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

HiPErJiT: Ένας Just-in-Time μεταγλωττιστής για την Erlang

Διπλωματική Εργασία

ΚΩΝΣΤΑΝΤΙΝΟΣ ΚΑΛΛΑΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

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

HiPErJiT: Ένας Just-in-Time μεταγλωττιστής για την Erlang

Διπλωματική Εργασία

ΚΩΝΣΤΑΝΤΙΝΟΣ ΚΑΛΛΑΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 6η Ιουλίου 2018.

.....	.....	.....
Κωνσταντίνος Σαγώνας Αναπληρωτής Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Αναπληρωτής Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2018

Κωνσταντίνος Καλλάς

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

Περίληψη

Παρουσιάζουμε το HiPErJiT, ένα Just-in-Time μεταγλωττιστή για τη γλώσσα προγραμματισμού Erlang που χρησιμοποιεί δεδομένα καταγραφής και βασίζεται στο HiPE, το μεταγλωττιστή πηγαίου κώδικα της Erlang. Το HiPErJiT χρησιμοποιεί δεδομένα καταγραφής χρόνου εκτέλεσης για να αποφασίσει ποιές ενότητες κώδικα να μεταγλωττίσει σε πηγαίο κώδικα, ποίες συναρτήσεις να εξειδικεύσει με δυναμικές πληροφορίες τύπων και ποιές κλήσεις συναρτήσεων να ενσωματώσει. Το HiPErJiT είναι ενσωματωμένο στο σύστημα χρόνου εκτέλεσης της Erlang και υποστηρίζει τα χαρακτηριστικά της γλώσσας που είναι απαραίτητα για τις εφαρμογές της, όπως η φόρτωση καυτού κώδικα. Παρουσιάζουμε την αρχιτεκτονική του HiPErJiT, περιγράφουμε τις βελτιστοποιήσεις που εκτελεί και συγκρίνουμε την επίδοση του σε σχέση με το BEAM, το HiPE και το Pyrlang. Το HiPErJiT διπλασιάζει την ταχύτητα εκτέλεσης διάφορων προγραμμάτων σε σχέση με το BEAM και προσφέρει επιδόσεις παρόμοιες με το HiPE, παρά τα κόστη καταγραφής και μεταγλώττισης που έχει σε αντίθεση με ένα Ahead-of-Time μεταγλωττιστή.

Λέξεις κλειδιά

JiT μεταγλωττιστής, HiPE, Erlang, καταγραφή χρόνου εκτέλεσης, εξειδίκευση τύπων, βελτιστοποιήσεις δεδομένων χρόνου εκτέλεσης

Abstract

We introduce HiPErJiT, a profile-driven Just-in-Time compiler for the BEAM ecosystem based on HiPE, the High Performance Erlang compiler. HiPErJiT uses runtime profiling to decide which modules to compile to native code and which of their functions to inline and type-specialize. HiPErJiT is integrated with the runtime system of Erlang/OTP and preserves aspects of Erlang's compilation which are crucial for its applications: most notably, tail-call optimization and hot code loading at the module level. We present HiPErJiT's architecture, describe the optimizations that it performs, and compare its performance with BEAM, HiPE, and Pyrlang. HiPErJiT offers performance which is about two times faster than BEAM and almost as fast as HiPE, despite the profiling and compilation overhead that it has to pay compared to an ahead-of-time native code compiler. But there also exist programs for which HiPErJiT's profile-driven optimizations allow it to surpass HiPE's performance.

Key words

JiT compiler, HiPE, Erlang, profiling, type specialization, profile-driven optimizations

Ευχαριστίες

Καταρχάς θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Κωστή Σαγώνα, αφενός μεν για την πολύτιμη καθοδήγηση του καθόλη τη διάρκεια της διπλωματικής εργασίας μου, αφετέρου δε για τη διάθεση του για συζήτηση κάθε φορά που είχα κάποια απορία. Ακόμη, θα ήθελα να εκφράσω την εκτίμηση μου για το Νίκο Παπασπύρου, γιατί τα μαθήματα που διδάσκει μαζί με τον Κωστή ήταν η πρώτη και πιο καθοριστική επαφή μου με τις γλώσσες προγραμματισμού.

