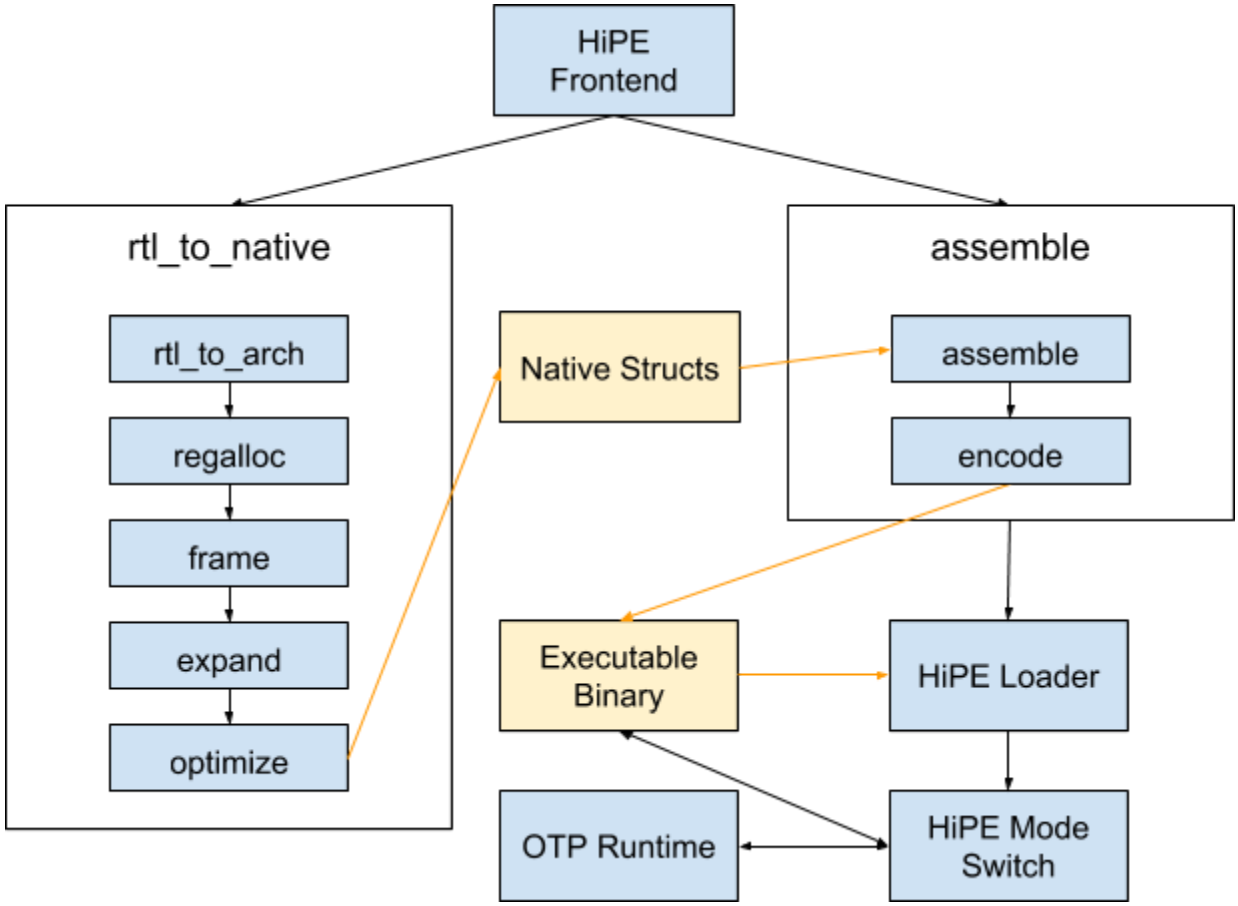


Σχήμα 3.1: Δομή και τρόπος λειτουργίας της υλοποίησης Erlang/OTP.

Ο τυπικός τρόπος λειτουργίας για την εκτέλεση ενός προγράμματος Erlang στο σύστημα Erlang/OTP, είναι η μεταγλώττισή του σε κώδικα για το υποσύστημα BEAM, μία εικονική μηχανή η οποία μπορεί να εκτελέσει (να προσομοιώσει) με αρκετά καλή απόδοση προγράμματα μεταφρασμένα στον bytecode της πάνω σε αρκετές διαφορετικές πλατφόρμες. Ο προσομοιωτής αναλαμβάνει την εξυπηρέτηση κλήσεων προς το σύστημα εκτέλεσης της Erlang όπου αυτές υπάρχουν.
Στην περίπτωση χρήσης του υποσυστήματος High Performance Erlang - HiPE, ο κώδικας BEAM αναλύεται περνώντας μέσα από διάφορα ενδιάμεσα στάδια, με σκοπό να μετατραπεί σε κώδικα μηχανής για το μηχάνημα - στόχο. Τα δύο πρώτα ενδιάμεσα στάδια, η συμβολική BEAM γλώσσα και ο Icode, είναι σχετικά ανεξάρτητα από την αρχιτεκτονική, έχοντας μεταβλητές και άπειρους νοητούς καταχωρητές. Κατά τη μετατροπή του Icode σε RTL, Register Transfer Language, ο HiPE χρησιμοποιεί κάποιες βασικές πληροφορίες για την αρχιτεκτονική, όπως ο αριθμός των registers, το μήκος λέξης και το endianness, και τους δείκτες σωρού και στοίβας, μιας και εκεί παράγονται τμήματα κώδικα όπως το σπρώξιμο παραμέτρων και μεταβλητών στη

στοίβα, και η πρόσβαση και επεξεργασία δεδομένων στο σωρό. Τέτοιες πληροφορίες λαμβάνονται από το αρχείο κώδικα lib/hipe/rtl/hipe_rtl_arch.erl, το οποίο δίνει στον μεταφραστή icode-to-rtl τις απαραίτητες πληροφορίες για την κάθε αρχιτεκτονική. Μερικές πληροφορίες δεν είναι δυνατόν να είναι γνωστές την ώρα της συγγραφής του hipe_rtl_arch, όπως π.χ. το endianness του μηχανήματος - στόχου, οπότε συμπεραίνονται την ώρα της μεταγλώττισης, παραγωγής και εγκατάστασης (build and install) του συστήματος Erlang/OTP στο μηχάνημα - στόχο, όπου ο μεταγλωττιστής που παράγει το εκτελέσιμο της υλοποίησης λ.χ. ο gcc παρέχει τέτοιες πληροφορίες αρχιτεκτονικής με κατάλληλες μακροεντολές, οι οποίες χρησιμοποιούνται και αποθηκεύονται σε σταθερές συστήματος (built-in literals) της Erlang στο αρχείο erts/emulator/hipe/hipe_mkliterals.c. Αργότερα, κατά την εκτέλεση του HiPE, το hipe_rtl_arch έχει όλες αυτές τις πληροφορίες που χρειάζεται ως built-in σταθερές.

Όταν πλέον ο κώδικας είναι σε μορφή RTL, δίνεται στο στο backend υποσύστημα που αντιστοιχεί στην αρχιτεκτονική - στόχο, το οποίο θα κάνει την τελική μεταγλώττιση σε κώδικα μηχανής. Υπάρχουν backends για αρκετές αρχιτεκτονικές, και όλα έχουν μία κοινή δομή και διαδικασία που ακολουθούν για την παραγωγή του τελικού κώδικα. Εμείς σε περίπτωση που θέλουμε να εξειδικεύσουμε σε κάποιο στοιχείο της δομής, θα βασιστούμε κυρίως στο backend της αρχιτεκτονικής ARM, μιας και είναι η πιο παραπλήσια σε αυτή που στοχεύουμε να υλοποιήσουμε.
Η δομή ενός υποσυστήματος backend του HiPE μπορεί να χωριστεί σε δύο μεγάλα βασικά τμήματα. Το τμήμα της μετάφρασης της RTL σε δομές της αρχιτεκτονικής, και το τμήμα της κωδικοποίησης των δομών της αρχιτεκτονικής σε δυαδικό κώδικα. Οι δύο διαδικασίες εκκινούνται με τη σειρά από το lib/hipe/main/hipe_main.erl, στις κλήσεις rtl_to_native και assemble αντίστοιχα.
Η διαδικασία του τμήματος της μετάφρασης της RTL σε δομές αρχιτεκτονικής (στο εξής ‘εγγενείς δομές’, ‘native structs’), η οποία βρίσκεται γενικά στο lib/hipe/<arch>/hipe_<arch>_main.erl, περιλαμβάνει μερικές συγκεκριμένες φάσεις, οι οποίες εκτελούνται σειριακά. Αυτές είναι με τη σειρά: η παραγωγή των εγγενών δομών (translate), η ανάθεση καταχωρητών σε δομές (register allocation), η κατασκευή κώδικα διαχείρισης στοίβας (frame allocation), και η αποσυμπίεση των δομών και οι βελτιστοποιήσεις (expansion, optimizations).
Οι εγγενείς δομές της αρχιτεκτονικής, οι οποίες περιγράφονται στο lib/hipe/<arch>/hipe_<arch>.hrl, αντιπροσωπεύουν πραγματικές εντολές της αρχιτεκτονικής, και είναι όσο το δυνατόν κοντά στην συμβολική της γλώσσα, όμως σε ένα αφαιρετικό επίπεδο το οποίο περιγράφει γενικά τη λειτουργία και τις δυνατότητες της κάθε εντολής, χωρίς να μπαίνει σε λεπτομέρειες αρίθμησης, κωδικοποίησης και διευθύνσεων. Κατά την αρχική παραγωγή τους έχουν μέσα κυρίως αναφορές και όχι τελικό κώδικα. Για παράδειγμα, μία εντολή add θα περιγραφόταν ως μία δομή “αριθμητικής πράξης χωρίς παρενέργειες” (υποθέτοντας ότι όλες οι αριθμητικές πράξεις στην εν λόγω αρχιτεκτονική έχουν όμοια κωδικοποίηση), η οποία έχει ως πεδία τον τύπο της αριθμητικής πράξης, αναφορές προς τις (μη αναθετημένες σε καταχωρητές) μεταβλητές, σταθερές ή atoms της rtl, άμεσες σταθερές, αναφορά στη μεταβλητή που θα αποθηκευτεί το αποτέλεσμα, και ίσως παραμέτρους ανάλογα την εντολή, όπως ‘ολίσθηση 3 θέσεις δεξιά πριν την πράξη’.



Σχήμα 3.2: Ροή λειτουργίας ενός HiPE backend.

Το στάδιο της μετάφρασης σε εγγενείς δομές βρίσκεται στο lib/hipe/<arch>/hipe_rtl_to_<arch>.erl, και γενικά πραγματοποιεί εν παραλλήλω τη μετάφραση των συναρτήσεων που υπάρχουν σε ένα module, οπότε η hipe_rtl_to_<arch>:translate γενικά αφορά μία μόνο συνάρτηση τη φορά. Σε αυτό το στάδιο εντολές rtl αντιστοιχούνται σε μία ή περισσότερες εγγενείς δομές, και οι άμεσες σταθερές εξετάζονται και μετατρέπονται σε μορφές φιλικές για τις εντολές της αρχιτεκτονικής, συνήθως με κλήσεις από το hipe_<arch>.erl. Δηλαδή, εκεί θα εξεταστεί αν η άμεση τιμή χωράει αυτούσια στην εντολή που χρησιμοποιείται, αν δεν χωράει, όμως μπορεί να χωρέσει με κάποια μετατροπή την οποία υποστηρίζει η εντολή λ.χ. ολίσθηση, η οποία μετατροπή θα συνοδέψει τη σταθερά μέσα στην εγγενή δομή, ή αν δε χωράει με κανένα τρόπο, οπότε η μετάφραση αυτού του σταδίου θα πρέπει να παράξει μία προηγούμενη δομή-εντολή φόρτωσης από τη μνήμη σε ενδιάμεσο καταχωρητή. Σε αυτό το στάδιο πρέπει να εισαχθούν όσο είναι αυτό δυνατόν όλοι οι ενδιάμεσοι καταχωρητές - μεταβλητές rtl που χρειάζονται, διότι ακολουθεί το στάδιο αντιστοίχισης καταχωρητών rtl σε πραγματικούς, και ύστερα χρήση νέων καταχωρητών είναι πλέον δύσκολη.

Η ανάθεση πραγματικών καταχωρητών σε μεταβλητές rtl είναι διαδικασία που εξαρτάται από τον αλγόριθμο με τον οποίο θα γίνει η ανάθεση. Ο αλγόριθμος μπορεί να είναι γραμμικός, αφελής, με χρωματισμό γράφων κ.α. Φυσικά η υλοποίηση αυτού του αλγορίθμου δε μπορεί να

είναι ευθύνη του backend της εκάστοτε αρχιτεκτονικής. Οι αλγόριθμοι υπάρχουν στο lib/hipe/regalloc/, και εκκινούνται μετά το βήμα της μετάφρασης, ανάλογα με ποιον αλγόριθμο έχει επιλέξει ο χρήστης - ή το επίπεδο της βελτιστοποίησης. Εξυπακούεται ότι η ανάθεση των πραγματικών καταχωρητών είναι σχετική με την κάθε αρχιτεκτονική, και έτσι οι αλγόριθμοι αυτοί παίρνουν πληροφορίες σχετικές με την αρχιτεκτονική ή και εκτελούν τμηματικά κομμάτια του κάθε backend ανά περίσταση, με κλήσεις που υπάρχουν στο regalloc/hipe_<arch>_specific.erl. Το αρχείο αυτό ανακατευθύνει τις κλήσεις στα κατάλληλα modules του backend, hipe_<arch>_cfg για δημιουργία και διαχείριση δομών απαραίτητων για τη διαδικασία, hipe_<arch>_ra_postconditions για παραγωγή επιπλέον εντολών π.χ. μετακίνησης δεδομένων μεταξύ καταχωρητών σε περιπτώσεις που είναι απαραίτητο (temp spilling), hipe_<arch>_liveness_gpr για διαχείριση δομών σχετικές με το διάστημα ζωής (liveness) των καταχωρητών γενικής χρήσης, hipe_<arch>_defuse για πληροφορίες σχετικά με το πως κάθε εγγενής δομή χρησιμοποιεί τους καταχωρητές που τις δίνονται, και hipe_<arch>_subst για έλεγχο τύπων και αντικατάσταση προσωρινών τιμών. Μαζί με αυτά, τα hipe_<arch>_ra_finalize για τελική τοποθέτηση των νέων καταχωρητών πίσω στις εγγενείς δομές, και τα hipe_<arch>_ra_<algorithm> για κλήσεις απαραίτητες για τον εκάστοτε αλγόριθμο.
Όλα αυτά τα αρχεία μοιάζουν στη μορφή μεταξύ αρχιτεκτονικών, καθώς διατηρούν την ίδια περίπου λογική. Η διαφορές τους αφορούν περισσότερο το ότι η κάθε αρχιτεκτονική έχει διαφορετικές εντολές και άρα διαφορετικές εγγενείς δομές που αυτά τα αρχεία επεξεργάζονται, και στο ότι σε μερικές εγγενείς δομές ή μορφές προσωρινών τιμών έχουν σε κάποιες αρχιτεκτονικές ιδιαίτερα χαρακτηριστικά π.χ. παρενέργειες.

Στο επόμενο στάδιο, έχοντας πλέον ξεκαθαρίσει κατά το στάδιο αντιστοίχισης καταχωρητών πόσες επιπλέον θέσεις στη στοίβα θα χρειαστούν για αποθήκευση προσωρινών τιμών την ώρα των πράξεων, είναι στιγμή να παραχθεί ο κώδικας προετοιμασίας της στοίβας, ο οποίος βρίσκεται στην αρχή της εκτέλεσης της παραγόμενης συνάρτησης, καθώς και σε απαραίτητα σημεία όπως η κλήσεις και επιστροφές άλλων συναρτήσεων μέσα στο σώμα της. Η παραγωγή του κώδικα γίνεται στο hipe_<arch>_frame.erl, και έχει δύο τμήματα, την παραγωγή του αρχικού κώδικα προετοιμασίας (πρόλογος συνάρτησης, prologue), και την επεξεργασία των εγγενών δομών που έχουν πρόσβαση στη στοίβα.
Η παραγωγή του προλόγου έχει ως ευθύνη της να αυξήσει τον δείκτη στοίβας κατά όσο χώρο θα χρειαστεί η συνάρτηση. Αυτός υπολογίζεται από το μέγιστο χώρο που χρειάζονται οι προσωρινές τιμές που αποθηκεύονται στη στοίβα, καθώς και οι παράμετροι για τυχόν κλήσεις που υπάρχουν. Εκεί παράγεται και κώδικας όπου εάν διαπιστωθεί ότι ο χώρος που παρέχει το σύστημα για τη στοίβα, ο οποίος διαπιστώνεται μέσω του process pointer, δεν επαρκεί για την εκτέλεση της συνάρτησης, καλεί το built-in function (BIF) inc_stack, το οποίο αναλαμβάνει να ζητήσει από το σύστημα διπλασιασμό του χώρου που παρέχεται για τη στοίβα. Ο process pointer είναι σταθερά τοποθετημένος σε γνωστό καταχωρητή από το περιβάλλον εκτέλεσης, και θα τον δούμε παρακάτω μαζί με τον τρόπο εκτέλεσης των BIFs.
Αφού δεσμευτεί χώρος στη στοίβα για τις προσωρινές τιμές, πρέπει σε κάθε σημείο που χρησιμοποιείται προσωρινή τιμή που δεν αποθηκεύεται σε καταχωρητή (pseudo temp) να τοποθετηθεί κώδικας πρόσβασης στη θέση της στοίβας που αντιστοιχεί στην κάθε προσωρινή τιμή. Η αντιστοίχιση προσωρινής τιμής με απόσταση (offset) από την κορυφή της στοίβας στην

οποία αποθηκεύεται γίνεται την ώρα της μεταγλώττισης με ένα απλό λεξικό (map / tree). Αναλύονται με τη σειρά όλες οι εγγενείς δομές της συνάρτησης, και γίνεται αντικατάσταση στα σημεία που χρειάζεται πρόσβαση στη στοίβα με τη σωστή ακολουθία εντολών.

Στο τελικό στάδιο, το οποίο βρίσκεται στο hipe_<arch>_finalize, υπάρχουν δύο εργασίες που γίνονται. Πρώτα γίνεται αποσυμπίεση σύνθετων εγγενών δομών σε απλούστερες, και ύστερα βελτιστοποίηση μέσω σύμπτυξης ακολουθιών εντολών σε εγγενείς δομές μεγαλύτερης απόδοσης, οι οποίες είναι ικανές να εκτελέσουν την ίδια εργασία που έκανε η ακολουθία από την οποία προήλθαν με λιγότερες όμως εντολές.
Η αποσυμπίεση είναι απλή διαδικασία, και σε αυτή σύνθετες εγγενείς δομές για τις οποίες η αρχιτεκτονική δεν έχει αντίστοιχες εντολές αναλύονται σε ακολουθίες από μικρές, απλούστερες. Για παράδειγμα, μία εντολή άλματος υπό συνθήκη με δύο περιπτώσεις, για αληθή και ψευδή συνθήκη, πιθανότατα θα μεταφραστεί σε δύο εντολές άλματος, μία υπό συνθήκη και μία χωρίς, η οποία δεύτερη θα εκτελεστεί υποχρεωτικά αν η πρώτη αποτύχει. Από την άλλη, σε περίπτωση μη βελτιστοποιήσεων, κάποιες εγγενείς δομές ίσως να μην αντιστοιχούν σε κάποια χρήσιμη εντολή της αρχιτεκτονικής, οπότε σε αυτό το στάδιο να πρέπει αναγκαστικά να απαλειφθούν ή να αντικατασταθούν από no-ops.
Η βελτιστοποίηση μέσω σύμπτυξης εντολών γίνεται μέσω της μεθόδου ‘κλειδαρότρυπας’ - peephole. Σε αυτή τη μέθοδο αναλύονται όλες οι εντολές - εγγενείς δομές με τη σειρά, σε κάθε εντολή που αναλύεται διαβάζονται και μερικές από τις κοντινές επόμενές της. Αν διαπιστωθεί ότι οι εντολές που ακολουθούν γίνεται να αντικατασταθούν από μία μικρότερη και απλούστερη εντολή, γίνεται η αντικατάσταση και η διαδικασία επαναλαμβάνεται, αλλιώς προχωρά στην ανάλυση της επόμενης εντολής. Σε αυτό το στάδιο ο χρήστης έχει την επιλογή να απενεργοποιήσει τις βελτιστοποιήσεις δίνοντας την κατάλληλη παράμετρο - διακόπτη για τη μεταγλώττιση, οπότε το στάδιο δεν θα εκτελεστεί.