Θα ήθελα επίσης να ευχαριστήσω τους συμφοιτητές και φίλους μου για όλες τις ενδιαφέρουσες συζητησεις μας, που με έβαλαν πολλές φορές σε σκέψεις, με διαμόρφωσαν, και συνέβαλαν στις πιο σημαντικές αποφάσεις που έχω πάρει στη ζωή μου. Εκτός αυτού, χρωμάτισαν τη φοιτητική μου ζωή με τις πιο ωραίες εμπειρίες και αναμνήσεις.

Τέλος, θέλω να πω το πιο μεγάλο ευχαριστώ στους γονείς μου, Γιάννη και Κατερίνα, και στον αδερφό μου Νίκο, που ήταν πάντα δίπλα μου, στα εύκολα και στα δύσκολα, μου έδειξαν αστείρευτη εμπιστοσύνη, και στήριξαν όλες μου τις επιλογές σαν να ήταν δικές τους.

Κωνσταντίνος Καλλάς,

Αθήνα, 6η Ιουλίου 2018

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-4-18, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιούλιος 2018.

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

Περιεχόμενα

Κατάλογος σχημάτων

Κώδικες

Κεφάλαιο 1

Εισαγωγή

Η Erlang είναι μια συναρτησιακή γλώσσα προγραμματισμού με χαρακτηριστικά που υποστηρίζουν την ανάπτυξη κατανεμημένων εφαρμογών και συστημάτων με απαιτήσεις για υψηλή διαθεσιμότητα και ανταπόκριση. Η κύρια υλοποίηση της, το σύστημα Erlang/OTP, συνοδεύεται από έναν μεταγλωττιστή bytecode για την εικονική της μηχανή, που ονομάζεται BEAM, η οποία παράγει φορητό και σχετικά αποδοτικό κώδικα. Για εφαρμογές με απαιτήσεις για καλύτερη απόδοση, μπορεί να επιλεγεί ο μεταγλωττιστής κώδικα μηχανής που ονομάζεται HiPE (High Performance Erlang). Για την ακρίβεια, bytecode και κώδικας μηχανής μπορούν να συνυπάρχουν χωρίς προβλήματα στο σύστημα χρόνου εκτέλεσης της Erlang.

Παρά την ευελιξία αυτή, η επιλογή τμημάτων μιας εφαρμογής προς μεταγλώττιση σε κώδικα μηχανής είναι προς το παρόν μη αυτόματη. Ίσως θα ήταν καλό εάν το ίδιο το σύστημα μπορούσε να αποφασίσει ποια τμήματα να μεταγλωττίσει σε κώδικα μηχανής με just-in-time τρόπο. Επιπλέον, θα ήταν καλύτερο αν η διαδικασία αυτή καθοδηγούταν από πληροφορίες που συλλέχθηκαν κατά τη διάρκεια του χρόνου εκτέλεσης και. ακόμα καλύτερο, αν η διαδικασία ήταν αρκετά έξυπνη ώστε να επιτρέπει τη συνεχή βελτιστοποίηση του κώδικα μιας εφαρμογής.

1.1 Στόχοι Διπλωματικής

Σε αυτή τη διπλωματική εργασία θα περιγράψουμε το σχεδιασμό και την υλοποίηση του HiPErJiT, ενός Just-in-Time μεταγλωττιστή για την Erlang με βάση το HiPE. Πιστεύουμε ότι αποτελεί ένα πρώτο βήμα προς ένα μεταγλωττιστή συνεχούς βελτιστοποίησης. Κύριοι στόχοι μας είναι το HiPErJiT να:

- Επιτυγχάνει καλύτερη απόδοση σε σχέση με άλλους μεταγλωττιστές της Erlang.
- Διατηρεί όλα τα σημαντικά χαρακτηριστικά του Erlang.
- Είναι εύκολο να διατηρηθεί συμβατό και σε αρμονία με τα υπόλοιπα τμήματα του Erlang/OTP.