Όταν πλέον οι εγγενείς δομές έχουν παραχθεί, και η βελτιστοποίηση και μορφή τους είναι τέτοια ώστε μία εγγενής δομή της αρχιτεκτονικής να αντιστοιχεί σε μία ακριβώς εντολή της συβολικής γλώσσας της, οι εγγενείς δομές από όλες τις συναρτήσεις μαζεύονται και σειριοποιούνται σε μία μεγάλη γραμμική λίστα εγγενών δομών προς μετάφραση, και δίνονται στη δεύτερη φάση του HiPE backend, της μετατροπής των εγγενών δομών σε δυαδικό κώδικα.
Αυτή η φάση έχει δύο στάδια. Στο πρώτο γίνεται η μετάφραση των εγγενών δομών σε συμβολική γλώσσα (assemble), και στο δεύτερο γίνεται η μετατροπή της συμβολικής γλώσσας σε δυαδικό κώδικα (encode). Η διαδικασία σε αυτό το σημείο είναι ιδιαίτερα σχετισμένη με την εκάστοτε αρχιτεκτονική, και έτσι το στάδιο assemble, και ειδικά το στάδιο encode, είναι τόσο διαφορετικό από backend σε backend, ώστε να είναι απαραίτητο ο κωδικοποιητής να πρέπει να ξαναγραφτεί από την αρχή, χωρίς μεγάλη δυνατότητα να μπορούν να αντιγραφούν χρήσιμα κομμάτια από άλλα backends για ευκολότερη ανάπτυξη. Η ανάπτυξη εδώ γίνεται πάντοτε με ταυτόχρονη προσεκτική ανάγνωση των πλήρων architecture specifications της κάθε αρχιτεκτονικής.
Το στάδιο μετατροπής εγγενών δομών σε συμβολική γλώσσα (assembly) είναι ίσως αυτό που ακολουθεί κάποιες γενικές συμβάσεις στη μορφή του, καθώς είναι το τμήμα που καλείται απευθείας από το front-end του HiPE και έτσι περιμένει είσοδο σε δεδομένη μορφή. Οι εγγενείς

δομές που δέχεται ως είσοδο αρχικά μετατρέπονται γραμμικά σε δικές του δομές οι οποίες θα δοθούν στον κωδικοποιητή, συνήθως πλειάδες αντικειμένων με τη σειρά που η αντίστοιχη εντολή της αρχιτεκτονικής σε bits κωδικοποιεί αυτά τα αντικείμενα.
Κατά τη γραμμική μετατροπή, είναι συχνά απαραίτητο να εξεταστεί αν η εντολή που τοποθετείται χρειάζεται για την εκτέλεσή της κάποια προετοιμασία κατά την ώρα της φόρτωσης. Συγκεκριμένα, οι κλήσεις συναρτήσεων είναι άλματα σε θέσεις σχετικές με τη θέση της εντολής, όμως μόνο ο φορτωτής γνωρίζει τη διεύθυνση της συνάρτησης - στόχου που καλείται, διότι αυτός την τοποθετεί στη μνήμη. Έτσι ο συμβολομεταφραστής θα πρέπει να τοποθετήσει πριν την κλήση ένα σχόλιο προς τον φορτωτή, λεγόμενο relocation entry, το οποίο δίνει την εντολή να φορτωθεί σε εκείνη τη θέση κατάλληλο άλμα με τη διεύθυνση της συνάρτησης - στόχου. Ομοίως πρέπει να γίνει και με τους stack descriptors μετά από κάθε κλήση συνάρτησης η οποία επιστρέφει, έτσι ώστε να μπορεί να γίνει σωστή προώθηση εξαιρέσεων (exception handling) οι οποίες θα συμβούν στην καλούμενη συνάρτηση.
Αφού οι εντολές μπουν στη σειρά, όλα τα σχετικά άλματα εξετάζονται και τους δίνεται διεύθυνση ανάλογα με την απόσταση από την ετικέτα (label) στην οποία αναφέρονται. Η ανάλυση διευθύνσεων των αλμάτων γίνεται τόσο αργά, διότι προηγουμένως είναι αδύνατο να γνωρίζουμε αποστάσεις από ετικέτες, καθώς δεν ξέρουμε τον όγκο των εντολών που διαμεσολαβούν, ούτε τα τμήματα μόνο δεδομένων και σταθερών που χρειάζεται κάθε τόσο να βάλουμε ώστε οι σταθερές να είναι αρκετά κοντά στο σημείο που χρησιμοποιούνται. Για τον ίδιο λόγο εξετάζονται και αντικαθίστανται και οι φορτώσεις δεδομένων από σχετική θέση, όπου η διεύθυνση αναφοράς είναι ετικέτα.
Όσον αφορά τις σταθερές που δεν είναι γνωστές κατά την ώρα της μεταγλώττισης, π.χ. atoms ή διευθύνσεις από δομές δεδομένων, ο συμβολομεταφραστής προσθέτει ένα σχόλιο (relocation entry) προς τον φορτωτή, και μία άδεια λέξη για χώρο αμέσως μετά όπου ο φορτωτής θα τοποθετήσει πριν την εκτέλεση τη σταθερά όπως του δίνεται από το περιβάλλον εκτέλεσης.

Στη συνέχεια, αφού πλέον η συμβολική γλώσσα έχει παραχθεί, δίνεται στον κωδικοποιητή για να τη μετατρέψει σε δυαδικό κώδικα. Εκεί, πλέον οι εντολές μεταφράζονται μία προς μία σε αυστηρά κατασκευασμένες δυαδικές αριθμητικές τιμές, σύμφωνα με τα specifications της αρχιτεκτονικής, και σειριοποιούνται σε ένα δυαδικό αρχείο στη μνήμη. Συνήθως οι αρχιτεκτονικές, και ειδικά οι RISC, προσφέρουν μία ιεραρχική οργάνωση των εντολών τους, όπου οι εντολές με παρόμοια λειτουργία έχουν μία βασική κοινή ‘μορφή’ (form), η οποία αλλάζοντας κάποια πεδία γίνεται να εξειδικευτεί. Αυτό είναι βολικό για τις υλοποιήσεις, καθώς για κάθε ξεχωριστή συμβολική εντολή μπορεί να κατασκευαστεί στον κωδικοποιητή μία συνάρτηση που παράγει τις παραμέτρους που της αντιστοιχούν, οι οποίες δίνονται στη συνάρτηση που συνθέτει την πιο κοινή μορφή ώστε να την παράξει. Ακόμα βοηθητικό για την υλοποίηση είναι να υπάρχουν συναρτήσεις ή μακροεντολές μετατροπής αριθμητικών παραμέτρων σε δυαδικό κώδικα, οι οποίες να ελέγχουν για σφάλματα μεγέθους (περίπτωση ο αριθμός να μη χωράει στο πεδίο της εντολής) και για μετατροπές ολίσθησης στη σωστή θέση και επεξεργασίας σταθερών με πρόσημο (τεμαχισμός αρνητικού αριθμού, όπου το μπροστινό τμήμα του έχει 1 αντί για 0 και έτσι δεν μπορεί να επεξεργαστεί με την απλή λογική των απρόσημων αριθμών). Χρησιμοποιώντας τέτοιες συναρτήσεις στην υλοποίησή μας για κάθε

δυαδικό αριθμό που τοποθετούσαμε στο πεδίο δυαδικών τιμών των εντολών εντοπίσαμε αρκετά σφάλματα κωδικοποίησης, τα οποία προήλθαν από προηγούμενη φάση του backend.

Ο φορτωτής λαμβάνει το δυαδικό αρχείο που παράχθηκε από το backend του HiPE και αναλαμβάνει να συμπληρώσει ό,τι δεν ήταν γνωστό την ώρα της μεταγλώττισης και να το τοποθετήσει στη μνήμη μαζί με συμπληρωματικά κομμάτια κώδικα απαραίτητα για την εκτέλεση. Η κυρίως υλοποίηση του φορτωτή είναι ανεξάρτητη από την αρχιτεκτονική, όμως για βασικές λειτουργίες χρειάζεται κλήσεις οι οποίες βρίσκονται στο erts/emulator/hipe/hipe_<arch>.c. Τέτοιες λειτουργίες είναι η δέσμευση και απελευθέρωση χώρου για τοποθέτηση κώδικα προς εκτέλεση, η συμπλήρωση διευθύνσεων (patching) γνωστών κατά την φόρτωση ανάλογα το αν η εντολή πρόκειται για κλήση, σταθερά ή άλλο, το οποίο συμπεραίνεται από το relocation entry, και η δημιουργία stubs, δηλαδή μικρών κομματιών κώδικα που λειτουργούν ως ενδιάμεσοι σε κλήσεις ανάμεσα στο δυαδικό εκτελέσιμο και το περιβάλλον εκτέλεσης. Το πιο προφανές stub είναι αυτό που καλείται όταν το δυαδικό εκτελέσιμο προσπαθεί να καλέσει ενσωματωμένη συνάρτηση (built-in function) της erlang υλοποιημένης στο περιβάλλον εκτέλεσης, όπου περνά τη διεύθυνση της συνάρτησης που θέλει να καλέσει ως παράμετρο στην nbif_callemu, η οποία αναλαμβάνει να ενεργοποιήσει το περιβάλλον εκτέλεσης μέσω μιας διαδικασίας που ονομάζεται mode switch.

Την ώρα της εκτέλεσης, οποιαδήποτε κλήση και επιστροφή από και προς το περιβάλλον εκτέλεσης γίνεται μέσω του mode switch, μία διαδικασία κάπως χρονοβόρα σε σχέση με την απλή δυαδική εκτέλεση όμως απαραίτητη, όπου το εκτελούμενο πρόγραμμα σταματά, το αίτημά του δρομολογείται από το περιβάλλον εκτέλεσης ως μήνυμα σε διαφορετική διεργασία, και πάλι με ένα ανάστροφο mode switch συνεχίζεται η εκτέλεση με την επιστροφή της απάντησης. Εκτός από τα BIFs της erlang, κάποιες κλήσεις που χρειάζονται mode switch είναι η αίτηση αύξησης μεγέθους σωρού (gc), η αίτηση αύξησης μεγέθους στοίβας (inc_stack), και η αίτηση προς το χρονοδρομολογητή, για να δοθεί χρόνος εκτέλεσης και σε άλλες διεργασίες (suspend). Έτσι ας είναι γνωστό κατά την υλοποίηση ότι παρόλο που ένα πρόγραμμα μπορεί να μην έχει κλήσεις BIFs, μπορεί να υπάρχουν σε λίγο πιο μεγάλα προγράμματα τέτοιες κλήσεις που προκαλούν mode switch και έτσι απαιτούν τους μηχανισμούς του να λειτουργούν άρτια ώστε να μην υπάρξει παραφθορά δεδομένων. Τέτοιοι μηχανισμοί που χρησιμοποιούνται κατά το mode switch όπως η αποθήκευση μη προσωρινών καταχωρητών, η τοποθέτηση σημαντικών τιμών σε καταχωρητές και η τοποθέτηση παραμέτρων σε καταχωρητές ή στοίβα υπάρχουν στα αρχεία κώδικα hipe_<arch>_asm.m4, hipe_<arch>_bifs.m4 και hipe_<arch>_glue.S.
Ο μηχανισμός του mode switch αναλαμβάνει πριν την εκτέλεση δυαδικού κώδικα να τοποθετήσει σε γνωστούς καταχωρητές τρεις τιμές οι οποίες είναι απαραίτητες για την λειτουργία ενός προγράμματος. Η πρώτη είναι η εγγενής κορυφή της στοίβας, native stack pointer - nsp. Αυτή η τιμή μπορεί να βρίσκεται και στον ίδιο τον καταχωρητή στοίβας της αρχιτεκτονικής, όμως πρέπει να ληφθούν υπόψη οι αρχιτεκτονικοί περιορισμοί. Η δεύτερη είναι ο δείκτης σωρού, heap pointer. Μέσα στο σωρό το σύστημα εκτέλεσης ή το εκτελέσιμο μπορεί να τοποθετήσει άτομα ή και μεγάλα αντικείμενα και δομές, και με αυτό τον τρόπο μπορούν και ανταλλάσσουν δεδομένα. Και η τελευταία είναι ο δείκτης διεργασίας, μία δομή που περιγράφεται στο erl_process και hipe_process.h, και περιέχει πολύ χρήσιμα στοιχεία για τη διεργασία που

έχει δημιουργήσει το σύστημα εκτέλεσης για το εκτελέσιμο, όπως το μέγεθος του σωρού και της στοίβας, το εάν χρειάζεται χρονοδρομολόγηση και άλλους σημαντικούς δείκτες. Οι καταχωρητές που επιλέγονται για να διατηρούν αυτές τις τιμές εξηγούνται καταρχάς στο erts/emulator/hipe/hipe_<arch>_asm.m4, και το lib/hipe/<arch>/<arch>_registers.erl.
Όταν πλέον το εκτελέσιμο ολοκληρώσει τη λειτουργία του επιστρέφει πάλι μέσω ενός mode switch στη διεργασία του hipe που το μεταγλώττισε, φόρτωσε και εκτέλεσε, ώστε να τελειώσει και αυτή τον κύκλο ζωής της.

4. Οργάνωση της επέκτασης

Γνωρίζοντας πλέον τη δομή της υλοποίησης που σκοπεύουμε να αλλάξουμε, καθώς και τα χαρακτηριστικά της αρχιτεκτονικής για την οποία θα την προσαρμόσουμε, μπορούμε να καταστρώσουμε ένα πλάνο για το πώς θα κινηθούμε στην εργασία. Στον κύκλο ζωής ενός λογισμικού, το πρώτο βήμα είναι η επιλογή της μεθοδολογίας ανάπτυξης. Μιας και η εργασία που πραγματοποιούμε έχει ήδη λειτουργήσει για άλλες αρχιτεκτονικές, η λογική πορεία είναι ο στόχος μας να είναι να επιτύχουμε παρόμοια αποτελέσματα με τα backends των άλλων αρχιτεκτονικών. Αυτό μας δίνει ένα σύστημα με ανάδραση: όσο τα αποτελέσματα της υλοποίησής μας δεν συμφωνούν με τις άλλες αρχιτεκτονικές, ας βελτιωθεί κάτι ώστε να συμφωνεί. Θα ήταν βολικό να είχαμε ένα εύκολο τρόπο να ξέρουμε (ο 'αισθητήρας' στον κλάδο ανάδρασής μας) αν κάποιο αποτέλεσμα της υλοποίησής μας δεν είναι σύμφωνο με τα ζητούμενα. Ευτυχώς για την περίπτωσή μας, η υλοποίηση OTP έχει μία εκτενή σουίτα δοκιμών, όπου προγράμματα erlang μεταγλωτίζονται, εκτελούνται και τα αποτελέσματά τους ελέγχονται εάν είναι τα αναμενόμενα. Μπορούμε λοιπόν να χρησιμοποιήσουμε τα αποτελέσματα αυτών των δοκιμών ως ανάδραση, και όσο κάποια δοκιμή δεν πετυχαίνει, θα εργαζόμαστε ώστε να πετύχει. Αυτή η μεθοδολογία ανάπτυξης ονομάζεται ανάπτυξη οδηγούμενη από δοκιμές (test-driven development), και θα είναι η επιλογή μας για αυτή την υλοποίηση, καθώς ταιριάζει στην περίπτωση και είναι αρκετά ταχεία.

Αρχίζοντας την υλοποίηση, θα γίνει προφανές ότι το να πετύχει ακόμα και μία από τις 93 δοκιμές του HiPE, είναι αρκετά υψηλός στόχος, διότι λείπουν από την υλοποίηση πολύ σημαντικά κομμάτια όπως η κλήση συναρτήσεων και η σύγκριση αποτελεσμάτων. Έτσι, θα οργανώσουμε τα βήματα της υλοποίησης ως εξής:

- Επιτυχία στην κατασκευή και εγκατάσταση (build) της Erlang/OTP σε μηχάνημα aarch64, με την παραγωγή του HiPE ενεργοποιημένη.
- Επιτυχής μεταγλώττιση απλού προγράμματος που επιστρέφει σταθερά. Δοκιμή προγράμματος ώστε να επιστρέφει σωστό αποτέλεσμα χωρίς διαφθορά μνήμης (segmentation faults). Αυτό το βήμα είναι ίσως το πιο μεγάλο, διότι τα αρχεία των διαφόρων βημάτων του backend γράφονται για πρώτη φορά. Είναι σημαντικό να συμπεριλάβουμε μόνο τα κομμάτια τους που είναι απαραίτητα ώστε να δημιουργηθεί το συγκεκριμένο πρόγραμμα, ώστε να αποφύγουμε την απρόσεκτη εισαγωγή μελλοντικών σφαλμάτων, για τα οποία δεν έχουμε ακόμα δοκιμή.
- Μεταγλώττιση και δοκιμή προγράμματος με απλή κλήση και επιστροφή. Κλήση με εισαγωγή μιας παραμέτρου. Σε αυτό το σημείο θα δοκιμαστεί μεγάλο κομμάτι του frame.
- Μεταγλώττιση και δοκιμή του basic_arith από τη σουίτα δοκιμών.
- Υλοποίηση για τη σουίτα δοκιμών basic_suite.
- Υλοποίηση για όλες τις σουίτες δοκιμών.

Παρόλο που ο όγκος κώδικα που ζητείται να μεταγλωττιστεί σε κάθε βήμα αυξάνεται εκθετικά, είναι αξιοπρόσεκτο ότι μεταξύ τους τα βήματα έχουν γενικά παρόμοιο χρόνο υλοποίησης.

Μπορούμε, γνωρίζοντας τη δομή ενός backend του HiPE, να προβλέψουμε και να κατηγοριοποιήσουμε και τα αρχεία που θα χρειαστεί να προσθέσουμε και να αλλάξουμε κατά

την εργασία. Εκτός από τα εγγενή κομμάτια του backend, θα χρειαστούν και μικροδιορθώσεις σε συγκεκριμένα σημεία του κορμού της υλοποίησης, ώστε να το δεχτεί και να το εκτελέσει επιτυχώς. Συγκεκριμένα έχουμε:

- Τα αρχεία ρύθμισης και παραγωγής (makefiles και configuration files). Τέτοια είναι τα erts/configure.in, erts/emulator/Makefile.in, erts/emulator/hipe/hipe_mkliterals.c, lib/hipe/Makefile, lib/hipe/aarch64/Makefile, και lib/hipe/regalloc/Makefile.
- Αρχεία του front end του HiPE, τα οποία θα εξυπηρετήσουν και θα καλέσουν εν τέλει το backend. Τέτοια είναι τα erts/emulator/hipe/hipe_arch.h, hipe_bif0.c, hipe_gc.h, hipe_mode_switch.c και .h, hipe_primops.h, hipe_process.h, hipe_risc_stack.c, τα lib/hipe/main/hipe.erl, main/hipe_main.erl, regalloc/hipe_aarch64_specifi.erl, rtl/hipe_rtl_arch.erl, και το lib/kernel/src/hipe_unified_loader.erl.
- Τα αρχεία του backend στον μεταγλωττιστή. Αυτά θα βρίσκονται στο lib/hipe/aarch64.
- Τα αρχεία του backend στον φορτωτή και το σύστημα εκτέλεσης. Αυτά θα είναι τα erts/emulator/hipe/aarch64<***>.

Επειδή προβλέπεται να έχουμε πολλά σφάλματα κατά την εργασία, και θα χρειαστεί να μελετήσουμε ανά διαστήματα παλαιότερο κώδικα που γράψαμε, θα μας εξυπηρετήσει να έχουμε ένα σύστημα ελέγχου εκδόσεων λογισμικού (Version Control System). Η υλοποίηση OTP είναι τοποθετημένη σε ένα αρχείο συστήματος git στο https://github.com/erlang/otp, οπότε είναι εύκολο να κάνουμε μία διακλάδωση ιστορικού (fork και νέο branch) στην οποία θα υλοποιήσουμε το backend μας. Δημιουργήσαμε τη διακλάδωση στο https://github.com/PwnzorBot4000/otp/tree/hipe/aarch64-testing. Με τη χρήση του git θα μπορούμε και να κάνουμε και συνεργατική ανάπτυξη και αξιολόγηση του κώδικα κατά την ώρα της συγγραφής.

Για τη δοκιμή της υλοποίησης, θα χρειαστούμε και ένα μηχάνημα αρχιτεκτονικής aarch64. Ως λύση χαμηλού κόστους βρέθηκε κατά τον καιρό της εργασίας η νέα συσκευή της εποχής, το Raspberry Pi 3B, το οποίο είναι εξοπλισμένο με έναν τετραπύρηνο 64-bit επεξεργαστή Cortex A53. Το λειτουργικό που χρησιμοποιήσαμε ήταν τα 64-bit Arch Linux for ARM. Για το μέγεθός του ήταν ικανό να μας εξυπηρετήσει με όχι κακή απόδοση, σε σχέση με τις ανάγκες μας.

5. Υλοποίηση

Καταρχάς λαμβάνουμε αντίγραφο της υλοποίησης Erlang/OTP μέσω git. Η έκδοση κατά τον καιρό της έναρξης της εργασίας ήταν η 'cd9b6371a13c37f8f82586fcd82f212d306d8fad', οπότε οι αλλαγές που περιγράφονται παρακάτω μπορούν να εφαρμοστούν σε εκείνη την έκδοση, γραμμή προς γραμμή.

Ως πρώτο βήμα, δοκιμάζουμε μία παραγωγή της OTP χωρίς καμία αλλαγή, για προσομοίωση:
./otp_build setup -a
Η υλοποίηση παράγεται, και με σωστή ρύθμιση του PATH, μπορούμε να γράψουμε και να εκτελέσουμε προγράμματα erlang με την εικονική μηχανή BEAM.
Στη συνέχεια δοκιμάζουμε να ενεργοποιήσουμε την παραγωγή του hipe, στο αρχείο ρυθμίσεων erts/configure.in. Η απλή μέθοδος είναι, όπου υπάρχει αναφορά σε αρχιτεκτονικές, να προστεθεί η aarch64 ως περίπτωση. Επίσης την προσθέτουμε σε ένα ακόμη σημείο όπου περιγράφονται αρχιτεκτονικές 64-bit, μιας και η aarch64 ανήκει σε αυτή την κατηγορία.

```
691: armv7hl)  ARCH=arm;;
692: aarch64)  ARCH=aarch64;;
2745: x86-linux|amd64-linux [...] |arm-linux|aarch64-linux|amd64-freebsd| [...]
3372: if test X$ac_cv_sizeof_void_p != X4 -a X$ARCH != Xamd64 [...]  -a X$ARCH != Xaarch64;
then
```

5.1. Τροποποίηση του φορτωτή

Αν προσπαθήσουμε να παράξουμε τώρα την υλοποίηση, κατά την παραγωγή του emulator το σύστημα θα διαμαρτυρηθεί για την έλλειψη κάποιων ορισμών. Πράγματι, τα αρχεία hipe_mode_switch.h και hipe_arch.h, ακόμα και αν δεν εκτελεστούν, για την παραγωγή τους χρειάζονται ορισμούς από την εκάστοτε αρχιτεκτονική. Η γενική μέθοδος εδώ είναι να ψάξουμε στα αρχεία του emulator για μακροεντολές #if defined(__architecture__), και να προσθέσουμε την περίπτωση της aarch64 αναλόγως. Αυτό όμως απαιτεί την ύπαρξη αρχείων ορισμών αρχιτεκτονικής. Αφού προς το παρόν το μόνο που μας ενδιαφέρει είναι η επιτυχής παραγωγή και όχι η εκτέλεση, θα αρκούσε αρχικά να αντιγράψουμε τα αντίστοιχα αρχεία της ARM αυτούσια. Όμως, κάποια από αυτά τα αρχεία περιλαμβάνουν μέσα τους συμβολική γλώσσα, και έτσι πρέπει αναγκαστικά να προσαρμοστούν ώστε να πετύχει η παραγωγή. Είναι κατανοητό ότι σε αυτό το βήμα οι γνώσεις μας περί του τρόπου λειτουργίας του συστήματος δεν είναι αρκετές για ενδελεχή συγγραφή αυτών των αρχείων, όμως όσο πιο προσεκτικά τα μελετήσουμε και προσαρμόσουμε τώρα, τόσο περισσότερα σφάλματα φόρτωσης και εκτέλεσης θα γλιτώσουμε στο μέλλον, τα οποία σε σχέση με τα σφάλματα μεταγλώττισης και κωδικοποίησης είναι αρκετά κρυπτικά και περίεργα ώστε να καταλάβει κανείς ότι προέρχονται από κάποιο σημείο του hipe εδώ, στον κώδικα του emulator. Ο πιο ασφαλής ίσως και απλός δρόμος είναι προς το παρόν να τοποθετήσουμε όσα κομμάτια των αρχείων δεν απαιτούνται για την παραγωγή σε σχόλια, και να προσαρμόσουμε με μεγάλη προσοχή όλα τα υπόλοιπα.