Το HiPErJiT χρησιμοποιεί την υποστήριξη καταγραφής δεδομένων του συστήματος χρόνου εκτέλεσης της Erlang, για να καταγράψει δεδομένα από την εκτέλεση του bytecode και να επιλέξει ποια τμήματα του προγράμματος θα μεταγλωττίσει σε κώδικα μηχανής. Για τα επιλεγμένα τμήματα του προγράμματος, αποφασίζει επιπρόσθετα ποιες συναρτήσεις να ενσωματώσει και ποιές να βελτιστοποιήσει με βάση τις πληροφορίες τύπων που έχουν καταγραφεί κατά την εκτέλεση του προγράμματος. Θεωρούμε ότι η λίστα των πρόσθετων βελτιστοποιήσεων που θα υλοποιηθούν με βάση τις πληροφορίες καταγραφής θα επεκταθεί στο μέλλον, ειδικά αν ο HiPErJiT γίνει ένας JiT μεταγλωττιστής που εκτελεί συνεχή βελτιστοποίηση προγραμμάτων με χρήση ανατροφοδότησης. Επί του παρόντος, η μεταγλώττιση ενεργοποιείται μόνο μία φορά για κάθε φορτωμένο τμήμα του προγράμματος και η καταγραφή διακόπτεται σε εκείνο το σημείο.

1.2 Επισκόπηση Κεφαλαίων

Η διπλωματική είναι οργανωμένη στα ακόλουθα κεφάλαια.

- Το Κεφάλαιο 2 περιλαμβάνει πληροφορίες σχετικά με τη γλώσσα Erlang, το Erlang Run-Time System, τους διάφορους μεταγλωττιστές της Erlang και πληροφορίες σχετικά με τη JiT μεταγλώττιση.
- Το Κεφάλαιο 3 παρουσιάζει την αρχιτεκτονική του HiPErJiT και το σκεπτικό πίσω από κάποιες από τις σχεδιαστικές αποφάσεις που πήραμε.
- Το Κεφάλαιο 4 περιγράφει τις βελτιστοποιήσεις που βασίζονται στην καταγραφή δεδομένων εκτέλεσης, που εκτελεί ο HiPErJiT.
- Το Κεφάλαιο 5 παρουσιάζει μια αξιολόγηση της απόδοσης του HiPErJiT σε σύγκριση με άλλους μεταγλωττιστές της Erlang.
- Το Κεφάλαιο 6 περιγράφει την τρέχουσα κατάσταση του HiPErJiT και προτείνει πιθανές μελλοντικές βελτιώσεις του.

Κεφάλαιο 2

Υπόβαθρο

2.1 Erlang, ERTS και HiPE

Πριν περιγράψουμε το έργο μας, είναι σημαντικό ο αναγνώστης να κατανοήσει τα βασικά στοιχεία της Erlang, το σύστημα χρόνου εκτέλεσης της και τον καθιερωμένο μεταγλωττιστή της HiPE.

2.1.1 Erlang

Η Erlang είναι μια δυναμική, αυστηρή, συναρτησιακή γλώσσα, ειδικευμένη στις κατανεμημένες εφαρμογές. Είναι σχεδιασμένη για εφαρμογές σε τηλεπικοινωνιακά συστήματα. Εξαιτίας αυτού, σχεδιάστηκε για να ικανοποιεί έξι βασικές απαιτήσεις [Arms03].

Ταυτοχρονισμός. Πολύ μικρές επιβαρύνσεις για τη δημιουργία, καταστροφή και διατήρηση μεγάλου αριθμού ταυτόχρονων διεργασιών.

Ενθυλάκωση Σφαλμάτων. Τα σφάλματα που εμφανίζονται σε μια διαδικασία δεν πρέπει ποτέ να βλάψουν άλλες διεργασίες του συστήματος.

Ανίχνευση Σφαλμάτων. Πρέπει να είναι δυνατή η ανίχνευση σφαλμάτων τόσο τοπικά (στην ίδια διεργασία) όσο και απομακρυσμένα (σε διαφορετική διεργασία).

Αναγνώριση Σφαλμάτων. Πρέπει να είναι δυνατή η αναγνώριση της αιτίας ενός σφάλματος, ώστε να είναι δυνατή η λήψη διορθωτικών μέτρων.

Αναβάθμιση Κώδικα. Θα πρέπει να είναι δυνατή η αλλαγή του κώδικα καθώς εκτελείται, χωρίς διακοπή ή επανεκκίνηση του συστήματος.

Σταθερή Αποθήκευση. Θα πρέπει να είναι δυνατή η αποθήκευση δεδομένων κατά τρόπο που να είναι ανθεκτικός σε περίπτωση που το σύστημα πέσει.