Αντιγράφουμε το hipe_arm.h ως hipe_aarch64.h. Βάζουμε την hipe_word32_address_ok σε σχόλιο, και προσαρμόζουμε το HIPE_RA_LSR_COUNT (3 στα 64 bit) και οπωσδήποτε το όνομα της αρχιτεκτονικής (`#define hipe_arch_name am_aarch64`), το οποίο θα μεταφραστεί σε erlang atom το οποίο αργότερα θα εμφανιστεί σε πολλά switches στο erlang κομμάτι του HiPE. Για να παραχθεί το atom, πρέπει να κατασκευάσουμε ακόμα το αρχείο - πίνακα ατόμων (atom table) hipe_aarch64.tab, με τον ορισμό ατόμου `atom aarch64` μέσα, και να το δηλώσουμε στο erts/emulator/Makefile.in, το οποίο θα απαιτήσει την ύπαρξή του κατά την παραγωγή: `590: HIPE_aarch64_TAB=hipe/hipe_aarch64.tab`.
Αντιγράφουμε το hipe_aarch64_glue.h, το οποίο ζητείται από το hipe_mode_switch.h, αφήνοντας έξω από σχόλια το `#include "hipe_risc_glue.h"`, το οποίο ξέρουμε ότι θα δουλεύει διότι η aarch64 είναι αρχιτεκτονική RISC. Αν κάτι πάει στραβά στο hipe_risc_glue, θα πρόκειται για κάποιο architecture switch ή #ifdef στο οποίο θα πρέπει να προσθέσουμε περίπτωση. Επίσης αφήνουμε και το `#include "hipe_aarch64_asm.h"` το οποίο θα αντιγράψουμε, διότι το χρησιμοποιεί ούτως ή άλλως το hipe_aarch64.h.
Προσθέτουμε περίπτωση __aarch64__ στο hipe_arch.h με `#include "hipe_aarch64.h"` και `#include "hipe_aarch64_asm.h"`, ώστε να ικανοποιηθούν τα απαιτούμενα για την παραγωγή. Επίσης στο hipe_mode_switch.h:

```
87: #elif defined(__aarch64__)
88: #include "hipe_aarch64_glue.h"
```

Και στο hipe_process.h, ενεργοποιούμε στο process structure το απαραίτητο πεδίο nra, native return address, το οποίο σε περίπτωση κλήσης συνάρτησης BEAM από το δυαδικό εκτελέσιμο, συγκρατεί τη διεύθυνση στον δυαδικό κώδικα όπου η εκτέλεση θα συνεχίσει όταν η συνάρτηση τελειώσει την εργασία της.

```
45: #if defined(__sparc__) || [...] || defined(__arm__) || defined(__aarch64__)
46: void (*nra)(void);
```

Στο erts/emulator/Makefile.in, προσθέτουμε την παραγωγή του hipe_aarch64_asm.h:

```
539: $(TTF_DIR)/hipe_arm_asm.h
540: $(TTF_DIR)/hipe_aarch64_asm.h
```

Αντιγράφουμε το hipe_arm_asm.m4 ως hipe_aarch64_asm.m4, τοποθετώντας όλο τον κώδικα σε σχόλια.
Αν δοκιμάσουμε παραγωγή τώρα, θα μας ζητηθεί να ορίσουμε τα NR_ARG_REGS και NR_LEAF_WORDS. Αυτά βρίσκονται στα hipe_aarch64_asm.m4 και hipe_aarch64_glue.h, οπότε τα βγάζουμε από σχόλια και τους βάζουμε σωστή ονομασία. Επόμενη προσπάθεια παραγωγής θα μας ζητήσει την hipe_word32_address_ok από το hipe_aarch64.h, οπότε τη βγάζουμε από σχόλιο και την ελέγχουμε αν είναι κατάλληλα γραμμένη για τη νέα παραγωγή.
Το επόμενο σφάλμα παραγωγής πρόκειται για μία έλλειψη στο hipe_gc.h. Συμπληρώνουμε τη μακροεντολή με περίπτωση __aarch64__ και αντιγράφουμε το hipe_aarch64_gc.h, προσαρμόζοντας κατάλληλα τα αρχεία που γίνονται #include.

Επόμενο απαιτούμενο για την παραγωγή είναι η ύπαρξη των αρχείων του φορτωτή, έτσι ώστε το mode switch να μπορεί να συνδεθεί μαζί τους για να κάνει τη φόρτωση και την εκτέλεση. Στο makefile.in, προσθέτουμε την παραγωγή τους:

```
970:    HIPE_aarch64_OBJS    =    $(OBJDIR)/hipe_aarch64.o    $(OBJDIR)/hipe_aarch64_glue.o
$(OBJDIR)/hipe_aarch64_bifs.o $(OBJDIR)/hipe_risc_stack.o $(HIPE_ARCH64_OBJS)
```

```
1077: $(OBJDIR)/hipe_aarch64_glue.o: hipe/hipe_aarch64_glue.S     $(TTF_DIR)/hipe_aarch64_asm.h
hipe/hipe_mode_switch.h $(TTF_DIR)/hipe_literals.h
$(TTF_DIR)/hipe_aarch64_bifs.S:     hipe/hipe_aarch64_bifs.m4         hipe/hipe_aarch64_asm.m4
hipe/hipe_bif_list.m4 $(TTF_DIR)/erl_bif_list.h hipe/hipe_gbif_list.h
$(OBJDIR)/hipe_aarch64_bifs.o:      $(TTF_DIR)/hipe_aarch64_bifs.S $(TTF_DIR)/hipe_literals.h
```

Στη συνέχεια, για την παραγωγή του hipe_aarch64_glue.S, θα χρειαστεί να ορίσουμε τους ειδικούς καταχωρητές που χρησιμοποιούνται κατά την εκτέλεση κώδικα μηχανής. Στο hipe_aarch64_asm.m4:

```
`#define P     x28'
`#define NSP   x27'
`#define HP    x26'
`#define TEMP_LR      x25'
```

Γνωρίζουμε από το procedure call standard ότι αυτοί είναι καταχωρητές σωζόμενοι από την καλούμενη συνάρτηση, οπότε μπορούμε να τους δεσμεύσουμε για αυτό το σκοπό. Ακόμα βγάζουμε από σχόλια, προσαρμόζοντας την assembly όπου χρειάζεται, τις `SET_GC_UNSAFE`, `SET_GC_SAFE`, `SAVE_CACHED_STATE`, `RESTORE_CACHED_STATE`, `LOAD_ARG_REGS`, `STORE_ARG_REGS`. Επίσης χρησιμοποιούμε και τον x24 ως προσωρινό καταχωρητή κατά την `hipe_arm_inc_stack`: `` `#define TEMP_ARG0      x24'. ``

Σε αυτό το σημείο μπορούμε να συμπεριλάβουμε το hipe_aarch64_glue.S στην παραγωγή. Το αντιγράφουμε, και αρχίζουμε να αναλύουμε τον κώδικά του.

Το πρώτο που βλέπουμε είναι ο κώδικας προετοιμασίας της στοίβας κατά την κλήση ή επιστροφή δυαδικού κώδικα. Στη στοίβα σπρώχνονται πρώτοι οι καταχωρητές σωζόμενοι από τον καλούμενο, οι οποίοι στο νέο call standard είναι οι x19-x28, και οι frame pointer (x29) και link register (x30). Επειδή ο stack pointer της aarch64 αυξομειώνεται σε βήματα των 16 bytes, χρησιμοποιούμε τις stp / ldp (store, load pair) για να τους εισάγουμε στη στοίβα, και μάλιστα μπορούμε να χρησιμοποιήσουμε την post-indexed μορφή, ώστε να γίνει αυτόματα και η αύξηση του stack pointer, για να αποφύγουμε αριθμητικά λάθη. Αφαιρούμε τον επιπλέον χώρο στη στοίβα που χρειαζόταν στην arm για κλήσεις με 5-6 παραμέτρους, αφού πλέον μπαίνουν όλες σε καταχωρητές. Με αυτά ακριβώς τα δεδομένα, ο χώρος που δεσμεύουμε στη στοίβα είναι ήδη στοιχισμένος (aligned), οπότε δεν χρειάζεται γέμισμα (padding). Τα ίδια κάνουμε και στον κώδικα επιστροφής, λίγο πιο κάτω. Εκεί, βλέπουμε ότι στον κώδικα arm φορτώνεται από τη στοίβα η παλιά διεύθυνση επιστροφής, στην οποία γίνεται άλμα. Όμως ο καταχωρητής στον οποίο φορτώνεται συμβαίνει να μην είναι ο link register, αλλά γίνεται απευθείας άλμα με φόρτωση του program counter. Ακολουθούμε την ίδια τακτική και διατηρούμε τα περιεχόμενα του link register (x30), διαλέγοντας ένα άλλο καταχωρητή μέσω του οποίου θα γίνει το άλμα. Η αρχιτεκτονική ορίζει τους x16 και x17 για χρήση σε άλματα, διαλέγουμε τον x17 σε περίπτωση που χρησιμοποιήσαμε τον x16 σε trampoline, οπότε ίσως θελήσουμε να δούμε τα περιεχόμενά του την ώρα της αποσφαλμάτωσης. Όχι σημαντικό μεν, αλλά αφού μας δίνεται η άνεση χώρου μπορούμε να την εκμεταλλευτούμε και έτσι.

Η υπόλοιπη εργασία στο αρχείο είναι κυρίως μετονομασίες από arm σε aarch64, αντικατάσταση παλιών καταχωρητών με νέους, και αντικατάσταση απευθείας εγγραφών του PC με br / blr / ret. Μετονομάζουμε και βγάζουμε από σχόλια και τις δηλώσεις στο hipe_aarch64_glue.h.

Σε αυτό το αρχείο βρίσκεται και η υλοποίηση της inc_stack. Είναι εγγενής στα αρχεία της αρχιτεκτονικής, διότι αυτή καλείται απευθείας από τον κώδικα του frame και έτσι χρειάζεται ένας

μικρός κώδικας προσαρμογής πριν κληθεί τελικά η hipe_inc_stack. Υλοποιώντας την, θα πρέπει να βάλουμε δηλώσεις της και στα hipe_aarch64.h (ως extern), hipe_aarch64.tab, και να κατασκευάσουμε το hipe_aarch64_primops.h με `PRIMOP_LIST(am_inc_stack_0, &hipe_aarch64_inc_stack)`, και να το συμπεριλάβουμε στην μακροεντολή επιλογής αρχείου ανά αρχιτεκτονική του hipe_primops.h.
Γενικά κατά την ώρα της προσαρμογής του glue.S θέλει προσοχή στο να καταλάβουμε ποιες συναρτήσεις προορίζονται για κλήση μιας συνάρτησης από το runtime προς το native ή από το native προς το runtime, ποιες για tailcall - κλήση χωρίς επιστροφή, και ποιες προορίζονται για επιστροφή από κλήση, στο runtime ή στο native, και να διαλέξουμε να χρησιμοποιήσουμε bl / blr, b / br ή ret αντίστοιχα.

Το βασικό αρχείο που παρέχει όλες τις architecture-specific συναρτήσεις τις οποίες χρειάζεται ο φορτωτής του HiPE για να εργαστεί, είναι το hipe_aarch64.c. Αντιγράφοντάς το εξηγούμε και προσαρμόζουμε τις συναρτήσεις που περιέχει.
Στην αρχή βρίσκεται η hipe_flush_icache_range, η οποία μετακινεί τον κώδικα που φορτώνεται στην μνήμη εκτέλεσης. Όπως εξηγήσαμε και στο κεφ. 2, θα πρέπει να χρησιμοποιήσουμε τον κώδικα που παρέχουν οι οδηγοί της aarch64, ο οποίος ελέγχει τον επεξεργαστή ώστε να πραγματοποιήσει αυτή τη λειτουργία [Βλέπε Cortex-A Series Programmer's Guide for ARMv8-A / Caches / Cache maintainance για σχετικό κώδικα, commit 336e45c για υλοποίηση]. Αν παραλείψουμε αυτό το βήμα θα εμφανιστούν τυχαία και απρόβλεπτα σφάλματα φόρτωσης, τα οποία θα κάνουν την αποσφαλμάτωση και τη σταθερή λειτουργία πολύ δύσκολη. Εν ολίγοις, η διαδικασία είναι: ερώτηση στον επεξεργαστή για το μέγεθος μίας μονάδας cache (cache line), διοχέτευση όσων cache lines χρειάζονται στην κύρια μνήμη, και φόρτωση των cache lines από την κύρια μνήμη στη μνήμη εκτέλεσης. Στην hipe_flush_icache_word βάζουμε μέγεθος προς φόρτωσης 8, καθώς αυτό είναι το μήκος λέξης.
Πιο κάτω βρίσκεται ο κώδικας παραγωγής trampolines. Τον αλλάζουμε όπως είχαμε προβλέψει, με μέγεθος trampoline 4 code words (16 bytes), 1 για τη φόρτωση, 1 για το άλμα και 2 για τη διεύθυνση. Ελέγχουμε με προσοχή τον κώδικα στις υπόλοιπες συναρτήσεις ώστε να χρησιμοποιεί τους σωστούς τύπους μεταβλητών, με το κατάλληλο μέγεθος. Η υλοποίηση OTP παρέχει τύπους Uint32, Uint64, Sint32 και Sint64 για να αποφευχθεί οποιαδήποτε σύγχυση. Ομοίως προσέχουμε και σε πράξεις αφαίρεσης τις μετατροπές μεταξύ τύπων ώστε να αποφύγουμε σφάλματα μετατροπής από προσημασμένο σε μη προσημασμένο μεγαλύτερου μεγέθους.
Στη συνέχεια συναντάμε τη συνάρτηση hipe_make_native_stub, η οποία λειτουργεί ως ενδιάμεσος μεταξύ native κώδικα και BIFs περιβάλλοντος εκτέλεσης. Ομοίως, προσαρμόζουμε τα μεγέθη, θυμούμενοι ότι οι λέξεις δεδομένων είναι διπλάσιες από τις λέξεις κώδικα, και μελετώντας τις κωδικοποιήσεις των εντολών που απαιτούνται στο Architecture Reference Manual, τις συνθέτουμε χρησιμοποιώντας bitwise OR με τις παραμέτρους τους και τις τοποθετούμε στις κατάλληλες θέσεις. Αλλάζουμε επίσης και τα όρια απόστασης για τα οποία θα χρειαστεί long jump (br) στα νέα, +-128MiB.
Όμοια όρια απόστασης που θα πρέπει να αλλάξουν βρίσκονται και στην hipe_patch_call, όπου πραγματοποιούμε παρόμοια φροντίδα. Ως τελευταία εργασία σε αυτό το αρχείο, θα πρέπει στην patch_b να αλλάξουμε τη διεύθυνση ενός άλματος σε αυτή που μας δίνει ο φορτωτής, χωρίς

όμως να αλλάξουμε τον τύπο του άλματος. Υπάρχουν δύο τύποι σταθερού άλματος, το b (0x14000000) και το bl (0x54000000). Για να μην επηρεάσουμε τον opcode της εντολής, φτιάχνουμε τις μάσκες 0xFC000000 και 0x03FFFFFF, και προσθέτουμε τον αρχικό opcode μεσω της πρώτης, και τη νέα διεύθυνση μέσω της δεύτερης.

Επόμενο βήμα που μας ζητείται είναι η ύπαρξη του hipe_aarch64_bifs.m4. Θα πρέπει να υλοποιήσουμε αρκετές από τις συναρτήσεις, οπότε κατά την αντιγραφή θα ακολουθήσουμε έναν αριθμό από βήματα για να βεβαιωθούμε ότι κατά την κλήση ενσωματωμένων συναρτήσεων δεν θα υπάρχουν προβλήματα.
Αρχικά, μετατρέπουμε όλους τους registers από arm σε aarch64: ο r1 γίνεται x1, ο lr γίνεται x30 και ούτω καθεξής. Όπου είναι δυνατόν, ας χρησιμοποιούνται οι μακροεντολές που ορίσαμε παραπάνω για τους ειδικούς καταχωρητές. Στη συνέχεια, θα πρέπει να φροντίσουμε όλες οι κλήσεις να γίνονται σύμφωνα με το νέο call standard. Συγκεκριμένα, ειδικά οι κλήσεις προς κώδικα C με 5 και 6 παραμέτρους πλέον δεν βάζουν τις παραμέτρους 5 και 6 στη στοίβα, αλλά χωράνε σε προσωρινούς καταχωρητές. Έτσι αφαιρούμε οποιαδήποτε αποθήκευση παραμέτρων στη στοίβα και την αντικαθιστούμε με εντολές μετακίνησης.
Προσαρμόζουμε το hipe_aarch64_asm.m4 ώστε να παρέχει τους ορισμούς που χρειάζεται το bifs.m4. Βγάζουμε από σχόλια και μετατρέπουμε την assembly στις `SAVE_CONTEXT_QUICK`, `RESTORE_CONTEXT_QUICK`, `SAVE_CONTEXT_BIF`, `RESTORE_CONTEXT_BIF`, `SAVE_CONTEXT_GC`, `RESTORE_CONTEXT_GC`, `NBIF_ARG`, `NBIF_RET`, `QUICK_CALL_RET`. Προσέχουμε οποιαδήποτε άλματα γίνονται με χρήση απευθείας μετακίνησης διεύθυνσης στον PC να αντικατασταθούν από τις νέες εντολές br.
Ως τελευταίο βήμα στη C μεριά, επειδή η aarch64 είναι 64-bit αρχιτεκτονική, χρειάζεται τον έλεγχο `hipe_word64_address_ok` στο hipe_aarch64.h. Τον προσθέτουμε βασιζόμενοι στην υπάρχουσα 32-bit συνάρτηση.

Αν τώρα προσπαθήσουμε παραγωγή, θα δούμε με ευχαρίστηση η μεταγλώττιση των αρχείων .c να πετυχαίνει, και την παραγωγή να σταματά πιο κάτω, κατά την εκτέλεση του εκτελέσιμου ERLC. Αυτό συμβαίνει διότι ο φορτωτής BEAM του συστήματος, lib/kernel/src/hipe_unified_loader.erl, χρειάζεται κάποιες προδιαγραφές ώστε να γνωρίζει τι αρχιτεκτονικής είναι ο κώδικας που διαβάζει. Θα πρέπει να επιλέξουμε, αν είναι δυνατόν με συνεννόηση με την ομάδα της OTP, ένα tag 4 χαρακτήρων, ο πρώτος H - συμβολίζοντας εκτελέσιμο HiPE, με το οποίο ο φορτωτής θα αναγνωρίζει ότι ο κώδικας είναι εκτελέσιμος HiPE aarch64. Επιλέξαμε τότε το tag “HAAR”. Το προσθέτουμε στους ορισμούς των tags, στο Architecture switch που αντιστοιχεί tags με atoms αρχιτεκτονικής - όπως το ορίσαμε στο hipe_aarch64.tab και στο hipe_aarch64.h, και βάζουμε περιπτώσεις στα switches του word size με 8 και στο needs trampolines με true.
Έχουμε τώρα τελειώσει την πρώτη φάση της εργασίας, καθώς η παραγωγή πετυχαίνει, και μπορούμε να εγκαταστήσουμε την υλοποίηση και να εκτελέσουμε προγράμματα erlang μέσω BEAM και πάλι χωρίς πρόβλημα. Οπότε πλέον είμαστε στο σημείο που αρχίσαμε, όμως με το υποσύστημα του HiPE εγκατεστημένο να περιμένει να φορτώσει και να εκτελέσει δυαδικά αρχεία, απλά χωρίς backend για την αρχιτεκτονική το οποίο μπορεί να τα παράξει. Έτσι, αν

προσπαθήσουμε να εκτελέσουμε hipe:c(module), ένα switch στο front end του HiPE θα αποτύχει, επειδή δεν έχει την περίπτωση για την αρχιτεκτονική στην οποία εργαζόμαστε.

5.2. Κατασκευή του backend

Για να κατασκευάσουμε ένα νέο HiPE backend, θα πρέπει να έχει το δικό του φάκελο στο lib/hipe. Φτιάχνουμε λοιπόν φάκελο aarch64, και τον προσθέτουμε στο lib/hipe/Makefile στη μεταβλητή HIPE_SUBDIRS. Μέσα του, αντιγράφουμε το αρχείο παραγωγής (Makefile) του arm backend, μιμούμενοι τη μορφή που έχουν τα υπόλοιπα backend, και αφαιρούμε ή βάζουμε σε σχόλια τα αρχεία που δεν υπάρχουν ακόμα. Για κάθε νέο αρχείο που θα δημιουργούμε, θα το προσθέτουμε σε αυτό το αρχείο παραγωγής. Επίσης θα φροντίζουμε να προσθέτουμε τα αρχεία που εξαρτώνται από το αρχείο ορισμών hipe_aarch64.hrl ή άλλα αρχεία του hipe στη λίστα εξαρτήσεων στο τέλος του αρχείου, παρατηρώντας πάλι από τα άλλα backends.
Δημιουργούμε τα αρχεία ορισμών hipe_aarch64.hrl και βοηθητικών συναρτήσεων hipe_aarch64.erl, αφήνοντάς τα προς το παρόν σχετικά άδεια. Οποιαδήποτε βοηθητική συνάρτηση ή ορισμό χρειαζόμαστε, θα τα προσθέτουμε σε αυτά τα αρχεία, κάνοντας μετατροπές εκείνη την ωρα.
Στο frontend του HiPE, αφού απενεργοποιήσουμε στις βελτιστοποιήσεις την παραγωγή πράξεων κινητής υποδιαστολής (-inline-fp), αντιγράφοντας την περίπτωση της arm, το πρώτο αρχείο που αποτυγχάνει είναι το hipe_rtl_arch.erl. Αυτό ζητάει κατά την ώρα της παραγωγής της rtl πληροφορίες για τους καταχωρητές της αρχιτεκτονικής. Θα πρέπει να κατασκευάσουμε το hipe_aarch64_registers, προσθέτοντάς το και στο νέο Makefile, μιμούμενοι τη δομή των άλλων αρχιτεκτονικών, και προσέχοντας τις ιδιαιτερότητες της aarch64. Είναι πιθανό σε σημεία όπως αυτό η μεταγλώττιση να αποτύχει πολύ πριν αρχίσει να εκτελείται το backend, διότι η rtl έχει ανάγκη να μάθει κάτι σχετικό με τους καταχωρητές της aarch64, και να μην υπάρχει αντίστοιχη συνάρτηση στο registers.erl. Σε αυτή την περίπτωση, συμπληρώνουμε την εκάστοτε έλλειψη για να συνεχίσουμε. Για την `is_precoloured_gpr`, θα πρέπει να ορίσουμε τον `LAST_PRECOLOURED`, δηλαδή το μέγιστο αριθμό που αντιστοιχεί σε πραγματικό καταχωρητή. Η rtl χρειάζεται ακόμα για να εκτελεστεί τις `first_virtual` και `live_at_return`, τις οποίες θα πρέπει να προσθέσουμε στα switches στο hipe_rtl_arch.erl. Η first_virtual αντιστοιχεί (συνήθως) σε `?LAST_PRECOLOURED + 1`, και η live_at_return ελέγχει αν ο καταχωρητής είναι κάποιος από τους heap (x26), stack (x27) ή proc (x28) pointers (όπως είδαμε παραπάνω), οπότε τους προσθέτουμε ως μακροεντολές που τους αντιστοιχούν σε αριθμούς καταχωρητών, και τους τοποθετούμε στο switch της συνάρτησης.
Το επόμενο αρχείο που αποτυγχάνει είναι το main/hipe_main.erl, στο σημείο όπου αρχίζει η εκτέλεση του backend. Το αρχείο που κατευθύνει το μεγαλύτερο μέρος της μετάφρασης είναι το hipe_aarch64_main.erl. Το αντιγράφουμε και μετονομάζουμε τις κλήσεις με διακριτικά aarch64. Το ότι δεν υπάρχουν ακόμα δεν μας επηρεάζει, καθώς δεν γίνονται σφάλματα παραγωγής, και η διαδικασία γίνεται σε στάδια, οπότε όταν εμφανιστεί σφάλμα ότι κάποια από αυτές τις κλήσεις θα ξέρουμε ότι έχουμε ολοκληρώσει το προηγούμενο στάδιο και είμαστε έτοιμοι να την υλοποιήσουμε. Ύστερα, μπορούμε στο switch του main/hipe_main.erl να προσθέσουμε την περίπτωση της αρχιτεκτονικής μας, καλώντας την `hipe_aarch64_main:rtl_to_aarch64`.

5.2.1. Παραγωγή εγγενών δομών

Η πρώτη κλήση στη σειρά της rtl_to_aarch64 αφορά το αρχείο hipe_rtl_to_aarch64. Το δημιουργούμε, και αντιγράφουμε μόνο το κομμάτι της translate που χρειαζόμαστε. Θέτουμε [] στα formals, καθώς η πρώτη συνάρτηση που θα υλοποιήσουμε δεν έχει παραμέτρους, οπότε δε θα χρειαστεί να υλοποιήσουμε τις συναρτήσεις επεξεργασίας παραμέτρων. Όμως, πρέπει να είμαστε έτοιμοι να συμπληρώσουμε αυτό το σημείο όταν θα εμφανιστεί πλέον συνάρτηση με παραμέτρους, οπότε ακριβώς από πάνω εισάγουμε τον έλεγχο (assertion) `[] = hipe_rtl:rtl_params(RTL)`, ώστε η μεταγλώττιση να μας ειδοποιήσει ότι αυτή η λειτουργία δεν έχει ακόμα υλοποιηθεί - έτσι ώστε να μην προχωρήσει στην παραγωγή εσφαλμένου κώδικα.
Στην translate, η κλήση που είναι απαραίτητο να εκτελεστεί είναι η conv_insn_list, η οποία παίρνει τις εντολές rtl και μέσω της conv_insn καλεί τις συναρτήσεις που τις μεταφράζουν σε εγγενείς δομές. Αντιγράφουμε την conv_insn_list, και την conv_insn άδεια, χωρίς επιλογές στο switch, καθώς δεν ξέρουμε ακόμα το πρόγραμμα προς δοκιμή τι θα χρειαστεί για τη μεταγλώττισή του. Κατασκευάζουμε ένα απλό erlang module το οποίο θα μας δώσει την απάντηση:

```
-module(test).
-export(test/0).
test() -> 42.
```

Το απλό αυτό πρόγραμμα φαίνεται να χρησιμοποιεί τέσσερις εντολές rtl: Την enter, για κλήση χωρίς επιστροφή, η οποία πιθανότατα χρησιμοποιείται από τη module_info, μία συνάρτηση που ο HiPE τοποθετεί υποχρεωτικά σε κάθε module που παράγει, η οποία όταν εκτελείται δίνει τον έλεγχο στο περιβάλλον εκτέλεσης, ζητώντας του να δώσει πληροφορίες για αυτό το module. Την label, η οποία θα δείχνει τη θέση όπου ο φορτωτής θα τοποθετήσει κάποιο απαραίτητο atom, την load_atom, η οποία είναι η πραγματική εντολή φόρτωσης του ατόμου στη μνήμη, και τη return, η οποία είναι η εντολή επιστροφής μιας τιμής ως αποτέλεσμα στο σύστημα εκτέλεσης, και ο τερματισμός λειτουργίας της συνάρτησης.

Η πιο απλή από τις εντολές για να αντιμετωπίσουμε είναι ίσως η label. Συμβουλευόμενοι και τον arm κώδικα, προσθέτουμε την conv_label με μία κλήση στην `hipe_aarch64:mk_label(hipe_rtl:label_name(I))`, την οποία mk_label προσθέτουμε στο hipe_aarch64, προσθέτοντας και τον ορισμό της εγγενούς δομής #label στο .hrl, με `-record(label, {label}).`.

Για να μετατρέψουμε την load_atom, μιας και το άτομο φορτώνεται σε κάποια μεταβλητή (καταχωρητή ή μνήμη) προορισμού, χρειαζόμαστε τη συνάρτηση conv_dst. Την αντιγράφουμε και μελετάμε αν χρειάζεται τροποποιήσεις ή προσθήκες. Γενικά, χρειάζεται να προσθέσουμε και τις vmap_lookup, vmap_empty, vmap_bind για την επεξεργασία της δομής dictionary που αντιστοιχεί μεταβλητές rtl σε προσωρινές τιμές aarch64. Για την παραγωγή προσωρινών τιμών (temps), χρειαζόμαστε την `hipe_aarch64:mk_new_temp`, την οποία τοποθετούμε και μετατρέπουμε, τοποθετώντας και την εγγραφή aarch64_temp στο .hrl. Η conv_dst τις χρησιμοποιεί δημιουργώντας, αν χρειάζεται, μία νέα προσωρινή τιμή, και αντιστοιχίζοντάς τη με τη μεταβλητή rtl που παίρνει ως είσοδο.

Αφού γίνει η συσχέτιση της προσωρινής τιμής - προορισμού, η conv_load_atom προχωρά στην παραγωγή μίας εγγενούς δομής που αντιπροσωπεύει τη φόρτωση από τη μνήμη, ή από άλλο καταχωρητή, γενικά από μία προσωρινή τιμή. Αυτή η δομή φόρτωσης θα μεταφραστεί στην ακολουθία εντολών που θα τοποθετήσουν το atom στον καταχωρητή προορισμού. Γράφουμε λοιπόν το `hipe_aarch64:mk_pseudo_li`, κάνοντας τις απαραίτητες μετατροπές, και τοποθετούμε το #psuedo_li record στο .hrl.

Επόμενη έχουμε την conv_enter, η οποία θα παράξει την ακολουθία για ένα tailcall, κλήση χωρίς επιστροφή. Αυτή παρατηρούμε ότι χρησιμοποιεί τις συναρτήσεις conv_src και conv_fun, που είναι παρόμοιες με την conv_dst, απλά για μεταβλητές προέλευσης και για κλήσεις αντίστοιχα. Τοποθετούμε λοιπόν τις conv_src_list, conv_src, conv_fun, και conv_mfa, με παρόμοια τακτική. Αυτές χρειάζονται τις `hipe_aarch64:mk_prim` και `hipe_aarch64:mk_mfa`, οι οποίες είναι αρκετά σαφείς στην τοποθέτηση, με τα #aarch64_prim και #aarch64_mfa records αντίστοιχα στο hrl. Ομοίως για την mk_enter τοποθετούμε τις `hipe_aarch64: mk_pseudo_tailcall_prepare`, `hipe_aarch64: mk_pseudo_tailcall` με τα αντίστοιχα records.

Τελευταία είναι η return, η οποία θα τοποθετήσει την (συνοδευόμενη από κάποιο δυαδικό tag) τιμή 42 στον καταχωρητή τιμής επιστροφής, και θα επιστρέψει τη ροή εκτέλεσης στο καλών πρόγραμμα. Είναι απαραίτητο λοιπόν να ορίσουμε τον καταχωρητή τιμής επιστροφής, στη συνάρτηση `hipe_aarch64_registers:return_value()`. Αυτή χρησιμοποιείται από την mk_rv, η οποία καλείται από την conv_return. Αντιγράφοντας αυτές τις δύο, βλέπουμε με προσπάθεια μεταγλώττισης ότι η conv_return χρειάζεται και την mk_move, για να κάνει φόρτωση άμεσης τιμής με τη mk_li, καθώς και μια εγγενή δομή που αντιπροσωπεύει την επιστροφή από το καλούμενο πρόγραμμα, και θα μεταφραστεί σε ret. Τοποθετούμε λοιπόν την mk_move με μόνη περίπτωση στο switch αυτή που αφορά την mk_li, και την `hipe_aarch64: mk_pseudo_blr` (που αργότερα διαπιστώθηκε ότι θα ήταν καλό να ονομαστεί pseudo_ret για να αποφευχθεί σύγχυση).
Η `hipe_aarch64:mk_li` δεν είναι απλή στην υλοποίηση. Αφορά τη μετατροπή άμεσης σταθεράς σε μορφή η οποία μπορεί να ενσωματωθεί σε εντολή μετακίνησης, οπότε πρέπει να ακολουθήσουμε τις αρχιτεκτονικές ιδιαιτερότητες που μελετήσαμε παραπάνω. Μπορούμε μέχρι ένα σημείο να ακολουθήσουμε την τακτική της arm υλοποίησης, η οποία έχει τις try_aluop_imm η οποία επιστρέφει την προσαρμοσμένη τιμή αν χωράει στην εντολή, αλλιώς δίνει [], την mk_li η οποία δίνει την προσαρμοσμένη εντολή μετακίνησης αν η try_aluop_imm πετύχει την προσαρμογή, αλλιώς δίνει ακολουθία φόρτωσης από τη μνήμη σε δύο βήματα μέσω προσωρινού καταχωρητή, την invert_alulop_imm η οποία, αν η πρώτη μετατροπή αποτύχει, δοκιμάζει να κάνει μετατροπή της ανεστραμμένης τιμής με αναστρέφουσα εντολή (πχ. αν η εντολή mov 111111..11100 δεν χωράει σε 32 bits, ίσως να χωράει η movn 011), και τέλος την imm_to_am1 η οποία δοκιμάζει να χωρέσει την τιμή στην εντολή, συνοδεύοντάς τη με απαραίτητες παρενέργειες αν χρειάζεται (πχ. ολίσθηση). Η ονομασία της εντολής προέρχεται στην πραγματικότητα από το ‘addressing mode 1’ της arm, που περιλαμβάνει τις 11 μορφές εντολών με παράμετρο η οποία μπορεί να υποστεί περιστροφή κατά συνοδευόμενη άμεση τιμή ή καταχωρητή. Η δομή am1 στο arm backend περιλαμβάνει την παράμετρο μιας εντολής και όλες τις παρενέργειες που γίνονται πάνω σε αυτή. Στην aarch64 δεν υπάρχει τέτοια

κατηγοριοποίηση, οπότε απλά θα συνοδεύουμε τις εντολές μας με δύο νοητά ‘addressing modes’, το am1, το οποίο αφορά τις παραμέτρους που περιέχουν δεδομένα, και την am2, που θα αφορά αργότερα τις παραμέτρους που περιέχουν διευθύνσεις.
Η μετατροπή της άμεσης σταθεράς που κάνουμε στην imm_to_am1 πραγματοποιείται με μία σύντομη επανάληψη, όπου η σταθερά ελέγχεται σε κάθε βήμα αν χωράει στην εντολή (πρέπει να είναι 16 bits για εντολές μετακίνησης), και αν δε χωράει, δοκιμάζεται να ολισθησει δεξιά κατά όσες θέσεις επιτρέπει η εντολή, μέχρι να χωρέσει. Αν υπάρχει απώλεια λιγότερο - σημαντικών δεδομένων κατά την ολίσθηση, ή αν ο αριθμός ολισθήσεων γίνει μεγαλύτερος από όσο υποστηρίζει η εντολή, η μετατροπή αποτυγχάνει. Οι εντολές μετακίνησης επιτρέπουν μέχρι 3 ολισθήσεις των 16 bit, οπότε φροντίζουμε να ενημερώσουμε την imm_to_am1 για τον τύπο της εντολής πριν τη μετατροπή με μία επιπλέον παράμετρο, ώστε να μπορεί να αποφανθεί αν η μετατροπή γίνεται.

Αφού γράψουμε την mk_li, προσθέτουμε και τη δομή #move καθώς και την mk_move την οποία χρησιμοποιεί σε περίπτωση επιτυχούς μετατροπής. Ύστερα από αυτές τις διαδικασίες, προσπάθεια μεταγλώττισης θα επιβεβαιώσει τη λειτουργία του hipe_rtl_to_aarch64, αποτυγχάνοντας στο επόμενο βήμα, το hipe_aarch64_cfg.

5.2.2. Ανάθεση μεταβλητών σε καταχωρητές

Αφού πλέον οι εντολές rtl της μεταγλωττιζόμενης συνάρτησης έχουν μεταφραστεί σε εγγενείς δομές, είναι καιρός να αντιστοιχίσουμε τους καταχωρητές που κάθε μία από αυτές τις εγγενείς δομές θα προσπελάσει. Για την έναρξη αυτής της διαδικασίας, χρειάζεται να δημιουργήσουμε μία ‘δομή διαρρύθμισης’ (configuration) για αυτή τη συνάρτηση, η οποία θα συνοδεύει και θα περιέχει τις εγγενείς δομές της μέχρι και το βήμα της βελτιστοποίησης, ώστε να παρέχει τις απαραίτητες πληροφορίες. Αυτή δημιουργείται στο `hipe_aarch64_cfg:init`, και έχει μέσα πληροφορίες όπως τις παραμέτρους της συνάρτησης, τις εγγενείς δομές, την ταυτότητά τις (MFA - module, function, arity), το εύρος κάθε μεταβλητής και ετικέτας, και αν είναι τελική συνάρτηση (leaf) ή ανώνυμη λάμδα (closure). Αντιγράφουμε λοιπόν το αρχείο και την init, και ό,τι αυτή χρειάζεται.
Όταν έχουμε τοποθετήσει αρκετούς ορισμούς στο cfg.erl ώστε να εξαλειφθούν τα σφάλματα μεταγλώττισης, το βήμα init θα περάσει επιτυχώς, και η μεταγλώττιση θα σταματήσει στο βήμα ra.

Τα απαραίτητα αρχεία και τμήματα κώδικα που θα πρέπει να έχουμε για να ξεπεράσουμε αυτό το βήμα, είναι: Καταρχάς στο hipe_aarch64_cfg, για κάθε εγγενή δομή που δημιουργείται στην προηγούμενη φάση, περιπτώσεις στη συνάρτηση is_branch για το αν αυτές οι εγγενείς δομές παράγουν άλμα, και για όσες από αυτές η is_branch επιστρέφει true, περιπτώσεις στη συνάρτηση branch_successors η οποία εξάγει τις ετικέτες όπου το άλμα κάθε μίας από αυτές τις εγγενείς δομές μπορεί να καταλήξει. Το αρχείο hipe_aarch64_defuse, με τις συναρτήσεις insn_def_gpr, η οποία εξάγει από τις εγγενείς δομές τους προορισμούς, και insn_use_gpr, που εξάγει τις παραμέτρους - πηγές, για να διαπιστωθεί ποιοι καταχωρητές επηρεάζονται. Πληροφορίες στο hipe_aarch64_registers για τους καταχωρητές της αρχιτεκτονικής, όπως ποιοι καταχωρητές είναι ελεύθεροι προς χρήση από τον αλγόριθμο ανάθεσης (allocatable), και ποιοι

θα χρησιμοποιούνται για πέρασμα παραμέτρων (στην erlang - όχι σύμφωνα με το procedure call standard απαραίτητα) στη συνάρτηση arg / args, και τους κατάλληλους ορισμούς στο erts/ emulator/ hipe/ hipe_mkliterals.c για τα σύμβολα που το registers.erl χρειάζεται για να διαπιστώσει αυτή την πληροφορία (`AARCH64_NR_ARG_REGS`, το οποίο πρέπει να οριστεί στο `hipe_aarch64_asm`, το οποίο πρέπει να συμπεριληφθεί στο hipe_mkliterals.c με τον ίδιο τυπικό τρόπο που γίνεται και στις υπόλοιπες αρχιτεκτονικές - #undef's). Το αρχείο hipe_aarch64_liveness_gpr με όλα όσα χρειάζεται για να δουλέψουν οι συναρτήσεις analyze και liveout που χρειάζεται, το οποίο θα πρέπει να το καλέσουμε στην `hipe_aarch64_specific: analyze`. Τις υπόλοιπες συναρτήσεις που θα χρειαστεί το `hipe_aarch64_specific` ώστε να λειτουργήσει, οι οποίες πιθανόν καλούν κάποιο από τα παραπάνω αρχεία, ένα από τα οποία είναι το αρχείο για αντικατάσταση προσωρινών τιμών με έλεγχο τύπων `hipe_aarch64_subst`, το οποίο θα πρέπει να δημιουργήσουμε. Και τέλος το αρχείο hipe_aarch64_ra, που καλεί τον κατάλληλο αλγόριθμο ανάθεσης καταχωρητών ανάλογα με το επίπεδο βελτιστοποίησης.
Γενικά, κάθε φορά που προσθέτουμε νέα εγγενή δομή θα πρέπει να προσθέτουμε περιπτώσεις στις `hipe_aarch64_defuse: insn_def_gpr`, `insn_use_gpr`, `hipe_aarch64_cfg: is_branch` και `branch_successors`, και όποιες συνοδευτικές συναρτήσεις αυτές χρειάζονται για να λειτουργήσουν. Και κάθε φορά που δοκιμάζουμε νέο επίπεδο βελτιστοποίησης μπορεί να χρειαστεί να προσθέσουμε περίπτωση στο switch της `hipe_aarch64_ra: ra`, που μπορεί να σημαίνει και δημιουργία νέου αρχείου για τον αλγόριθμο ανάθεσης καταχωρητών, hipe_aarch64_ra_<algo>.erl. Γενικά σε αυτή τη φάση της υλοποίησης, όπου υπάρχουν switches με εγγενείς δομές ή άλλες μη πλήρως υλοποιημένες περιπτώσεις, αποφεύγουμε να τοποθετούμε περιπτώσεις default (_ -> …), ώστε να αντιλαμβανόμαστε εγκαίρως τις νέες ανάγκες, αλλά τοποθετούμε όλες τις περιπτώσεις που κανονικά θα έπεφταν στην προκαθορισμένη (default) περίπτωση μία-μία ξεχωριστά με ακριβώς τον ίδιο κώδικα.

Το βήμα της ανάθεσης καταχωρητών έχει κάποια τελειωτικά βήματα μετά την ανάθεση ώστε οι εγγενείς δομές να ετοιμαστούν κατάλληλα για τα επόμενα βήματα. Μετά από τα παραπάνω, θα χρειαστεί να δημιουργήσουμε το αρχείο hipe_aarch64_ra_postconditions, το οποίο ανάλογα το πως έγινε η ανάθεση των καταχωρητών, ίσως χρειαστεί να τροποποιήσει ή να προσθέσει εγγενείς δομές στο πρόγραμμα, για παράδειγμα εάν μία μεταβλητή τοποθετηθεί στη στοίβα, θα χρειαστούν επιπλέον εντολές φόρτωσης. Αντιγράφουμε το αρχείο κάνοντας απαραίτητες μετατροπές, αφήνοντας όμως μόνο τον κώδικα που χρειάζονται οι εγγενείς δομές που τοποθετήσαμε στα παραπάνω αρχεία για να πετύχει αυτό το βήμα, #move, #pseudo_li, #pseudo_tailcall, #pseudo_tailcall_prepare, #pseudo_blr, ώστε να πετύχει η κλήση που ζητά τις δυνατότητες αυτού του αρχείου, η `hipe_aarch64_specific: check_and_rewrite`. Συναρτήσεις που χρειάζονται για την εξυπηρέτηση αυτών των εγγενών δομών είναι οι fix_funv, fix_src<#>, και fix_dst. Γενικά το αρχείο δεν έχει ιδιαίτερες αλλαγές από την arm έκδοση, εκτός ίσως από την τακτοποίηση στις περιπτώσεις fix_am1 και fix_am2, για να εξυπηρετήσουν τις καινούριες δομές παραμέτρων.
Όμοια είναι και η μετατροπή και τοποθέτηση του hipe_aarch64_ra_finalize. Αφού αντιγράψουμε τις απαραίτητες συναρτήσεις, καθώς τοποθετούμε νέες περιπτώσεις εγγενών δομών στο switch

της do_insn, η μεταγλώττιση θα προχωράει όλο και πιο πολύ, μέχρι το βήμα ανάθεσης καταχωρητών να τελειώσει, και η διαδικασία να αποτύχει στην κλήση του βήματος frame.

5.2.3. Κώδικας διαχείρισης στοίβας

Ο κώδικας διαχείρισης στοίβας, ο οποίος βρίσκεται στην αρχή κάθε συνάρτησης και σε ενδιάμεσά της απαραίτητα σημεία, αποτελείται από έναν ελάχιστο αριθμό εντολών ο οποίος αναλαμβάνει να την προετοιμάσει και να σπρώξει (push) ή να τραβήξει (pop) τιμές σε αυτή, όπου αυτό χρειάζεται. Για να πραγματοποιηθούν αυτές οι λειτουργίες χρειαζόμαστε πριν από όλα δύο εντολές της αρχιτεκτονικής, την str (αποθήκευση καταχωρητή στη μνήμη) και ldr (φόρτωση από μνήμη). Στο στάδιο αυτό θα αρκεστούμε στην ύπαρξη των εγγενών δομών που τις αντιπροσωπεύουν. Πρώτη απαραίτητη είναι η δομή #store, και η συνάρτηση κατασκευής της (constructor) `hipe_aarch64: mk_store`.
Η υλοποίηση της mk_store έχει μερικές διαφορές από την arm έκδοση. Καταρχάς, ανάλογα το μέγεθος του δεδομένου που αποθηκεύεται στη μνήμη η str έχει και διαφορετική κωδικοποίηση, οπότε είναι απαραίτητο η εγγενής δομή #store να συνοδεύεται από το διακριτικό μεγέθους που χρειαζόμαστε (8, 16, 32, 64 bit). Στην υλοποίησή μας το ενσωματώσαμε στον opcode της δομής, και ο συμβολομεταφραστής αναλαμβάνει να συμπεράνει από αυτόν την εντολή (str, strh, strb) και την κωδικοποίηση.
Ακόμα, έχει αλλάξει το εύρος στο οποίο η εντολή έχει πρόσβαση. Πλέον αποθήκευση μπορεί να γίνει μόνο σε θετική απόσταση 4096 μονάδων (words, word32s, halfwords, bytes) προς τα μπροστά, ή αστοίχιστη (unaligned) σε απόσταση +-256 bytes. Πρέπει λοιπόν να έχουμε έλεγχο ανάλογα τη σταθερά απόστασης (offset) και το μέγεθος φόρτωσης. Χρησιμοποιούμε τη νέα δομή παραμέτρου am2, που περιέχει αναφορά σε διεύθυνση, και την παράγουμε αν η απόσταση είναι εντός ορίων, αλλιώς παράγουμε πρόσβαση στη μνήμη μέσω ενδιάμεσου καταχωρητή - 2 εντολές. Φυσικά, σε αυτό το στάδιο οι καταχωρητές έχουν αντιστοιχιστεί, και είναι αδύνατον να διαλέξουμε κάποιο νέο για προσωρινό καταχωρητή. Για αυτή την περίπτωση, έχουμε ορίσει έναν από τους καταχωρητές στο hipe_aarch64_registers ως non-allocatable (τον temp2, x29), τον οποίο έχουμε στην διάθεσή μας τώρα να χρησιμοποιήσουμε ως προσωρινό καταχωρητή χωρίς να επηρεάσουμε το υπόλοιπο πρόγραμμα.
Η υλοποίηση της mk_load, για την ανάγνωση τιμών στη στοίβα, είναι παρόμοια με της mk_store, με την ίδια νοοτροπία αποθήκευσης του μεγέθους δεδομένου στην εγγενή δομή για χρήση της κατάλληλης κωδικοποίησης. Η μόνη ίσως σημαντική διαφορά είναι ότι σε περίπτωση απόμακρης φόρτωσης με ενδιάμεσο προσωρινό καταχωρητή, όπου έχουμε σε καταχωρητές μία διεύθυνση βάσης και μία απόσταση, αν ο καταχωρητής φόρτωσης του αποτελέσματος δεν είναι ο καταχωρητής που περιέχει την διεύθυνση βάσης, μπορεί για ενδιάμεσος προσωρινός καταχωρητής για συγκράτηση της απόστασης να χρησιμοποιηθεί ο καταχωρητής αποτελέσματος. Έτσι, κατά τη λειτουργία, θα μπει η διεύθυνση βάσης σε ένα καταχωρητή Α και η απόσταση, λόγω μεγέθους, σε έναν Β, και αφού γίνει η φόρτωση, το αποτέλεσμα θα μπει στον Β, χρησιμοποιώντας έτσι μόνο δύο καταχωρητές αντί για τρεις.