Για την Erlang όλα είναι *διεργασίες* και κάθε *διεργασία* είναι απομονωμένη, δεν μοιράζεται πόρους και αλληλεπιδρά με το περιβάλλον της μόνο μέσω της μετάδοσης μηνυμάτων. Δεδομένου ότι δεν υπάρχουν διαμοιραζόμενες δομές δεδομένων[1] και άλλες συμβατικές μορφές συγχρονισμού στην Erlang, είναι εύκολο να καταλάβουμε πώς λειτουργεί η γλώσσα αφού κατανοήσουμε το σειριακό υποσύνολο της. Αυτό το υποσύνολο είναι μια δυναμική, αυστηρή, συναρτησιακή γλώσσα προγραμματισμού, η οποία σε μεγάλο βαθμό δεν περιέχει παρενέργειες.

[1] Τεχνικά υπάρχουν και άλλοι τρόποι για επικοινωνία, όπως το Erlang Term Storage (ETS), το οποίο επιτρέπει διαμοιραζόμενη πρόσβαση σε δεδομένα από όλες τις διεργασίες ενός συστήματος.

Σειριακή Erlang

Η Erlang έχει οκτώ βασικούς τύπους δεδομένων.

- Integers, που είναι ακέραιοι απεριόριστης ακρίβειας. Παραδείγματα: `42, 2#101, 16#1f`.
- Atoms, που χρησιμοποιούνται για να υποδηλώνουν διακριτές τιμές. Πρόκειται για ακολουθίες αλφαριθμητικών χαρακτήρων που αρχίζουν με ένα μικρό γράμμα. Σε περίπτωση που περιέχουν διαστήματα ή κεφαλαία γράμματα, πρέπει να περικλείονται σε απλά εισαγωγικά. Παραδείγματα: `hello, phone_number, 'Monday'`.
- Floats, δεκαδικοί αριθμοί, που αντιπροσωπεύονται ως αριθμοί κινητής υποδιαστολής IEEE 754 64 bit [IEEE08]. Παραδείγματα: `2.3, 2.3e3, 2.3e−3`.
- Οι αναφορές (references) είναι μοναδικά σύμβολα των οποίων η μόνη ιδιότητα είναι ότι μπορούν να συγκριθούν για ισότητα. Δημιουργούνται με την κλήση της συνάρτησης `make_ref/0`.
- Bitstrings, δυαδικές ακολουθίες, που χρησιμοποιούνται για την αποτελεσματική αποθήκευση δυαδικών δεδομένων. Παραδείγματα: `<<10,20>>, <<"ABC">>, <<1:1,0:1>>`.
- Pids, που είναι αναγνωριστικά διεργασίας, τα οποία χρησιμοποιούνται για την αναφορά διεργασιών. Παράδειγμα: `<0.51.0>`.
- Port Identifiers, που είναι παρόμοια με τα αναγνωριστικά διεργασιών, αλλά χρησιμοποιούνται για τον εντοπισμό θυρών, οι οποίες αποτελούν τον βασικό μηχανισμό επικοινωνίας με τον εξωτερικό κόσμο. Δημιουργούνται με την κλήση της συνάρτησης `open_port/2`.
- Funs, κλεισίματα συναρτήσεων, γνωστά και ως ανώνυμες συναρτήσεις σε άλλες γλώσσες. Παράδειγμα: `fun (X) -> X+1` **end**.

Η Erlang επίσης υποστηρίζει τρεις σύνθετους τύπους.

- Tuples, πλειάδες, που είναι συλλογές σταθερού μεγέθους. Παράδειγμα: `{adam,24,{july,29}}`.
- Lists, που είναι λίστες μεταβλητού μεγέθους. Είναι παρόμοιες με τους τύπους δεδομένων λιστών σε άλλες συναρτησιακές γλώσσες προγραμματισμού. Η μόνη λειτουργία που υλοποιείται στις λίστες είναι το cons `|` που χωρίζει το πρώτο στοιχείο της λίστας από την ουρά της. Παραδείγματα: `[], [a,2,{c,4}], [1|[2|[]]]`.
- Maps, που περιέχουν έναν μεταβλητό αριθμό από ζεύγη με κλειδιών και τιμών, όπου το κλειδί και η τιμή είναι τύποι δεδομένων της Erlang. Παράδειγμα: `#{name=>adam, age=>24, birthdate=>{29,july,1984}}`.