Κατά την έναρξη της εκτέλεσης της συνάρτησης, ο κώδικας προετοιμασίας στοίβας θα πρέπει να συγκρίνει το διαθέσιμο χώρο στη στοίβα σε σχέση με το πόσο θα χρειαστεί η συνάρτηση για να δουλέψει, και να ζητήσει αύξηση της στοίβας από το σύστημα αν χρειάζεται. Για να το κάνει

αυτό, είναι απαραίτητη η χρήση μίας εντολής σύγκρισης και άλματος. Υλοποιούμε λοιπόν στο hipe_aarch64.erl τις mk_cmp και mk_pseudo_bc, και επειδή η σύγκριση είναι στην πράξη μια αφαίρεση, τις mk_alu και mk_addi. Ομοίως τις εγγενείς δομές #alu, #cmp, #pseudo_bc. Η κλήση της inc_stack του συστήματος χρειάζεται την εντολή κλήσης με επιστροφή bl, και την παραγωγή stack descriptors μετά την επιστροφή, και η εντολή σύγκρισης χρειάζεται ύστερα το άλμα σε γνωστή ετικέτα, οπότε προσθέτουμε και τις mk_bl, mk_b_label, mk_sdesc, και #bl, #b_label και #aarch64_sdesc. Ο κώδικας για την υλοποίηση όλων αυτών είναι σχετικά απλός και δεν χρειάζεται ιδιαίτερες αλλαγές, καθώς το μόνο που κάνει είναι να παράγει τις εγγενείς δομές, γεμίζοντάς τις με πληροφορίες, εκτός ίσως από τη mk_addi, η οποία πραγματοποιεί την αφαίρεση, και έχει λογική υλοποίησης αντίστοιχη με ό,τι είδαμε ως τώρα στις mk_load και mk_store. Η mk_pseudo_bc έχει ακόμα την ιδιότητα να χρησιμοποιεί την αντίθετη συνθήκη σε άλματα (πχ. A > B αντί για ! (A <= B)) εάν αυτό είναι πιο αποδοτικό σε σχέση με την υποθετική εκτέλεση εντολών που κάνει ο επεξεργαστής (branch prediction). Η τοποθέτηση και αυτής δεν παρουσιάζει δυσκολία καθώς τα ονόματα (και οι 4-bit κωδικοποιήσεις) των συνθηκών σύγκρισης δεν έχουν αλλάξει από την εποχή της arm.
Σε περίπτωση που ο αριθμός των χρησιμοποιούμενος καταχωρητών είναι αρκετά μεγάλος ώστε να χρειαστεί να χρησιμοποιηθεί και ο καταχωρητής που περιέχει τη διεύθυνση επιστροφής, ο link register, ο κώδικας διαχείρισης στοίβας πρέπει να προνοήσει τοποθετώντας τον lr στη στοίβα μέχρι πλέον να χρειαστεί να τον χρησιμοποιήσει για επιστροφή. Για την παραγωγή του κώδικα που εκτελεί αυτή τη λειτουργία, προσθέτουμε ακόμα στο hipe_aarch64.erl τις mk_lr, mk_mflr, mk_mtlr, mk_move.

Αφού έχουμε τοποθετήσει τα απαιτούμενα για να εκτελεστεί ο κώδικας διαχείρισης, είμαστε έτοιμοι να τον προσθέσουμε. Κατασκευάζουμε το αρχείο hipe_aarch64_frame.erl, εξετάζοντας προσεκτικά τα περιεχόμενά του. Το πρώτο αξιοπρόσεκτο σημείο είναι η do_insn, η οποία εξετάζει μία μία τις εγγενείς δομές και παράγει κώδικα διαχείρισης στοίβας όπου χρειάζεται. Προς το παρόν, οι μόνες δύο εγγενείς δομές που χρειαζόμαστε είναι η pseudo_blr (επιστροφή από συνάρτηση) και η pseudo_tailcall (κλήση χωρίς επιστροφή). Για όλες τις υπόλοιπες εγγενείς δομές που έχουμε, που κανονικά πέφτουν στην περίπτωση default, αποφεύγουμε την περίπτωση default και προσθέτουμε μία-μία περιπτώσεις με την προκαθορισμένη ενέργεια, για λόγους αποσφαλμάτωσης όπως εξηγήσαμε και πιο πάνω.
Ο τρόπος με τον οποίο παράγεται ο κώδικας διαχείρισης στοίβας κατά την ανάλυση είναι η δημιουργία μιας δομής στο ξεκίνημα, με όνομα context, η οποία περιέχει πληροφορίες για τη στοίβα όπως το μέγεθός της, αν χρησιμοποιήθηκε ο link register (από το regalloc), και μία αντιστοίχιση των μεταβλητών που τοποθετούνται στη στοίβα, με τις θέσεις τους σε σχέση με την κορυφή της στοίβας. Το μόνο λοιπόν που χρειάζεται να κάνει ο κώδικας για την περίπτωση της pseudo_blr είναι η παραγωγή μιας εγγενούς δομής πριν την εντολή, που θα επαναφέρει τη διεύθυνση επιστροφής στον lr (αν αυτή βρίσκεται μόνο στη στοίβα), και η μείωση του δείκτη κορυφής στοίβας όσες θέσεις χρειάζεται, ώστε η στοίβα να επαναφερθεί στην προηγούμενή της κατάσταση, πριν την κλήση της συνάρτησης.
Όσο για την pseudo_tailcall, θα πρέπει να προετοιμαστεί η στοίβα ώστε η κληθείσα συνάρτηση να πιστεύει ότι μόνο αυτή κλήθηκε και όχι εμείς. Πρέπει λοιπόν να βάλουμε όσες παραμέτρους μπορούμε σε καταχωρητές, και να σβήσουμε τα πάντα από τη στοίβα εκτός από τις

παραμέτρους που δε χωράνε σε καταχωρητές και πρέπει να μπουν εκεί, και ρυθμίζουμε ανάλογα την κορυφή της στοίβας. Ακόμα επαναφέρουμε τη διεύθυνση επιστροφής ώστε η συνάρτηση που καλούμε να επιστρέψει απευθείας στη συνάρτηση που μας κάλεσε.
Γενικώς οι διαφορές από arm σε αυτό το σημείο δεν είναι ιδιαίτερες, και το περισσότερο μέρος πραγματοποιείται με αντιγραφή και διόρθωση των σφαλμάτων μεταγλώττισης, και υλοποίηση βοηθητικών συναρτήσεων που λείπουν από τα frame, registers ή hipe_aarch64. Όμως στην περίπτωση που έχουμε χρησιμοποιήσει τον ειδικό καταχωρητή δείκτη στοίβας (SP) της αρχιτεκτονικής για να συγκρατεί την κορυφή της στοίβας μας, θα πρέπει να λάβουμε υπόψη τους αρχιτεκτονικούς περιορισμούς και στην adjust_sp να τον αυξομοιώνουμε κατά μονάδες των 16 bytes ή πολλαπλάσια.
Το δεύτερο και ογκωδέστερο σημείο που προσέχουμε είναι η do_prologue, η οποία παράγει τον κώδικα προετοιμασίας στοίβας στην αρχή της συνάρτησης, αφού εξεταστούν οι εντολές και οι ανάγκες τους. Ομοίως, αντιγράφουμε τη διαδικασία και όποιες απαραίτητες βοηθητικές συναρτήσεις, προσαρμόζοντας στη νέα αρχιτεκτονική τις παραγόμενες εντολές. Πάλι, αν θέλουμε να χρησιμοποιήσουμε τον SP, στο σημείο που συμπεραίνεται το τελικό μέγεθος της στοίβας από τις εντολές, σε περίπτωση που το μέγεθος δεν είναι πολλαπλάσιο των 16 bytes, προσθέτουμε μία πλασματική μεταβλητή στη στοίβα (dummy, padding) ώστε ο δείκτης στοίβας να αυξηθεί σωστά, με τον ίδιο τρόπο που το κάνει και η ensure_minframe, η οποία σε περίπτωση που το μέγεθος στοίβας είναι μικρότερο από όσο μπορεί να απαιτηθεί κατά κάποια κλήση χωρίς επιστροφή για τις παραμέτρους της, προσθέτει πλασματικές θέσεις στη στοίβα ώστε να υπάρχει αρκετός χώρος για την κλήση.

Σε αυτό το σημείο που βάλαμε επιπλέον εγγενείς δομές στην υλοποίηση, θα πρέπει να ενημερώσουμε το hipe_aarhc64_cfg για το αν αποτελούν άλματα, και πως να εξάγει από αυτές τις ετικέτες - στόχους που τα σφάλματα καταλήγουν. Προσθέτουμε λοιπόν και τις #b_fun, #load, #store, #alu, #b_label.
Είναι πιθανόν καθώς το κάνουμε αυτό να μας εμφανιστεί κατά τη μεταγλώττιση ένα παράξενο και ανησυχητικό μήνυμα σφάλματος, που θα δυσκολευτούμε να καταλάβουμε τι ακριβώς μας λέει. Αυτό συμβαίνει διότι η branch_successors δεν καλείται απευθείας από τον κώδικά μας, αλλά αναφορικά, περνώντας ως παράμετρος σε μία συνάρτηση η οποία την καλεί για όλα τα basic blocks του μεταγλωττιζόμενου κώδικα (basic blocks: κομμάτια κώδικα που σε όλες τις περιπτώσεις εκτελούνται γραμμικά. Τα χρησιμοποιεί ο κώδικας ανάθεσης καταχωρητών, καθώς και το frame για την εμβέλεια των μεταβλητών). Έτσι, το μήνυμα σφάλματος δεν θα δείξει τη συνάρτηση που φταίει, αλλά τη συνάρτηση η οποία την πέρασε ως παράμετρο, μπερδεύοντας ίσως τον ανειδίκευτο υλοποιητή. Η λύση όμως είναι πολύ απλή και δεν υπάρχει λόγος ανησυχίας, και το μόνο που έχουμε να κάνουμε είναι να προσέξουμε ότι το μήνυμα σφάλματος, ακριβώς πριν την εκτύπωση της στοίβας (stacktrace), μας δείχνει ένα εντελώς διαφορετικό σημείο στον κώδικα, με όνομα αρχείου και γραμμή. Ακολουθώντας το θα καταλήξουμε στο hipe_aarch64_cfg, στην branch_successors, όπου θα διαπιστώσουμε ότι απλά λείπει μία περίπτωση από το switch.
Ύστερα από μία ή δύο τέτοιου είδους μικροπεριπέτειες, ο κώδικας του frame θα εκτελεστεί επιτυχώς, και η εκτέλεση θα σταματήσει στο βήμα finalize.

5.2.4. Αποσυμπίεση και βελτιστοποιήσεις

Το τελευταίο αρχείο που μένει για να εκτελεστεί η rtl_to_aarch64, είναι το hipe_aarch64_finalise. Σε αυτό, γίνονται δύο λειτουργίες, η αποσυμπίεση σύνθετων εγγενών δομών σε απλούστερες, και η απλούστευση ακολουθιών εντολών από πιο ισχυρότερες που μπορούν να πραγματοποιήσουν την ίδια εργασία, δηλαδή βελτιστοποίηση.
Οι εντολές που πρέπει να αποσυμπιεστούν γενικώς έχουν την ονοματολογία pseudo - ψευδοεντολές, καθώς δεν αντιπροσωπεύουν πραγματικές εντολές της αρχιτεκτονικής, αλλά αντικαθίστανται από ακολουθίες εντολών της, περίπου όπως λειτουργούν οι μακροεντολές σε έναν συμβολομεταφραστή. Στην παρούσα φάση η μόνη ψευδοεντολή που φτάνει μέχρι αυτό το σημείο είναι η pseudo_tailcall_prepare, η οποία είναι εντελώς εικονική, και έτσι κατά την αντικατάσταση απλώς απαλείφεται.
Στο στάδιο της βελτιστοποίησης μπορούμε να είμαστε αρκετά επιφυλακτικοί, και να σβήσουμε οποιεσδήποτε βελτιστοποιήσεις δεν είμαστε σίγουροι ότι είναι συμβατές με τις νέες δομές, καθώς οι βελτιστοποιήσεις δεν είναι απαραίτητες για να παραχθεί με επιτυχία ο κώδικας. Προς το παρόν αντιγράφουμε τις full-word (64-bit) load-after-store, jump-to-next-instruction και move με destination == origin, οι οποίες είναι αρκετά απλές στην αναγνώριση και υλοποίηση, χωρίς ιδιαίτερες διαφορές από arm.
Διορθώνοντας όποια επιπλέον reference errors εμφανιστούν, ο κώδικας εκτελεί πλήρως την rtl_to_aarch64 παράγοντας όλες τις εγγενείς δομές, σταματώντας στο switch της `hipe:assemble`.

5.2.5. Παραγωγή συμβολικής γλώσσας

Το τελευταίο, ίσως, σημαντικό αρχείο που θα χρειαστεί να αντιγράψουμε είναι το hipe_aarch64_assemble. Βάζουμε την κλήση του στο switch του hipe.erl και το μελετούμε ενδελεχώς.
Στην πρώτη φάση, ο συμβολομεταφραστής αναλαμβάνει την τοποθέτηση των σταθερών που χρησιμοποιεί ο κώδικας σε κατάλληλα σημεία, ώστε να βρίσκονται πάντα στην εμβέλεια των εντολών που τις χρησιμοποιούν. Όταν μεταφραστεί αρκετός αριθμός εντολών, τοποθετείται ένα άλμα, το οποίο παρακάμπτει τις σταθερές που ακολουθούν. Πλέον η εμβέλεια των εντολών έχει αλλάξει, οπότε αυξάνουμε τον αριθμό των εντολών που απαιτούνται ώστε να χρειαστεί η παραγωγή του άλματος και η τοποθέτηση των σταθερών. Στη must_flush_pending, θέτουμε το όριο σε 16#FFFF8 bytes, που είναι η εμβέλεια της εντολής φόρτωσης (16#FFFFC) μείον μία εντολή άλματος (4 bytes). Ως βελτιστοποίηση, μπορούμε να συμπεριλάβουμε και τις σταθερές του επόμενου μπλοκ, καθώς η εμβέλεια της εντολής είναι 16#FFFFF προς την αρνητική κατεύθυνση, και αυτό το κάνουμε στην ήδη υπάρχουσα expire_previous. Διορθώνουμε ταυτόχρονα οποιαδήποτε αναφορά σε μέγεθος σταθεράς στη μνήμη σε 8 bytes, και συγκεκριμένα στην insn_size με 4 αν πρόκειται για εντολή, 8 για σταθερά και 0 στις άλλες περιπτώσεις.
Στη δεύτερη φάση, της μετατροπής εγγενών δομών, διορθώνουμε τις περιπτώσεις στην is_not_fallthrough_insn, και μετά ενεργοποιούμε μία μία τις εντολές στην translate_insn που έχουμε χρησιμοποιήσει ή εμφανίζονται ως σφάλματα κατά την εκτέλεση του συμβολομεταφραστή.

Οι πιο απλές στην υλοποίηση είναι οι do_pseudo_blr (ή do_ret) και do_label, οι οποίες απλά παράγουν πλειάδες με ‘ret’ και ‘.label’ opcodes για άμεση κατανάλωση από τον κωδικοποιητή και το 3ο στάδιο του συμβολομεταφραστή αντίστοιχα. Όμοια είναι και η do_b_fun, η οποία όμως παράγει ένα relocation entry ώστε να φορτωθεί η διεύθυνση της κλήσης στο άλμα κατά τη φόρτωση. Επόμενες έρχονται οι do_load, do_store, do_alu και do_mov, που πριν τη γραμμικοποίηση του opcode και των παραμέτρων σε πλειάδα όσο το δυνατόν στη σειρά που τις περιμένει η υλοποίηση της αρχιτεκτονικής, πρέπει να γίνει η γραμμικοποίηση των παρενεργειών (side effects) της δεύτερης παραμέτρου μέσω των do_am1 και do_am2.
Οι παρενέργειες των εντολών δεν χρειάζονται ιδιαίτερη επεξεργασία παρά γραμμικοποίηση, καθώς ο έλεγχός τους και η μετατροπή τους σε μορφή που καταλαβαίνει η αρχιτεκτονική έχει ήδη γίνει στο στάδιο της παραγωγής εγγενών δομών, στην hipe_aarch64: try_aluop_imm και τις συναφείς συναρτήσεις. Όμως είναι απαραίτητη η συνοδεία αυτών των τιμών με κατάλληλα atoms, ώστε να δοθούν υποδείξεις στον κωδικοποιητή και να προληφθούν πιθανά σφάλματα κωδικοποίησης. Έτσι, οι τιμές με συγκεκριμένο αριθμό bit συνοδεύονται π.χ. ως {imm16, <16-bit value>} ώστε ο κωδικοποιητής να ξέρει ότι μπορεί να τις τεμαχίσει ώστε να χωρέσουν σε 16 bit, οι τιμές που αντιστοιχούν σε θέση στη μνήμη με απευθείας τιμή απόστασης ως {immediate_offset, Src, {imm12, Offset}} και μέσω αναφοράς σε καταχωρητή ως {register_offset, Src, do_reg(Reg)}, και οι καταχωρητές μέσω της do_reg ως { r, <register no.> } ώστε να χρησιμοποιήσει τη μορφή εντολής φόρτωσης από καταχωρητή και όχι από σταθερά, και να εμφανιστεί σφάλμα στην περίπτωση χρήσης λάθος μορφής. Φυσικά θα πρέπει ο κωδικοποιητής να αφαιρέσει τον αριθμό του καταχωρητή από το ζεύγος, ώστε να κάνει ταυτόχρονα τον έλεγχο. Ίσως το πιο συχνό σφάλμα κατά την κωδικοποίηση που αντιμετωπίσαμε ήταν η προσπάθεια τοποθέτησης ολόκληρου του ζεύγους {r, R} στα bits της εντολής, όμως θα εμφανίζονταν πολύ πιο επίμονα σφάλματα κωδικοποίησης αν δεν είχαμε αυτή τη μορφή ασφάλειας να μας προλάβει.
Η πιο μεγάλη σε κώδικα υλοποίησης εντολή προς συμβολομετάφραση είναι η pseudo_li, διότι είναι απαραίτητη η αναζήτησή της αναφερόμενης σταθεράς στα τμήματα του κώδικα όπου τοποθετούνται οι εντολές, και η συγκράτηση της θέσης όπου βρίσκεται. Ακόμα, για σύνθετες σταθερές (όπως πλειάδες), δε συμφέρει η τοποθέτησή τους μέσα στον κώδικα, και πρέπει να τοποθετηθεί εντολή στον φορτωτή ώστε να παράξει τη σταθερά στο σωρό κατά την εκτέλεση. Ευτυχώς αυτός ο κώδικας δεν έχει μεγάλη διαφορά ανάμεσα σε αρχιτεκτονικές, και μπορούμε να τον αντιγράψουμε σχετικά αυτούσιο.

Στην τρίτη φάση καλείται ο κωδικοποιητής, και αφού διορθώσουμε κάποια μεγέθη (`<<Value:64/integer-native>>` για σταθερές), το σημείο στο οποίο πρέπει να δώσουμε σημασία είναι η συνάρτηση fix_pc_refs. Εκεί επεξεργάζονται όσες εντολές έχουν αναφορά σε άλλη θέση του προγράμματος (labels), και τοποθετείται η σωστή τιμή απόστασης μέσα τους, τώρα που μετά τη σειριοποίηση η απόσταση είναι γνωστή. Οι δύο εντολές που ενδιαφέρουν είναι το άλμα (b) και η φόρτωση (pseudo_li).
Στην περίπτωση του άλματος, ελέγχουμε τον τύπο του άλματος και βάζουμε έναν έλεγχο γαι ασφάλεια (assertion) αν η απόσταση από την ετικέτα χωράει στον τύπο του άλματος (26bit για απευθείας άλμα και 19bit για άλμα υπό συνθήκη, προσημασμένα). Ομοίως κάνουμε και στη

φόρτωση (19bit μέγεθος προσημασμένης απόστασης για τη ldr), και τοποθετούμε τις πραγματικές τιμές απόστασης στις πλειάδες που προορίζονται για τον κωδικοποιητή.

5.2.6. Κωδικοποίηση

Έχοντας παράξει τις πλειάδες παραμέτρων που αντιστοιχούν σε συμβολικές εντολές με τις παραμέτρους τους, μπορούμε να προχωρήσουμε στην κωδικοποίησή τους σε δυαδικό κώδικα. Με επανειλημμένη εκτέλεση του μεταγλωττιστή στο πρόγραμμα δοκιμής, υλοποιούμε μία - μία τις εντολές που λείπουν στον κωδικοποιητή με τη σειρά, γεμίζοντας την insn_encode/2.
Για την κωδικοποίηση των εντολών είναι απαραίτητη η ανάγνωση του Architecture Reference Manual, το οποίο περιγράφει επακριβώς τις θέσεις και τη μορφή των bits. Είναι μεγάλης βοήθειας να έχουμε τις βοηθητικές μακροεντολές BIT και BF που υπήρχαν στην arm υλοποίηση για τοποθέτηση αριθμητικών τιμών σε συγκεκριμένες θέσεις bit ενός 32-bit bitfield, καθώς και μία αντίστοιχη BFS που κατασκευάζουμε για τοποθέτηση προσημασμένων τιμών. Γενικά, και ως αρχιτεκτονικής RISC, οι κωδικοποιήσεις παρόμοιων εντολών μοιάζουν αρκετά μεταξύ τους στα δυαδικά τους μέρη, και το εγχειρίδιο τις έχει ομαδοποιήσει σε κάποιες βασικές ‘μορφές’. Εκμεταλλευόμαστε λοιπόν αυτή την ευκολία δημιουργώντας ‘backend’ συναρτήσεις που παράγουν τις βασικές μορφές μέσω παραμέτρων, και ‘frontend’ συναρτήσεις που είναι αυτές που καλούνται για να παράξουν τις συνήθεις εντολές, οι οποίες όμως το κάνουν καλώντας τις βασικές μορφές με τις κατάλληλες παραμέτρους που αντιστοιχούν στην εκάστοτε εντολή.

Πρώτα συνθέτουμε την βασική μορφή για αριθμητικές πράξεις με απευθείας όρισμα, data_imm_addsub_form, και με αυτή τις κωδικοποιήσεις των εντολών add και sub, οι οποίες με ένα switch εξετάζουν αν το όρισμα είναι απευθείας τιμή, και καλούν τη βασική μορφή. Συνεχίζουμε με τη βασική μορφή πρόσβασης στη μνήμη με απευθείας όρισμα, ldstr_imm_form, και με την κωδικοποίηση της str (immediate, 64-bit). Μετά, τη βασική μορφή πρόσβασης στη μνήμη με όρισμα απόστασης από τον program counter, ldstr_pcrel_form, η οποία θα παράξει τις φορτώσεις των σταθερών που τοποθετήσαμε κατά την assemble, και τις κωδικοποιήσεις των ldr (immediate, 64-bit) και ldr (pc-relative, 64-bit). Ύστερα τη βασική μορφή για απευθείας άλμα, b_form, και την κωδικοποίηση της b (branch). Τη μορφή για τοποθέτηση δεδομένων σε καταχωρητές, mov_form, και την κωδικοποίηση της mov (immediate). Και τέλος τη βασική μορφή άλματος σε καταχωρητή, b_reg_form, και την κωδικοποίηση της ret.

Υλοποιώντας και το τελευταίο στάδιο της κωδικοποίησης, αν τώρα δοκιμάσουμε μεταγλώττιση, το πρόγραμμα θα μεταγλωττιστεί επιτυχώς.

5.3. Αποσφαλμάτωση. Υλοποίηση κλήσης - επιστροφής

Σε αυτό το σημείο, ανάλογα το πόσο προσεκτικά έχουμε υλοποιήσει το backend μας, το πρόγραμμα που μεταγλωττίστηκε ενδέχεται κατά την εκτέλεσή του να μας παρουσιάσει ποικιλία αποτελεσμάτων. Σε λίγες σπάνιες αλλά αρκετά ευχάριστες περιπτώσεις, θα μας δείξει το σωστό αποτέλεσμα. Σε κάποιες άλλες, κάποιον άλλο αναπάντεχο, και συνήθως δυσανάγνωστο αριθμό. Και στις περισσότερες και χειρότερες περιπτώσεις, ολόκληρο το σύστημα της erlang θα

αποτύχει παταγωδώς, με το λειτουργικό σύστημα να το σταματά αναφέροντας ύπαρξη segmentation fault.
Στις τελευταίες δυσάρεστες περιπτώσεις, πρέπει να ξέρουμε να μην πανικοβαλλόμαστε, καθώς κάθε μία από αυτές έχει τον τρόπο αντιμετώπισής της. Στην περίπτωση παραγωγής εσφαλμένου αποτελέσματος, η γενική μέθοδος είναι η τύπωση, κατά την ώρα της μεταγλώττισης, της συμβολικής γλώσσας που παράγουμε, καθώς και της RTL που μας δίνεται από το προηγούμενο στάδιο, και η εξέταση του αν η παραγόμενη συμβολική γλώσσα συμφωνεί στην εκτέλεσή της με τη δεδομένη RTL. Για την τύπωση της συμβολικής γλώσσας, μπορούμε να υλοποιήσουμε την pp_asm που θα βρίσκεται στο assemble καθώς και στο hipe_aarch64_pp. Διαπιστώσαμε ότι κατά την εξέταση της RTL και της συμβολικής γλώσσας βοήθησε αρκετά η δημιουργία διαγραμμάτων ροής από αυτές σε χαρτί, καθώς είναι πιο εύκολα εκτελέσιμα από άνθρωπο. Ακόμα είναι πολύ βοηθητική η σύγκριση με τη συμβολική γλώσσα που παράγουν άλλα, παρόμοια backends (arm).
Στην περίπτωση σφάλματος κατακερματισμού (segmentation fault), μαζί με την εκτύπωση και ανάγνωση του κώδικα, θα μας βοηθήσει και ο gdb. Βλέποντας το stack trace ή τις τιμές των καταχωρητών, της στοίβας, του σωρού, καθώς και τις κωδικοποιήσεις του κώδικα, μπορούμε να διαπιστώσουμε τι μπορεί να συμβαίνει λάθος. Ακόμα, εκμεταλλευόμενοι αυτή τη δυνατότητα, μπορούμε να εισάγουμε τεχνητά σφάλματα κατακερματισμού πριν το πραγματικό σφάλμα (π.χ. ldr από τη θέση 0x00), ώστε με gdb να διαπιστώσουμε το σημείο από το οποίο αρχίζει η ζημιά, ή αν ο κώδικας κατά την εκτέλεσή του φτάνει σε κάποιο σημείο.
Σε κάθε περίπτωση, θυμόμενοι τις αρχές του test-driven development, αν κάποια προσπάθεια αποσφαλμάτωσης μας πάρει υπερβολικά πολύ χρόνο, είναι σκόπιμο να στρέψουμε την προσπάθειά μας σε υλοποίηση άλλου test case, η οποία είναι πιθανόν, όπως και πολλές φορές μας συνέβη, να δείξει εναλλακτικό τρόπο να ανιχνεύσουμε ή να λύσουμε το πρόβλημα.

Ένα ακόμα εργαλείο αποσφαλμάτωσης που θα βοηθήσει γενικά σε όλες τις περιπτώσεις είναι και η κλήση της erlang:display/1, η οποία μπορεί να μας δείξει αν η εκτέλεση φτάνει σε κάποιο σημείο, και να τυπώσει μία τιμή που χρειαζόμαστε. Για να μπορέσουμε να καλέσουμε όμως την erlang:display, πρέπει να προσθέσουμε στο μεταγλωττιστή μας τις δυνατότητες παραγωγής κώδικα κλήσης - επιστροφής, το οποίο ήταν ανέκαθεν και το επόμενο βήμα μας. Τροποποιούμε λοιπόν το πρόγραμμά μας ως εξής:

```
-module(test).
-export(test/0).
test() ->
  erlang:display(42),
  42.
```

Επειδή η erlang:display είναι built-in function (BIF), πρέπει να σιγουρευτούμε ότι έχουμε ενεργοποιήσει την aarch64 στον emulator, στις μακροεντολές του hipe/hipe_bif0.c. Η νέα εντολή που τώρα παράγεται από την RTL και φτάνει στο hipe_rtl_to_aarch64, είναι η call. Προχωράμε λοιπόν στην ανάγνωση και υλοποίηση της conv_call. Το μεγαλύτερο μέρος του κώδικα που εξετάζουμε αφορά την τακτοποίηση και το διαχωρισμό των παραμέτρων (split_args) σε αυτές που τοποθετούνται σε καταχωρητές, και στις υπόλοιπες οι οποίες πρέπει να τοποθετηθούν στη στοίβα. Με βάση αυτό το διαχωρισμό, είναι πιθανόν να παραχθούν επιπλέον εντολές μετακίνησης (move_actuals) και αποθήκευσης (mk_push_args) πριν την εντολή της κλήσης. Η

ίδια η εγγενής δομή της κλήσης συνοδεύεται από ετικέτες άλματος που ορίζουν που θα επιστρέψει η λειτουργία μετά από την επιτυχή εκτέλεση, καθώς και που θα γίνει άλμα σε περίπτωση σφάλματος εξαίρεσης (exception).
Προσθέτοντας τις νέες εγγενείς δομές στα βοηθητικά αρχεία cfg και defuse που χρησιμοποιούνται από το frame, τοποθετούμε και ορισμούς για τους υπόλοιπους καταχωρητές της αρχιτεκτονικής στο registers, τους οποίους παραλείψαμε στην αρχή για λόγους ασφαλείας και αποσφαλμάτωσης, που τώρα χρειάζεται ο νέος κώδικας για να λειτουργήσει, καθώς κατά την επιστροφή από την κλήση το αποτέλεσμα πιθανόν να πρέπει να τοποθετηθεί σε κάποιο ελεύθερο καταχωρητή πριν χρησιμοποιηθεί. Στο αρχείο cfg, προσθέτουμε νέα περίπτωση στην branch_preds για στατική πρόβλεψη άλματος κατά τις κλήσεις. Ζητάμε από το hipe_bb_weights την πιθανότητα της κλήσης να παράξει σφάλμα εξαίρεσης, και την αναθέτουμε στα άλματα της κλήσης, ώστε κατά την ανάθεση καταχωρητών να δοθεί προτεραιότητα στο basic block του κώδικα εξαίρεσης, αν αυτό είναι πιο πιθανό να εκτελεστεί.
Ομοίως προσθέτουμε και νέες περιπτώσεις στα αρχεία της ανάθεσης καταχωρητών, όπου παρατηρείται αν κάποιες από τις παραμέτρους που δίνονται στην κλήση μπορούν να χρησιμοποιηθούν απευθείας η αν χρειάζεται αντιγραφή των τιμών σε νέους καταχωρητές. Ακόμα, στον κώδικα διαχείρισης στοίβας κατασκευάζουμε τη δομή stack descriptor για το stack unwinding κατά την εξαίρεση, και δεσμεύουμε, αν χρειάζεται, θέσεις στη στοίβα για τις παραμέτρους.
Στη finalize, το συμβολομεταφραστή και τον κωδικοποιητή η νέα εντολή που έχουμε να προσθέσουμε είναι η bl, η κλήση μετ' επιστροφής. Η εντολή σε δυαδικό κώδικα είναι πανομοιότυπη με την απλή b, με ένα μόνο bit να δείχνει ότι πρέπει να αποθηκευτεί η παρούσα διεύθυνση στον καταχωρητή διεύθυνσης επιστροφής, οπότε την προσθέτουμε στην ίδια περίπτωση με την b, όπου η διεύθυνσή της υπολογίζεται μετά τη σειριοποίηση με καταμέτρηση της απόστασης από την ετικέτα αν είναι τοπική κλήση, ή με σχόλιο προς τον φορτωτή αν είναι εξωτερική του module, όπως και ισχύει με την display.

Μετά από αυτά, η κλήση της display πετυχαίνει, και πριν επιστραφεί το αποτέλεσμα της δοκιμαστικής συνάρτησης - ή συμβεί κάποιο σφάλμα κατά την επιστροφή, τυπώνεται στην οθόνη το atom 42.

5.4. Έναρξη δοκιμών

Είναι καιρός να αρχίσουμε την επιδίωξη του βασικού μας στόχου, που είναι η σωστή εκτέλεση των επίσημων δοκιμών του HiPE που βρίσκονται στο lib/hipe/test. Οι δοκιμές χωρίζονται σε μερικές μεγάλες κατηγορίες - "σουίτες", τη βασική (basic suite), συμβολοσειρές δυαδικών τιμών (bitstrings), δομές αντιστοίχισης (maps), και δύο μικρότερες (sanity, opt_verify). Για αρχή δοκιμάζουμε να εκτελέσουμε την πρώτη δοκιμή από τη βασική σουίτα, της βασικής αριθμητικής. Βάζοντας σε σχόλια την εκτέλεση των διαφόρων συναρτήσεων που εκτελούνται μέσα της, μπορούμε να εξετάσουμε ποιες από αυτές λειτουργούν σωστά και ποιες παρουσιάζουν σφάλματα. Είναι καλό να σημειώσουμε σε αυτό το σημείο, όπου έχουμε πολλές συναρτήσεις που μεταγλωτίζονται παράλληλα, ότι για να τυπώσουμε τις εγγενείς δομές που παράγονται για

λόγους αποσφαλμάτωσης, πρέπει να ενεργοποιήσουμε τη σειριακή μεταγλώττιση με την επιλογή no_concurrent_comp.
Το πρώτο σημαντικό σφάλμα που θα μας παρουσιαστεί είναι η έλλειψη εγγενών δομών για τη βασική εντολή που οι δοκιμές χρησιμοποιούν για να διαπιστώσουν αν τα αποτελέσματα κάθε καλούμενης συνάρτησης είναι σωστά, την assertion (<τιμή> = <έκφραση>). Αυτή η εντολή χρειάζεται τη δομή RTL της αριθμητικής πράξης με διακλάδωση (alub), η οποία λειτουργεί σαν κλασική if - εξετάζει την τιμή μιας έκφρασης και πραγματοποιεί ένα άλμα, εν προκειμένω σε κώδικα σφάλματος, αν αποτύχει. Αυτή απαιτεί και την ύπαρξη της δομής απλής αριθμητικής πράξης, alu, αν και μόνο για σύγκριση. Προσθέτουμε λοιπόν στο hipe_rtl_to_aarch64 τα σχετικά της alub, καθώς και της alu, αλλά μόνο για τις περιπτώσεις των switch που παρουσιάζονται σφάλματα. Ακόμα το πρόγραμμα απαιτεί και τις δομές load και store, των οποίων την κωδικοποίηση ήδη έχουμε από τις προηγούμενές μας υλοποιήσεις, οπότε μπορούμε να τις συνδέσουμε με τον υπάρχοντα κώδικα.
Το σημείο που θέλει ίσως περισσότερη προσοχή κατά την υλοποίηση της σύγκρισης και των αριθμητικών πράξεων, είναι κατά τη μετατροπή των άμεσων τιμών σε κωδικοποιημένες, η εκτίμηση ανάλογα την εντολή του κατά πόσο μπορεί να χρησιμοποιηθεί η αρνητική ή η ανεστραμμένη τιμή σε σχέση με την αρχική, και η συμπληρωματική εντολή της αρχικής. Για παράδειγμα, μπορούμε αντί για cmp x0, 5 να χρησιμοποιήσουμε την cmn x0, NOT(5). Με προσεκτική ανάγνωση για τις ιδιαιτερότητες κάθε εντολής στον επίσημο οδηγό της αρχιτεκτονικής, μπορούμε να κατασκευάσουμε κατάλληλες συναρτήσεις για την μετατροπή κάθε συνδυασμού εντολής και άμεσης τιμής, αποφεύγοντας όσο περισσότερο γίνεται την χρήση ξεχωριστού καταχωρητή για την προσωρινή αποθήκευση της άμεσης τιμής.
Ένα ενδιαφέρον τμήμα κώδικα που μπορούμε να προσέξουμε εδώ είναι ο πολλαπλασιασμός με έλεγχο υπερχείλισης. Λόγω νοοτροπίας της RTL και των εξαιρέσεων στην OTP, όλοι οι πολλαπλασιασμοί που γίνονται έχουν έλεγχο υπερχείλισης. Παλιότερα, στην arm, ο πολλαπλασιασμός μεγάλων προσημασμένων τιμών (smull) ήταν μία εντολή που έγραφε ταυτόχρονα σε δύο καταχωρητές, στον πρώτο τα πρώτα 32 bits του αποτελέσματος του πολλαπλασιασμού, και στον δεύτερο τα πιο σημαντικά 32 bits. Ύστερα, το backend παρήγαγε πλήρη ολίσθηση του προσήμου του αποτελέσματος προς τα δεξιά (asr), παράγοντας 11...11 ή 00...00, και σύγκρινε το αποτέλεσμα με τον καταχωρητή υπερχείλισης, για να διαπιστώσει αν υπήρξε υπερχείλιση. Πλέον στην aarch64 δεν υπάρχει καταχωρητής υπερχείλισης, και επειδή ο πολλαπλασιασμός δεν ενημερώνει τις σημαίες, χρησιμοποιούμε ξεχωριστές εντολές mul και smulh, οι οποίες πολλαπλασιάζουν ξεχωριστά για τα χαμηλότερα 64 bits και τα υψηλότερα αντίστοιχα, για να διαπιστώσουμε αν έγινε υπερχείλιση. Η παλιά εντολή smull πλέον χρησιμοποιείται για πολλαπλασιασμό 32-bit αριθμών με χρήση των κάτω μισών των καταχωρητών (w0-w31). Αναρωτηθήκαμε σε αυτό το σημείο αν ήταν πιο αποδοτικό αντί για δεύτερο πολλαπλασιασμό των άνω 64 bit να συνέφερε να κάνουμε έλεγχο για υπερχείλιση με άλλο τρόπο, και εφήυραμε για δοκιμή έναν αλγόριθμο ο οποίος μέσω της εντολής καταμέτρησης bits προσήμου (csb) έβρισκε το μέγιστο μέγεθος σε bits του αποτελέσματος από το άθροισμα των bits των δύο παραμέτρων, και διαπίστωνε υπερχείλιση αν ήταν μεγαλύτερο του 64. Ύστερα, μετά τον πολλαπλασιασμό, μέσω πράξεων xor και bit testing διαπίστωνε αν υπάρχει εσφαλμένη αλλαγή προσήμου, και άρα πάλι υπερχείλιση. Όμως, διαβάζοντας τον οδηγό

βελτιστοποίησης του A-57, αντιληφθήκαμε ότι ένας δεύτερος πολλαπλασιασμός κρατά μόλις 3 κύκλους, και έτσι δε συνέφερε να εισάγουμε τόσες πολλές εντολές απλά για να τον γλιτώσουμε.
Ενημέρωση χρειάζονται ακόμα και τα αρχεία cfg, defuse, regalloc και ra_postconditions, με περιπτώσεις για τις νέες δομές, καθώς και ελάχιστη προσθήκη κώδικα, διότι οι αριθμητικές πράξεις και συγκρίσεις χρησιμοποιούν άμεσες τιμές, οι οποίες επηρεάζουν πιθανόν την ανάθεση καταχωρητών, και οι load και store μάλιστα χρησιμοποιούν άμεσες τιμές τύπου am2 - απόσταση στη μνήμη, η οποία ως διαφορετική δομή χρειάζεται ξεχωριστή συνάρτηση που θα εξάγει συμπεράσματα για τους καταχωρητές οι οποίοι χρησιμοποιούνται από τη δομή. Ακολουθούν ακόμα λίγες προσθήκες στο frame, για να εξυπηρετήσουν τις load και store, οι οποίες είναι πιθανόν να μετακινούν τιμές από και προς τη στοίβα, ανάλογα με τις αποφάσεις του register allocator, και έτσι να χρειάζονται τη θέση των μεταβλητών που πρέπει να προσπελάσουν, μέσα στο frame.
Προσθέτοντας και την περίπτωση του άλματος σε ετικέτα στο συμβολομεταφραστή, την κωδικοποίηση του οποίου έχουμε ήδη από όταν υλοποιήσαμε την κλήση χωρίς επιστροφή, η μεταγλώττιση της εντολής ελέγχου (assertion) πετυχαίνει.

Σε αυτό το σημείο κατά την υλοποίησή μας επιλύσαμε τα πρώτα σφάλματα κωδικοποίησης, παρατηρώντας τη συμπεριφορά του προγράμματος, προσθέτοντας σημεία ελέγχου για να διαπιστώσουμε την εντολή με την εσφαλμένη συμπεριφορά, και μελετώντας την αποκωδικοποίηση του δυαδικού κώδικα που παραγόταν.
΄Υστερα από αυτό, είμαστε έτοιμοι να προχωρήσουμε στην επίλυση του επόμενου σφάλματος μεταγλώττισης, που αφορά την παραγωγή συναρτήσεων με παραμέτρους. Απαιτείται για αυτό, στο hipe_rtl_to_aarch64, πριν τη μετάφραση των εγγενών δομών, να διαχωρίσουμε και να τακτοποιήσουμε τις παραμέτρους της συνάρτησης που παράγουμε, σε παραμέτρους καταχωρητών και στοίβας, ώστε να γίνουν οι απαραίτητες φορτώσεις τιμών σε καταχωρητές όπου χρειάζονται. Και πάλι, έχουμε έτοιμη την split_args και σχετικές συναρτήσεις στη διάθεσή μας, λόγω της εργασίας που έχουμε κάνει ήδη για την κλήση συναρτήσεων.
Για να μεταφραστεί σωστά το υπόλοιπο basic_arith, πρέπει να προσθέσουμε υποστήριξη για την υπόλοιπη γκάμα των αριθμητικών πράξεων που υπάρχουν, στην υλοποίησή μας. Η mk_alu έχει τρεις περιπτώσεις, για αριθμητικές πράξεις μεταξύ καταχωρητών, για πράξεις μεταξύ καταχωρητή και άμεσης τιμής, και σε περίπτωση που έχουμε απενεργοποιήσει τις βελτιστοποιήσεις, μεταξύ άμεσων τιμών, η οποία συνήθως υλοποιείται με αποθήκευση της μίας άμεσης τιμής σε προσωρινό καταχωρητή και ύστερα πραγματοποίηση της πράξης μεταξύ της άλλης άμεσης τιμής και του καταχωρητή. Όπου υπάρχει χρήση άμεσης τιμής, καλούμε τη μετατροπή της σε κωδικοποιημένη μορφή, και τον έλεγχο του αν αυτή γίνεται, μέσω της fix_aluop_imm.
Περιμένουμε τώρα ότι οι νέες αριθμητικές πράξεις που απαιτεί η δοκιμή θα εμφανιστούν στον κωδικοποιητή. Πράγματι, εμφανίζεται ζήτηση και υλοποιούμε τις tst (σύγκριση για ισότητα), mvn (μετακίνηση συμπληρώματος, πιθανότατα από τον κώδικα μετατροπής σταθερών), ldr (register-offset), asr (αριθμητική ολίσθηση προς τα δεξιά), lsl (ολίσθηση αριστερά), orr, and (δυαδικό ή, και), και cmn (σύγκριση συμπληρώματος, πάλι από μετατροπή σταθερών).
Λόγω της χρήσης εντολών με δυαδικές μάσκες στις νέες δοκιμές, όπως η ολίσθηση και οι δυαδικές πράξεις, έχουν πλέον φτάσει στον κωδικοποιητή και οι δομές am1 με bitmask

immediates, άμεσες τιμές κωδικοποιημένες σε δυαδική μάσκα. Εάν έχουμε κάνει σωστά την μετατροπή στη hipe_aarch64: imm_to_am1, μπορούμε τοποθετώντας τις τιμές της μάσκας στις σωστές θέσεις της δυαδικής λέξης των εντολών ubfm και sbfm να τις κάνουμε να δουλέψουν σωστά. Λόγω της αβέβαιης συμπεριφοράς των δυαδικών μασκών σε ακραίες περιπτώσεις, μιας και πρόκειται για νέα τεχνοτροπία, είναι καλό κατά την υλοποίηση των εντολών να πραγματοποιούμε διεξοδικές δοκιμές με διάφορες στοχευμένες τιμές, και να κάνουμε αρκετό σχεδιασμό για τις ιδιαίτερες περιπτώσεις, όπως π.χ. τις τιμές 0 ή (111...111).