Υπάρχουν επίσης δύο κύριες μορφές "συντακτικής ζάχαρης" όσον αφορά αυτούς τους τύπους δεδομένων.

- Strings, που είναι είναι απλές λίστες ακέραιων κωδικών ASCII. Παραδείγματα: `"hello", [104,101,108,108,111]`
- Records, που επιτρέπουν στα στοιχεία μιας πλειάδας να αποκτήσουν ένα όνομα έτσι ώστε να μπορούν να αναφέρονται με το όνομα τους και όχι με τη θέση τους στην πλειάδα. Παράδειγμα: `#person{name=adam, age=24, birthdate={29,july,1984}}`.

Η Erlang περιέχει μεταβλητές όπως οι περισσότερες άλλες γλώσσες. Μπορούν να θεωρηθούν ως μεταβλητές μεμονωμένης εκχώρησης από άλλες τυπικές γλώσσες. Στην πράξη δεσμεύονται σε μια τιμή και στη συνέχεια την αντιπροσωπεύουν για το υπόλοιπο πρόγραμμα. Γράφονται χρησιμοποιώντας μια ακολουθία χαρακτήρων ξεκινώντας με ένα κεφαλαίο γράμμα ή την κάτω πάυλα _. Παραδείγματα: `Var, _V, _`.

Οι μεταβλητές στην Erlang δεσμεύονται σε τιμές χρησιμοποιώντας έναν μηχανισμό αντιστοίχισης μοτίβων (pattern matching). Στην Erlang ένας όρος ορίζεται αναδρομικά είτε ως βασικός τύπος δεδομένων είτε ως πλειάδα όρων ή ως μια λίστα όρων ή ως ένας χάρτης όρων. Πρακτικά ένας όρος στην Erlang είναι οποιοσδήποτε τύπος δεδομένων.

Τα μοτίβα (patterns) ορίζονται αναδρομικά είτε ως βασικοί τύποι δεδομένων είτε ως μεταβλητές ή ως πλειάδες μοτίβων ή ως λίστες ή ως χάρτες.

Αφού ορίσαμε τους όρους και τα μοτίβα, η αντιστοίχιση μοτίβων μπορεί να θεωρηθεί ως μια αναδρομική σύγκριση μεταξύ ενός μοτίβου και ενός όρου.

Ακολουθούν μερικά παραδείγματα επιτυχημένων και ανεπιτυχών αντιστοίχίσεων μοτίβων.

```
%% Successful pattern matches
1 = 1.
X = 2.
{Y, Z} = {1,3}.
[A, 2] = [42, 2].

%% Unsuccessful pattern matches
{B, 1} = {1,2}.
[] = [1].
```

Listing 2.1: Παραδείγματα επιτυχημένων και ανεπιτυχών αντιστοιχίσεων μοτίβων.

Μερικές φορές είναι χρήσιμο να επεκτείνεται μια αντιστοίχιση προτύπου με ένα σύνολο περιορισμών στο αντιστοιχισμένο μοτίβο. Αυτό επιτυγχάνεται με τη χρήση εκφράσεων φρουρών (guards), οι οποίες είναι εκφράσεις που περιορίζουν τα μοτίβα με κατηγορήματα. Εισάγονται με τη λέξη-κλειδί **when** και μπορούν να περιέχουν μόνο μια ακολουθία δυαδικών τελεστών, όπως `<,>, ...` και ορισμένες ενσωματωμένες συναρτήσεις (built-in functions). Παρακάτω παρατίθενται ορισμένα παραδείγματα μοντέλων που έχουν επεκταθεί με εκφράσεις φρουρούς.

```
{Tag, Message} when Tag =/= quit
Number when Number >= 0
```

Listing 2.2: Παραδείγματα φρουρούμενων μοτίβων.

Το πιο θεμελιώδες κατασκεύασμα της Erlang είναι οι συναρτήσεις (functions). Οι συναρτήσεις στην Erlang ορίζονται με παρόμοιο τρόπο όπως και σε άλλες συναρτησιακές γλώσσες. Ο ορισμός μιας συνάρτησης περιλαμβάνει υποορισμούς (clauses), καθένας από τους οποίους αποτελείται από μια κεφαλή και ένα σώμα. Η κεφαλή περιέχει μοτίβα και φρουρούς και το σώμα περιέχει μια ακολουθία εκφράσεων.

```
FunctionName(P11, ..., P1N) when G11, ..., G1M ->
  Body1;
FunctionName(P21, ..., P2N) when G21, ..., G2M ->
  Body2;
...
FunctionName(PK1, ..., PKN) when GK1, ..., GKM ->
  BodyK.
```

Listing 2.3: Παράδειγμα ενός ορισμού μιας συνάρτησης.