Με αυτά, η δοκιμή basic_arith εκτελείται με επιτυχία, και μπορούμε να προχωρήσουμε σε επόμενες δοκιμές της σουίτας.

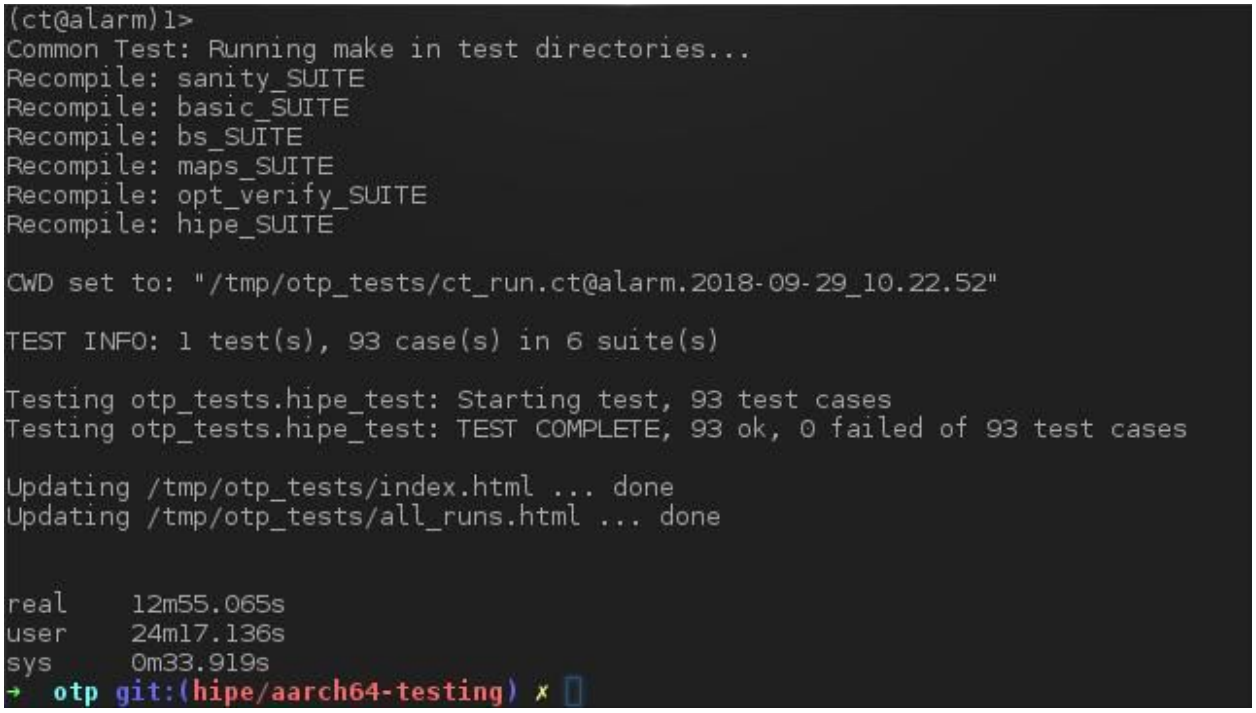
5.5. Ικανοποίηση βασικής σουίτας. Ολοκλήρωση δοκιμών

Προχωρώντας στις επόμενες δοκιμές της σουίτας, ερχόμαστε αντιμέτωποι με νέες προκλήσεις. Ο νέος κώδικας ζητά πρόσβαση στη μνήμη σε θέσεις που δεν είναι στοιχισμένες στα 64 bit, οπότε πρέπει να εξοπλιστούμε με περισσότερες εντολές της αρχιτεκτονικής που είναι ικανές να πραγματοποιήσουν αυτή τη λειτουργία. Έχοντας παράξει τις αντίστοιχες δομές των load και store στον κώδικα διαχείρισης στοίβας, οι οποίες συνοδεύονται από δομές am2 με πιθανόν μη στοιχισμένες αποστάσεις στη μνήμη, μένει να παράξουμε τον δυαδικό κώδικα των εντολών της αρχιτεκτονικής που πραγματοποιούν τη μη στοιχισμένη πρόσβαση στον κωδικοποιητή. Με ένα switch πάνω στον τύπο της δομής am2, παράγουμε την κωδικοποίηση των ldur, stur, ldurh, sturh κ.λ.π. με τη βασική μορφή ldstr_unscaled_form, και τις κατάλληλες δυαδικές παραμέτρους μεγέθους.
Ακόμα μας ζητείται, για τη λειτουργία των δοκιμών χωρίς βελτιστοποίηση, η ύπαρξη του απλούστερου αλγόριθμου ανάθεσης καταχωρητών της γραμμικής σάρωσης (linear_regalloc). Η τοποθέτησή του είναι εύκολη, με αλλαγή των κλήσεων στα αντίστοιχα αρχεία του νέου backend.
Επιπλέον εντολές για τις οποίες ζητείται η κωδικοποίησή τους, καθώς προχωράμε σε αυτή και άλλες σουίτες, είναι το άλμα ή η κλήση σε διεύθυνση σε καταχωρητή (bx / blx), ο πολλαπλασιασμός (smull / smulh), η φόρτωση και αποθήκευση στη μνήμη με απόσταση σε καταχωρητή (ldr / str, register offset), η ολίσθηση με αριθμό θέσεων σε καταχωρητή (lslv / lsrv / asrv), και μερικές ακόμα αριθμητικές - λογικές εντολές την κωδικοποίηση των οποίων δεν είχαμε χρειαστεί ως τώρα (strb, ldrb, exclusive or). Για την υλοποίηση της exclusive or συνοδευόμενης από άλμα χρειάστηκε να χρησιμοποιήσουμε τις νέες εντολές cbz, cbnz (conditional branch), και μία νέα εγγενή δομή, την pseudo_cb, καθώς η eor δεν ενημερώνει τις σημαίες (flags) κατά την εκτέλεση, και ο μόνος αφελής τρόπος που θα είχαμε για να το πετύχουμε αυτό με παραδοσιακό άλμα υπό συνθήκη, θα ήταν η εισαγωγή μιας επιπλέον εντολής σύγκρισης με το μηδέν πριν το άλμα, που θα επιβάρυνε τον κώδικα κατά μία εντολή.
Μετά από μερικές δοκιμές, ζητήθηκε τελικά και η υλοποίηση της δομής switch. Ο τρόπος με τον οποίο λειτουργεί, είναι η τοποθέτηση των ετικετών - στόχων του switch σε ένα διάνυσμα στη μνήμη, από όπου η επίλυση της συνθήκης του switch επιλέγει την 1η, 2η… η διεύθυνση. Η κύρια διαφορά από arm είναι ότι λόγω της προσπελασιμότητας του PC η διεύθυνση - στόχος μπορούσε να φορτωθεί απευθείας από το διάνυσμα στον PC, ενώ τώρα χρειάζεται φόρτωση σε προσωρινό καταχωρητή και ύστερα bx (άλμα σε καταχωρητή). Για να ζητήσουμε νέο

προσωρινό καταχωρητή από το σύστημα, πρέπει η χρήση του να γίνει πριν την ανάθεση καταχωρητών, και έτσι τον δεσμεύουμε με mk_new_temp όσο είμαστε ακόμα στο rtl_to_aarch64, συνοδεύοντας την εγγενή δομή του switch με αυτόν.
Υπήρξαν αρκετά σφάλματα κωδικοποίησης κατά την υλοποίηση όλων αυτών, και σε δύσκολα σημεία με πολλές επαναλήψεις, όπως στη σουίτα δυαδικών συμβολοσειρών (bitstrings), χρειάστηκε να είμαστε εφευρετικοί, κοιτώντας με τύπωση συμβολικής γλώσσας και με τον gdb την τοποθέτηση των συμβολοσειρών στο σωρό, και εισάγοντας κώδικα κατά τη μεταγλώττιση ο οποίος θα παρήγαγε σφάλμα κατακερματισμού μετά από έναν αριθμό εκτελέσεων μιας συγκεκριμένης ετικέτας. Σε αρκετές περιπτώσεις, όταν βλέπαμε ότι η επίλυση ενός σφάλματος έπαιρνε μέρες, για να μην συναντήσουμε τέλμα στην ανάπτυξη, αναγκαστήκαμε να μεταβούμε στην ανάπτυξη άλλων δοκιμών, και τα περισσότερα προβλήματα είτε λύνονταν στην πορεία με επίλυση εναλλακτικών, πιο σύντομων σε δυαδικό κώδικα και άρα ευανάγνωστων δοκιμών, είτε αποδείχτηκε σε μεγάλες δοκιμές που είχαν μη κανονικότητα στην παρουσίαση των σφαλμάτων, ότι ήταν σφάλματα συγχρονισμού της κρυφής μνήμης εντολών και δεδομένων, όπως εξηγήσαμε στο κεφάλαιο 2, και λύσαμε εκ των υστέρων σύμφωνα με τις οδηγίες που δίνει η arm για την διαχείριση της cache σε aarch64.

6. Αποτελέσματα

Έχοντας λύσει και τα τελευταία σφάλματα, επόμενη προσπάθεια μεταγλώττισης και εκτέλεσης των δοκιμών θα μας παρουσιάσει την εξής οθόνη επιτυχίας, που θα επιβεβαιώσει την ολοκλήρωση της εργασίας μας:



Αυτό το μήνυμα σημαίνει ότι ο μεταγλωττιστής μας κατάφερε να μεταγλωττίσει με επιτυχία και τα 93 αρχεία δοκιμών του HiPE με όλες τις συναρτήσεις τους, σε όλα τα επίπεδα βελτιστοποίησης, και αυτά να εκτελεστούν με επιτυχία και να παράγουν το σωστό αποτέλεσμα σε κάθε συνάρτηση, οι οποίες περιελάμβαναν από τις πιο απλές πράξεις μέχρι τις πιο ακραίες περιπτώσεις, καθώς και γνωστές πράξεις που ήταν γνωστό ότι προκάλεσαν σφάλματα στον HiPE στο παρελθόν. Αυτό δε σημαίνει βέβαια ότι έχουμε καλύψει όλα τα προγράμματα που υπάρχουν. Ίσως υπάρχει ακόμα ένα ελάχιστο ποσοστό προγραμμάτων που μπορούν να μας παρουσιάσουν προβλήματα που δεν περιμέναμε, όμως για αυτό το λόγο χρειαζόμαστε τη χρήση του μεταγλωττιστή από το ευρύ κοινό, ώστε να ανιχνεύσουμε αυτές τις περιπτώσεις. Ευτυχώς, λόγω του ότι η υλοποίηση OTP είναι ανοιχτού κώδικα, αυτό είναι υπόθεση λίγου χρόνου ύστερα από την παράδοση της εργασίας.

Παρόλο που ένας λειτουργικός, χωρίς σφάλματα κώδικας είναι βασική προϋπόθεση για την περαιτέρω ανάπτυξη ενός ποιοτικού προϊόντος, θα ήταν ιδανικό αν αυτή η βασική υλοποίηση έχει ήδη απόδοση τουλάχιστον ισάξια με την εικονική μηχανή BEAM, καθώς και συγκρίσιμη απόδοση σε σχέση με τις υλοποιήσεις άλλων αρχιτεκτονικών. Για να το διαπιστώσουμε αυτό, πρέπει να εκτελέσουμε δοκιμές ταχύτητας (benchmarks) στις διάφορες αρχιτεκτονικές, οι οποίες θα μας δώσουν από τους χρόνους εκτέλεσής τους το αποτέλεσμα.

Αναζητώντας αλγορίθμους για να συνθέσουμε τη δοκιμή μας, προμηθευτήκαμε τα προγράμματα από τη σελίδα της benchmarksgame-team[1]. Τροποποιήσαμε ελαφρά τον κώδικα ώστε να αγνοεί το μεγαλύτερο μέρος των εντολών εισόδου και εξόδου, και κατασκευάσαμε ένα πρόγραμμα - οδηγό, το οποίο είναι ικανό να μεταγλωττίσει και να εκτελέσει ύστερα τα προγράμματα προς δοκιμή πολλές φορές σε επανάληψη, με τυχαία σειρά, για να μειωθεί η αβεβαιότητα των μετρήσεων.
Για να γίνουν οι δοκιμές σε διαφορετικά αρχιτεκτονικά backends, χρειαζόμασταν μηχανήματα με κατάλληλες αρχιτεκτονικές. Έχοντας το raspberry pi 3B για την aarch64, εγκαταστήσαμε σε ξεχωριστή μονάδα μνήμης λειτουργικό σύστημα 32 bit, ενεργοποιώντας την λειτουργία AARCH32 του επεξεργαστή, η οποία εκτελεί κώδικα αρχιτεκτονικής arm. Ομοίως, σε μηχάνημα αρχιτεκτονικής x86-64 εγκαταστήσαμε λειτουργικό σύστημα 32 bit, αποκτώντας πρόσβαση και στην x86. Με αυτό τον τρόπο, χρησιμοποιώντας δηλαδή το ίδιο μηχάνημα με διαφορετικό αριθμό bits, μπορούμε να δούμε την καθαρή διαφορά στην απόδοση μεταξύ των αρχιτεκτονικών 32 και 64 bit, ανεξάρτητα από την πρόοδο στο hardware που θα μπορούσε να έχει το μηχάνημα των 64 bit αν χρησιμοποιούσαμε διαφορετικά μηχανήματα.

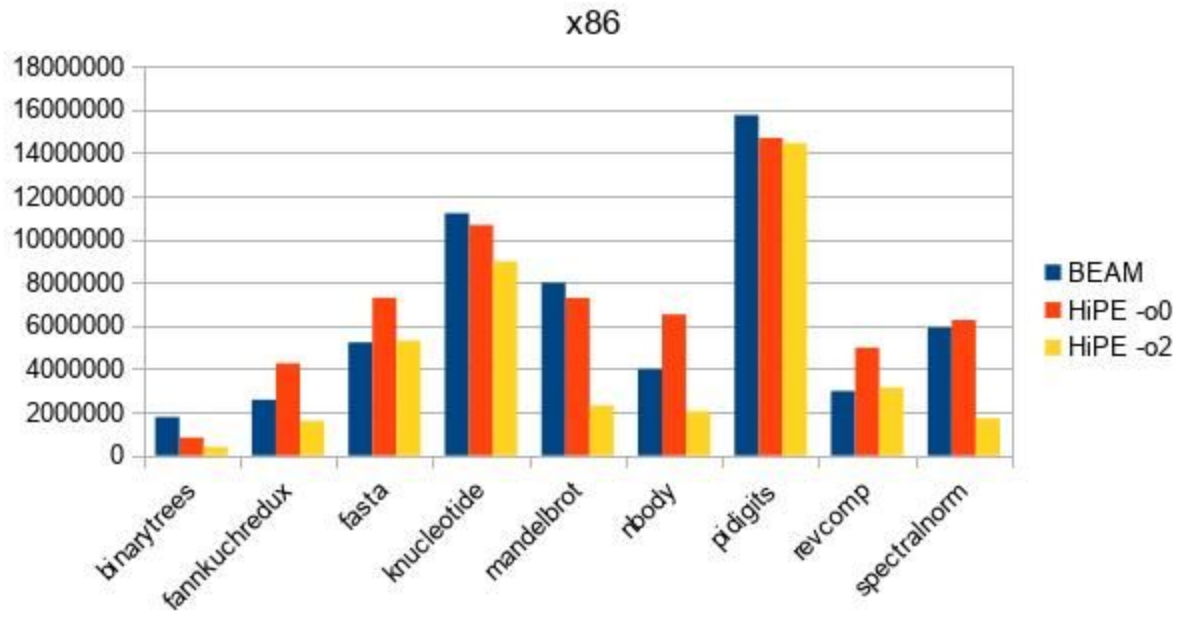
Παρακάτω βλέπουμε τα αποτελέσματα των μετρήσεων στις τέσσερις αρχιτεκτονικές, καθώς και διαγράμματα για οπτική σύγκριση. Για κάθε δοκιμή, μετρήθηκε ο χρόνος εκτέλεσης του προγράμματος προσομοιωμένου με την εικονική μηχανή BEAM, μεταγλωττισμένου με τον HiPE χωρίς βελτιστοποιήσεις (o0), και μεταγλωττισμένου με τον HiPE με συνήθεις βελτιστοποιήσεις ενεργές (o2). Οι μετρήσεις είναι σε μονάδες microseconds.

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/measurements/hipe.html

Επεξεργαστής: Intel Core i5, Haswell
Λειτουργικό σύστημα: Linux Mint 32-bit
Σύστημα: Gigabyte GA-H81M-SP2V, 8GB DDR4 @1600MHz

x86	Binary trees	fannkuch redux	fasta	knucleotide	mandelbrot	nbody	pidigits	revcomp	spectralnorm
BEAM	1773340	2573184	5228185	11202553	7987994	3989224	15739052	298308	5924340
HiPE -o0	827411	4255793	7294424	10660550	7291225	6520301	14690162	498476	6262461
HiPE -o2	392154	1596676	5309497	8976160	2313047	2036760	14447191	313702	1712950

Πίνακας 6.1: Μετρήσεις απόδοσης σε αρχιτεκτονική x86.

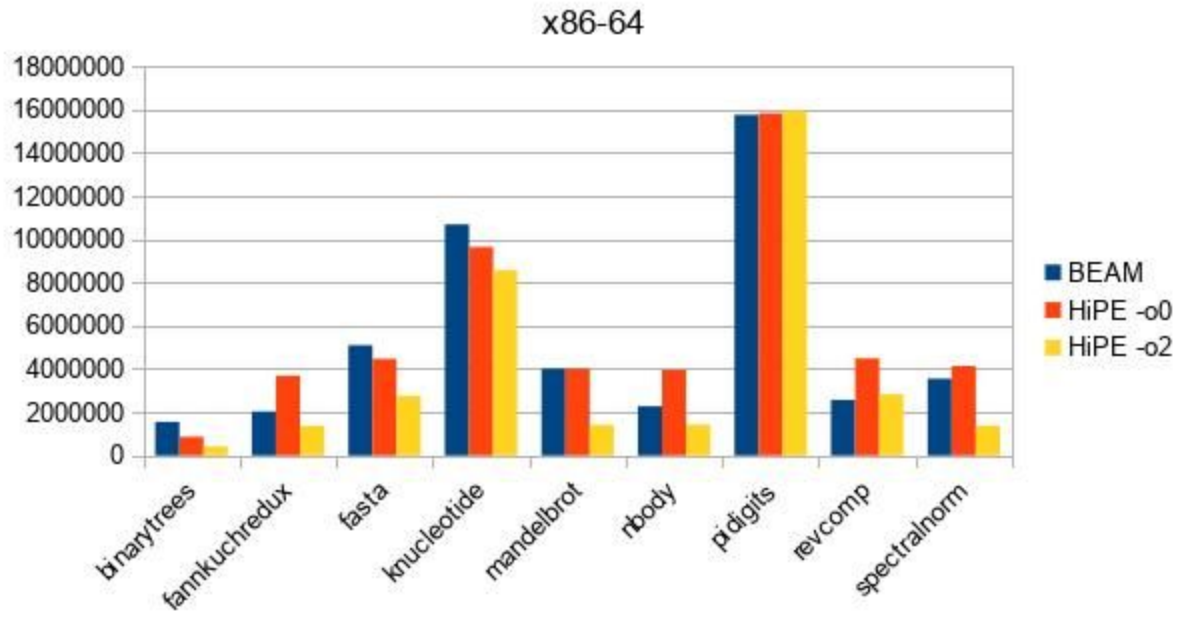


Σχήμα 6.1: Διάγραμμα απόδοσης σε αρχιτεκτονική x86.

Επεξεργαστής: Intel Core i5, Haswell
Λειτουργικό σύστημα: Arch Linux
Σύστημα: Gigabyte GA-H81M-SP2V, 8GB DDR4 @1600MHz

x86-64	binarytrees	fannkuchredux	fasta	knucleotide	mandelbrot	nbody	pidigits	revcomp	spectralnorm
BEAM	1553187	2034445	5100296	10685832	4017200	2270125	15755329	2569158	3548162
HiPE -o0	862891	3685685	4467732	9644081	4014757	3947931	15830835	4487805	4133658
HiPE -o2	415921	1364000	2755161	8566521	1407360	1424842	15948117	2827769	1376475

Πίνακας 6.2: Μετρήσεις απόδοσης σε αρχιτεκτονική x86-64.

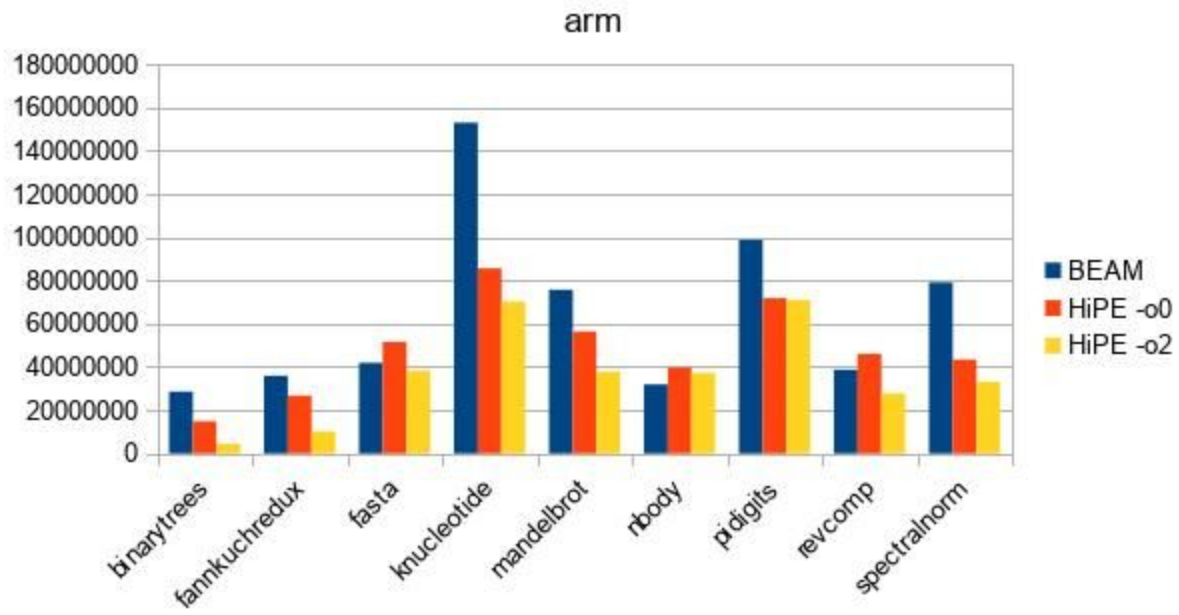


Σχήμα 6.2: Διάγραμμα απόδοσης σε αρχιτεκτονική x86-64.