Στο παραπάνω παράδειγμα, το `FunctionName` είναι atom, τα `P11, ..., PKN` είναι μοτίβα, οι `G11, ..., GKM` είναι φρουροί και οι `Body1, ... , BodyK` είναι ακολουθίες εκφράσεων.

Για να αποτιμηθεί μια κλήση συνάρτησης `Fun(Arg1, Arg2, ..., ArgN)` πρέπει να βρεθεί ένας ικανοποιητικός ορισμός. Ο ικανοποιητικός ορισμός είναι ο πρώτος που έχει το ίδιο όνομα συνάρτησης `Fun = FunctionName` και όλα τα ορίσματα ταιριάζουν με τα προστατευμένα μοτίβα `Arg1 = Pi1 when Gi1, ...`. Αφού βρεθεί ένας ικανοποιητικός ορισμός, όλες οι ελεύθερες μεταβλητές που εμφανίζονται στα πρότυπα αυτού δεσμεύονται με τα πραγματικά ορίσματα. Στη συνέχεια αποτιμούνται οι εκφράσεις του σώματος. Η κλήση συνάρτησης στη συνέχεια επιστρέφει την τιμή της τελευταίας έκφρασης στο σώμα της.

Ένα σημαντικό χαρακτηριστικό των συναρτήσεων στην Erlang είναι ότι μπορεί να είναι αναδρομικές. Μια συνάρτηση αναδρομής-ουράς (tail-recursive) είναι μια συνάρτηση της οποίας οι τελικές εκφράσεις στο σώμα της είναι είτε μεταβλητές είτε κλήσεις συναρτήσεων, γνωστές και ως κλήσεις ουράς (tail-calls). Ας το ξεκαθαρίσουμε με ένα παράδειγμα:

```
1  %% Not tail recursive as one body ends with N * factorial(N-1)
2  factorial(0) -> 1;
3  factorial(N) -> N * factorial(N-1).
4
5  %% A tail recursive way of writing factorial
6  factorial(N) -> factorial_1(N, 1).
7  factorial_1(0, X) -> X;
8  factorial_1(N, X) -> factorial_1(N-1, N*X).
```

Listing 2.4: Παράδειγμα μιας tail-recursive συνάρτησης.

Ο λόγος που η αναδρομή ουράς είναι σημαντική είναι ότι μπορεί να αντικαταστήσει επαρκώς τους βρόχους που υπάρχουν σε άλλες γλώσσες. Επειδή η Erlang δεν περιέχει βρόχους, η αναδρομή ουράς είναι ο μόνος τρόπος για να επιτευχθεί αυτή η συμπεριφορά. Ας δούμε το παρακάτω παράδειγμα για να γίνει πιο ξεκάθαρο:

```
1  p() ->
2    ...
3    q(),
4    ...
5
6  q() ->
7    r(),
8    s().
```

Listing 2.5: Παράδειγμα αναδρομής ουράς που χρησιμοποιείται σαν απλός βρόχος.

Σε κάποιο σημείο κατά τη διάρκεια της αποτίμησης της `p`, καλείται η συνάρτηση `q`. Η τελική έκφραση στο σώμα της `q` είναι μια κλήση στην `s`. Η `s` θα επιστρέψει μια τιμή στην `q`, και η `q` θα την επιστρέψει απλώς χωρίς τροποποίηση στην `p`. Οι κλήσεις συναρτήσεων κανονικά μεταγλωττίζονται σε κώδικα που διατηρεί κάπου κάποια διεύθυνση επιστροφής (συνήθως στη στοίβα), οπότε τα όρια μνήμης επιτρέπουν περιορισμένο αριθμό κλήσεων λειτουργίας, καθώς η στοίβα θα γεμίσει. Δεδομένου ότι η `q` δεν τροποποιεί την επιστρεφόμενη τιμή με κάποιο τρόπο, δεν χρειάζεται να διατηρηθεί η διεύθυνση επιστροφής `q` και μια κλήση ουράς μπορεί να μεταγλωττιστεί σε μια απλή εντολή άλματος [Stee77]. Εξαιτίας αυτού, οι tail-recursive συναρτήσεις μπορούν να αντικαταστήσουν τους βρόχους χωρίς να καταναλώνουν περαιτέρω μνήμη.