Επεξεργαστής: Cortex A-53
Λειτουργικό σύστημα: Arch Linux Arm, 32-bit
Σύστημα: Raspberry Pi 3B

ARM	binarytrees	fannkuchredux	fasta	knucleotide	mandelbrot	nbody	pidigits	revcomp	spectralnorm
BEAM	28580455	35907832	41940735	152996857	75735253	31892677	98799884	38670728	79053761
HiPE -o0	14910693	26773516	51573630	85632390	56372812	39677903	71839893	46046538	43449725
HiPE -o2	4386876	10224287	38319309	70276773	37807022	37229305	71050707	27732280	33127988

Πίνακας 6.3: Μετρήσεις απόδοσης σε αρχιτεκτονική arm.

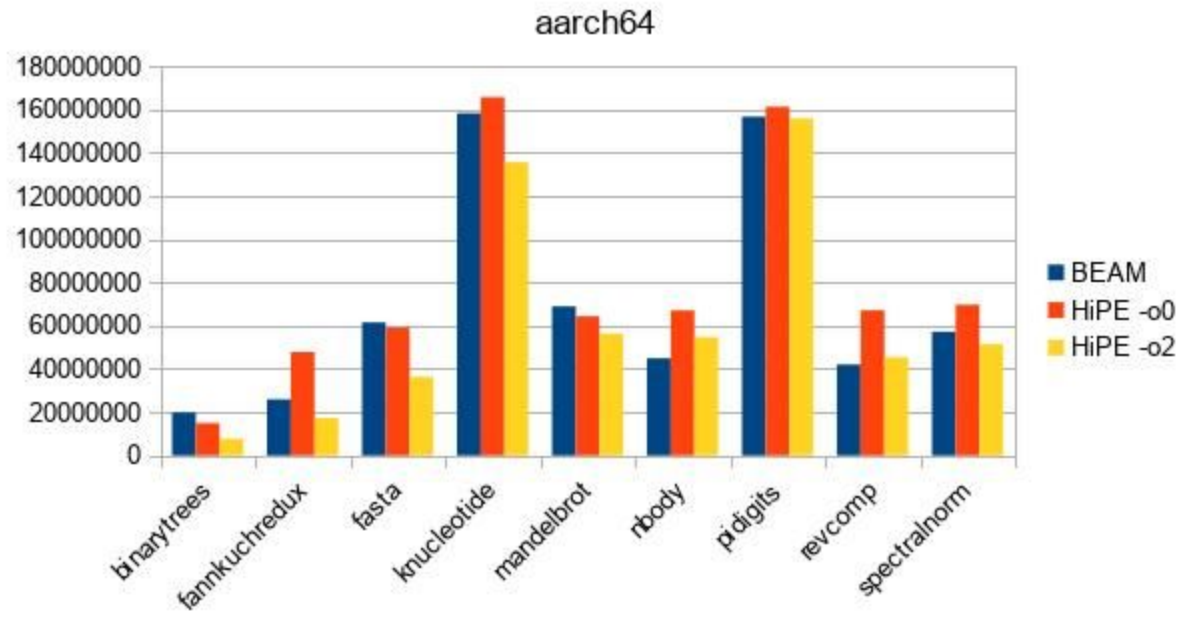


Σχήμα 6.3: Διάγραμμα απόδοσης σε αρχιτεκτονική arm.

Επεξεργαστής: Cortex A-53
Λειτουργικό σύστημα: Arch Linux Arm, 64-bit
Σύστημα: Raspberry Pi 3B

AArch64	Binary trees	fannkuchredux	fasta	knucleotide	mandelbrot	nbody	pidigits	revcomp	spectralnorm
BEAM	19895477	25973618	61497361	158264617	68933461	44942688	156696765	42041404	57217946
HiPE -o0	14980561	47825889	59134175	165744759	64363895	67080548	161322146	67109059	69757281
HiPE -o2	7672729	17227431	36382563	135627716	56303305	54716062	155901947	45411577	51366716

Πίνακας 6.4: Μετρήσεις απόδοσης σε αρχιτεκτονική AArch64.



Σχήμα 6.4: Διάγραμμα απόδοσης σε αρχιτεκτονική AArch64.

Κοιτώντας εποπτικά τα διαγράμματα απόδοσης, διαπιστώνουμε σε γενικές γραμμές ότι η απόδοση του backend που δημιουργήσαμε, αν και στη γενική περίπτωση ισάξια ή και καλύτερη από την απόδοση της εικονικής μηχανής BEAM, υστερεί ελαφρά σε κάποιες δοκιμές σε σχέση με τη διαφορά απόδοσης HiPE - BEAM που έχουν άλλες πλατφόρμες στις ίδιες δοκιμές. Ακόμα, παρόλο που η λογική της υλοποίησής μας είναι όμοια με το προηγούμενο arm backend, βλέπουμε σε αρκετές δοκιμές σχετικά καλύτερη απόδοση της προηγούμενης αρχιτεκτονικής arm όχι μόνο στα μεταγλωττισμένα με HiPE προγράμματα, αλλά και στα προσομοιούμενα μέσω της BEAM. Οφείλουμε λοιπόν μελετώντας τα αποτελέσματα να εξηγήσουμε για ποιους λόγους υπάρχουν αυτές οι διαφορές, αποκτώντας έτσι και έναν μελλοντικό οδηγό για το αν και το πως θα μπορούσαν αυτές να διορθωθούν, και να εξάγουμε χρήσιμα και ενδιαφέροντα συμπεράσματα για τις διαφορές μεταξύ αυτών των τεσσάρων αρχιτεκτονικών.

Για να εξηγήσουμε αρχικά τις διαφορές ανάμεσα στις αρχιτεκτονικές τύπου arm και x86, αφού σκεφτούμε την προφανή διαφορά μεταξύ RISC και CISC, καθώς και το ότι το μηχάνημα δοκιμής της x86 έχει μεγαλύτερης ισχύος επεξεργαστή, κάνοντας την κλίμακα του χρόνου στα διαγράμματα να έχει 10 φορές μικρότερες τιμές, καλό είναι να λάβουμε υπόψη και ότι ενώ οι αρχιτεκτονικές τύπου x86 εκτελούν όλες τις πράξεις κινητής υποδιαστολής στο υλικό, η αρχιτεκτονική arm δεν έχει προβλέψει μονάδα πράξεων κινητής υποδιαστολής, και στην αρχιτεκτονική aarch64 η αντίστοιχη μονάδα δεν χρησιμοποιείται από τον HiPE, διότι το backend μας έχει βασιστεί πάνω στο backend της arm, και οι πράξεις κινητής υποδιαστολής αποτελούν ξεχωριστό σύνολο εντολών. Έτσι, δοκιμές όπως η mandelbrot και nbody δεν έχουν την απόδοση που θα ελπίζαμε. Μάλιστα, επειδή η BEAM στην aarch64 παράγεται από τον gcc, ο οποίος χρησιμοποιεί τη μονάδα πράξεων κινητής υποδιαστολής, βλέπουμε ότι στις ίδιες δοκιμές ο HiPE υστερεί κάπως της BEAM στην υλοποίησή μας. Ιδού λοιπόν ένας πρώτος στόχος για μελλοντική εργασία: η επέκταση της υλοποίησής μας για το νέο σύνολο εντολών της aarch64 για πράξεις κινητής υποδιαστολής.

Εξετάζοντας τις διαφορές ανάμεσα στα αποτελέσματα για arm και aarch64, διαπιστώσαμε ότι δοκιμές με έντονη χρήση κλήσεων συναρτήσεων (k-nucleotide, Pi digits, Fannkuch redux) έχουν αρκετά χειρότερη απόδοση στην υλοποίησή μας. Παρόμοια συμπεριφορά βλέπουμε και σε δοκιμές που χρησιμοποιούν κλήσεις προς το περιβάλλον εκτέλεσης, παρόλο που αυτές οι κλήσεις θεωρούνται ούτως ή άλλως αργές λόγω χρήσης του mode switch - και μιας και δεν υπάρχει μονάδα πράξεων κινητής υποδιαστολής, όλες οι πράξεις κινητής υποδιαστολής αποτελούν και αυτές κλήσεις προς το περιβάλλον εκτέλεσης. Μία όμως σημαντική διαφορά που πρέπει να θυμηθούμε αε αυτό το σημείο για τις δύο αρχιτεκτονικές, είναι το πόσο σημαντικότερος έγινε ο ρόλος του καταχωρητή που συγκρατεί την κορυφή της στοίβας (stack pointer). Ενώ στην arm μπορούσε πρακτικά να είναι οποιοσδήποτε καταχωρητής, στην aarch64 υπάρχει ειδικός καταχωρητής σχεδιασμένος για αυτό το σκοπό, και μάλιστα έχει ειδικές απαιτήσεις κατά την πρόσβασή του. Στα περισσότερα backends του HiPE, αντί για το δείκτη στοίβας του συστήματος, επιλέγεται συνήθως ένας εναλλακτικός για ευελιξία του αλγορίθμου ανάθεσης καταχωρητών, ο λεγόμενος Native Stack Pointer. Διαβάζοντας το αρχείο registers του amd64 backend, θα δούμε ότι εκεί επιλέχθηκε για NSP να είναι ο αυθεντικός δείκτης στοίβας της αρχιτεκτονικής, το οποίο έγινε για λόγους απόδοσης. Είναι λοιπόν λογικό, λαμβάνοντας υπόψη

τους περιορισμούς, να ερευνήσουμε ή να δοκιμάσουμε την απόδοση της υλοποίησης, αν την τροποποιήσουμε ώστε να χρησιμοποιεί τον x31 ως δείκτη στοίβας. Ακόμα, ίσως μία πιο προσεκτική ανάλυση του frame και του mode switch, σημεία ύποπτα στην κλήση συναρτήσεων τοπικών ή και περιβάλλοντος εκτέλεσης, μπορεί να μας δείξει τρόπους βελτιστοποίησης ή ακόμα και αυξομειώσεις στην απόδοση που μπορεί να υπάρχουν λόγω της αύξησης του μεγέθους λέξης κατά δύο φορές κατά τη μετάβαση από 32 στα 64 bit.

Η αύξηση του μήκους λέξης γενικά θα έπρεπε να θεωρείται κάτι θετικό, και είναι, αν κοιτάξουμε την ευεργετική διαφορά στην ταχύτητα που έχουν οι αρχιτεκτονικές x86 και x86-64, όπου στα 64 bit το μεγαλύτερο μήκος λέξης σημαίνει και ταυτόχρονη επεξεργασία περισσότερων δεδομένων σε ορισμένες περιπτώσεις. Όμως, αν δε γίνει προσεκτική εκμετάλλευση, σε άλλες περιπτώσεις σημαίνει ότι για τον ίδιο αριθμό, ενώ πριν χρειάζονταν 32 bits, τώρα χρειάζονται 64. Αν κοιτάξουμε την υλοποίηση του κωδικοποιητή της amd64, θα δούμε αρκετά καλή εργασία σε όλα τα μεγέθη αριθμών, ακόμα και στις πράξεις των 8 bit, το οποίο δείχνει ότι μπορούμε να κερδίσουμε σε απόδοση αν μειώσουμε τον χώρο που χρησιμοποιούμε για τα δεδομένα μας στο ελάχιστο απαραίτητο. Στην περίπτωση των arm αρχιτεκτονικών, η ταχύτητα του HiPE αντί να αυξηθεί, μειώνεται στα 64 bit, κατά ένα σταθερό όρο στις περισσότερες δοκιμές, εκτός ίσως από τις fasta και spectralnorm, οι οποίες εκμεταλλευόμενες τα 64 bit, παρήγαγαν κώδικα καλύτερης απόδοσης. Καλό είναι όμως να λάβουμε υπόψη και τα συστήματα στα οποία οι δοκιμές εκτελούνταν, όπου στις x86 αρχιτεκτονικές είχαμε στη διάθεσή μας μηχάνημα με σύγχρονη, καλής ποιότητας μητρική πλακέτα, ενώ στις arm ο χώρος του συστήματος έχει ελαχιστοποιηθεί στο μέγεθος του raspberry pi, και έτσι οι δίαυλοι επικοινωνίας του συστήματος να μην επιτρέπουν αρκετή κίνηση για να εκμεταλλευτούν τα οφέλη του μεγέθους των 64-bit λέξεων, δημιουργώντας πιθανά ‘μποτιλιαρίσματα’ δεδομένων (bottlenecks). Δοκιμές απόδοσης σε μεγαλύτερα μηχανήματα αρχιτεκτονικής arm μπορούν να μας πείσουν για το αν και κατά πόσο αυτό ισχύει.
Όσον αφορά τη διαφορά απόδοσης BEAM και HiPE στις 32-bit και 64-bit υλοποιήσεις, θα ήταν απαραίτητο να λάβουμε υπόψη και το κατά πόσο η BEAM και ο HiPE χρησιμοποιούν τα νέα σύνολα εντολών (sse*n* κ.α.), και συγκεκριμένα, τις πράξεις τύπου SIMD, οι οποίες είναι πλέον διαθέσιμες στην aarch64, και είναι από προεπιλογή ενεργές στον gcc. Ίσως τα επόμενά μας βήματα να αποσκοπήσουν στη χρήση εντολών πλέον των βασικών που απαιτούνται απλά για την επιτυχή εκτέλεση των παραγόμενων δυαδικών προγραμμάτων.

7. Συμπεράσματα

7.1. Συνεισφορά

Με την ανάθεση αυτού του έργου ως διπλωματική εργασία, αποκομίσαμε γνώσεις μεγάλης εκπαιδευτικής αξίας, καθώς μελετήθηκαν από πρώτο χέρι οι εσωτερικές υλοποιήσεις της θεωρίας των μεταγλωττιστών που διδάσκονται στα πανεπιστημιακά μαθήματα. Παράλληλα, αυτή η διπλωματική εργασία, με τη σειρά της, αποτελεί μία επαρκή πηγή γνώσεων για τις διαφορές ανάμεσα στις αρχιτεκτονικές arm και aarch64, ένα αντικείμενο το οποίο προγραμματιστές συστημάτων αναζητούν συχνά για την εργασία τους, και για το οποίο οι γνώσεις που συναντώνται στο διαδίκτυο, με εξαίρεση τα έγγραφα της arm, είναι κατά το πλείστον αποσπασματικές. Ακολουθώντας τα βήματά μας, υλοποιητές της aarch64, προγραμματιστές που συνεισφέρουν στην υλοποίηση OTP, αλλά και γενικότερα προγραμματιστές που αναπτύσσουν μεταγλωττιστές, θα έχουν έναν χρήσιμο οδηγό για την εργασία τους.
Με αυτή την εργασία, ολοκληρώθηκε ένα έργο μεγάλης χρησιμότητας, καθώς με την είσοδο της νέας αρχιτεκτονικής η ανάγκη επέκτασης υφιστάμενων μεταγλωττιστών όπως ο HiPE γίνεται όλο και πιο επιτακτική, λόγω του σταδιακά αυξανόμενου όγκου των εφαρμογών που εξαρτώνται από αυτούς. Εφαρμογές σε erlang μπορούν τώρα να μεταγλωτίζονται στα μηχανήματα aarch64 σε γλώσσα μηχανής, δυνατότητα η έλλειψη της οποίας προηγουμένως ίσως τις αποθάρρυνε να πραγματοποιήσουν τη μετάβαση στα νέα μηχανήματα 64-bit. Η υλοποίησή μας αποτελεί μία ισχυρή βάση, ώστε με την περεταίρω βελτιστοποίησή της και εκμετάλλευση των δυνατοτήτων των 64 bit, που θα αυξήσει την απόδοση των μεταγλωττιζόμενων προγραμμάτων, θα κάνει τις εφαρμογές που πραγματοποιούν τελικά τη μετάβαση να δουλεύουν ταχύτερα και με λιγότερη κατανάλωση ενέργειας. Σε συνδυασμό με την ευέλικτη δικτυακή λογική της erlang, αρκετά είδη εφαρμογών μπορούν να επωφεληθούν από αυτό, όπως κυψέλες και διαμοιραστές τηλεπικοινωνιών, συστήματα παρακολούθησης, καταγραφής μετρήσεων και απομακρυσμένου ελέγχου, συνεργατικά δίκτυα, και μάλιστα ως συνέπεια του τελευταίου, οι εφαρμογές που βασίζονται στη νέα νοοτροπία διαδικτύου των πραγμάτων (IoT), καθώς αρχές της είναι η ανταλλαγή μηνυμάτων, βασική δυνατότητα της erlang, και η χαμηλή κατανάλωση, χαρακτηριστικό των αρχιτεκτονικών RISC. Μεγάλες εταιρίες χρησιμοποιούν την erlang ήδη για τις εφαρμογές τους, και με την aarch64, η ιδέα της κατασκευής ενός ‘πράσινου’ κέντρου δεδομένων (data center) χαμηλής κατανάλωσης δεν είναι απόμακρο όνειρο.

7.2. Μελλοντική έρευνα

Έχοντας ολοκληρώσει την εργασία μας, ανασκοπώντας όλη τη διάρκεια της ανάπτυξης, αλλά και έχοντας παρατηρήσει τα αποτελέσματα των μετρήσεων απόδοσης, μπορούμε να βγάλουμε συμπεράσματα που θα υποδείξουν τις επόμενες κινήσεις έρευνας με βάση την παρούσα.

Καταρχάς, τα αποτελέσματα δείχνουν ότι παρά το σύγχρονο της τεχνολογίας των 64 bit, αν δεν επενδύσουμε στα νέα σύνολα εντολών, και αν δε χρησιμοποιήσουμε όλες τις νέες δυνατότητες

που μας προσφέρει η αρχιτεκτονική, δε θα μπορέσουμε να εκμεταλλευτούμε την πλήρη δύναμή της, και μάλιστα το μέγεθος της λέξης θα είναι μάλλον εμπόδιο στον αγώνα κατάκτησης της απόδοσης. Οπότε, μελλοντικές εργασίες περιλαμβάνουν την επέκταση του παρόντος backend, πρώτα για το σύνολο εντολών κινητής υποδιαστολής (FP), και ύστερα για τις εντολές διανυσμάτων (SIMD). Ακόμα, μπορεί να γίνει αρκετό έργο στις βελτιστοποιήσεις (peep - hole και άλλες optimizations), για την εκμετάλλευση των αχρησιμοποίητων εντολών της aarch64 σε μεταγλωττιστή και φορτωτή, καθώς και του προκαθορισμένου δείκτη στοίβας (SP).

Θα ήταν ενδιαφέρον αν είχαμε δυνατότητα να εντοπίσουμε συνθήκες όπου η εκτέλεση ενός προγράμματος είναι πάντοτε ταχύτερη και ταυτόχρονα επιτρεπτή σε 32 bit. Θα μπορούσε να γίνει αναζήτηση του κατά πόσο είναι εύκολο και γρήγορο, κατά τον εντοπισμό μιας τέτοιας περίπτωσης, ίσως μέσω just-in-time compilation (βλ. HiPErJIT), να ενεργοποιείται ή να ζητείται από το σύστημα, πιθανόν μέσω ενός kernel module, η ενεργοποίηση του AARCH32 mode του επεξεργαστή, το οποίο θα εκτελέσει την εν λόγω περίπτωση προγράμματος σε σύνολο εντολών αρχιτεκτονικής arm.

Μιας και η aarch64 πλέον εισάγεται στα κινητά με σταθερό ρυθμό, μία εργασία με αρκετό μέλλον θα ήταν η δημιουργία ή υλοποίηση συνεργατικού δικτύου wi-fi μέσω κινητών, σε erlang, και η μέτρηση της απόδοσής του και της κατανάλωσής του. Η τεχνική δυσκολία σε αυτό το σημείο θα ήταν το να προσαρμοστεί η υλοποίηση OTP ώστε να μπορεί να εγκατασταθεί σε λογισμικό Android, όμως με το ρυθμό που τα κινητά τηλέφωνα κατακτούν την αγορά, ίσως αυτό να παρουσιαστεί ούτως ή άλλως ως ανάγκη στο κοντινό μέλλον.

8. Βιβλιογραφία

ARM Information Center, *What is the difference between a von Neumann architecture and a Harvard architecture?* infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3839.html

ARM Information Center, *Procedure Call Standard for aarch64*. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055c/IHI0055C_beta_aapcs64.pdf

ARM Information Center, *Cortex A57 Software Optimization Guide*. http://infocenter.arm.com/help/topic/com.arm.doc.uan0015b/Cortex_A57_Software_Optimization_Guide_external.pdf

"ARM Discloses Technical Details Of The Next Version Of The ARM Architecture." *Arm | The Architecture for the Digital World*, www.arm.com/about/newsroom/arm-discloses-technical-details-of-the-next-version-of-the-arm-architecture.php.

Arm Ltd. "ARM Architecture Reference Manual ARMv8, for ARMv8-A Architecture Profile | ARM Architecture Reference Manual ARMv8, for ARMv8-A Architecture Profile – Arm Developer." *ARM Developer*, developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile.

Beck, Kent. *Test-Driven Development by Example*. Addison-Wesley, 2014.

Fitzpatrick, Jason. "An Interview with Steve Furber." *ACM*, 1 May 2011, cacm.acm.org/magazines/2011/5/107684-an-interview-with-steve-furber/fulltext.

Hachman, Mark. "AMD Tops Intel with Its 32-Core Threadripper 2, Which Will Ship This Year."

PCWorld, PCWorld, 5 June 2018, www.pcworld.com/article/3279264/components-processors/amd-tops-intel-with-32-core-threadripper-2-computex.html.

"The High-Performance Erlang Project." *Department of Information Technology - Uppsala University*, Department of Information Technology, Uppsala University, Sweden, www.it.uu.se/research/group/hipe/.

Hudak, Paul. "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys*, vol. 21, no. 3, Jan. 1989, pp. 359–411., doi:10.1145/72551.72554.

Hussung, Tricia. "What Is the Software Development Cycle?" *Husson University*, 6 Apr. 2018, online.husson.edu/software-development-cycle/.

"Introduction to RISC Technology." *RISC Pros and Cons*, www.inf.fh-dortmund.de/personen/professoren/swik/risc/intro_to_risc/irt0_index.html.

"The Long Road to 64 Bits - ACM Queue." *Research for Practice: Cryptocurrencies, Blockchains, and Smart Contracts; Hardware for Deep Learning - ACM Queue*, queue.acm.org/detail.cfm?id=1165766.

Pettersson, Mikael, et al. "The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation." *Functional and Logic Programming Lecture Notes in Computer Science*, 2002, pp. 228–244., doi:10.1007/3-540-45788-7_14.

"Raspberry Pi 3." *Raspberry Pi 3 | Arch Linux ARM*, archlinuxarm.org/platforms/armv8/broadcom/raspberry-pi-3.

Sagonas, K., et al. "All You Wanted to Know about the HiPE Compiler." *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang - ERLANG '03*, 2003, doi:10.1145/940880.940886.

Vajda, Andras. *Programming Many-Core Chips*. Springer, 2014.