Η Erlang περιέχει επίσης μια συλλογή ενσωματωμένων συναρτήσεων(built-in functions — BIFs), που είναι υλοποιημένα σε κώδικα C, και υλοποιούν διεργασίες που συνήθως είναι πολύ χαμηλού επιπέδου για να υλοποιηθούν σε Erlang.

Ο κώδικας Erlang οργανώνεται σε αρθρώματα (modules). Τα modules περιέχουν μια ακολουθία από χαρακτηριστικά (attributes) και ορισμούς συναρτήσεων. Ένα παράδειγμα δίνεται παρακάτω.

```
1  -module(factorial).  % module attribute
2  -export([fact/1]).   % module attribute
3
4  fact(N) when N>0 ->  % beginning of function declaration
5    N * fact(N-1);     %  |
6  fact(0) ->           %  |
7    1.                 % end of function declaration
```

Listing 2.6: Παράδειγμα ορισμού ενός module.

Ταυτόχρονη Erlang

Αφού περιγράψαμε συνοπτικά το σειριακό υποσύνολο της Erlang, είναι δυνατό να προχωρήσουμε στο παράλληλο τμήμα του. Όπως σημειώθηκε παραπάνω, η Erlang βασίζεται σε μεμονωμένες διεργασίες, οι οποίες αλληλεπιδρούν με τη μετάδοση μηνυμάτων.

Μια διεργασία (process) είναι μια αυτόνομη ξεχωριστή μονάδα υπολογισμού, η οποία υπάρχει ταυτόχρονα με άλλες διεργασίες στο ίδιο σύστημα. Δεν υπάρχει εγγενής ιεραρχία μεταξύ των διεργασιών, ωστόσο ο προγραμματιστής μπορεί να επιβάλει ρητά μια, αν το επιθυμεί.

Η δημιουργία διεργασιών γίνεται με την κλήση του BIF **spawn**/3, το οποίο δημιουργεί μια νέα διεργασία και ξεκινά την εκτέλεση της, χρησιμοποιώντας τη δεδομένη συνάρτηση και τα ορίσματα. Μια κλήση στη **spawn** επιστρέφει αμέσως μετά τη δημιουργία της νέας διεργασίας και δεν περιμένει μέχρι την αποτίμηση της συνάρτησης εκκίνησης. Όταν ολοκληρωθεί η αποτίμηση της συνάρτησης έναρξης, η διεργασία τερματίζει αυτόματα.

Οι διεργασίες επικοινωνούν μεταξύ τους μόνο μέσω της μετάδοσης μηνυμάτων. Τα μηνύματα αποστέλλονται σε μια διεργασία με `Pid` χρησιμοποιώντας την εντολή αποστολής `Pid ! Message`.

Η αποστολή (!) πρώτα αποτιμάει τα ορίσματα της και επιστρέφει το μήνυμα που στάλθηκε μετά την αποτίμηση. Η εντολή αποστολής είναι ασύγχρονη, οπότε θα επιστρέψει αμέσως και δεν θα περιμένει το μήνυμα να φτάσει στον προορισμό ή να παραληφθεί. Το σύστημα δεν θα ειδοποιήσει τον αποστολέα εάν η διεργασία στην οποία αποστέλλεται το μήνυμα έχει ήδη τερματιστεί. Η ίδια η εφαρμογή πρέπει να εφαρμόζει όλες τις μορφές ελέγχου. Το σύστημα διασφαλίζει μόνο ότι τα μηνύματα παραδίδονται πάντα στον παραλήπτη και ότι παραδίδονται πάντα με την ίδια σειρά που αποστέλλονται από τον ίδιο αποστολέα στον ίδιο δέκτη. Για παράδειγμα, μέσα στον ίδιο κόμβο (node), εάν δύο μηνύματα `Msg1` και `Msg2` αποστέλλονται από τη διεργασία A στη διεργασία B με αυτή τη σειρά, θα φτάνουν πάντα στη B με αυτή τη σειρά.

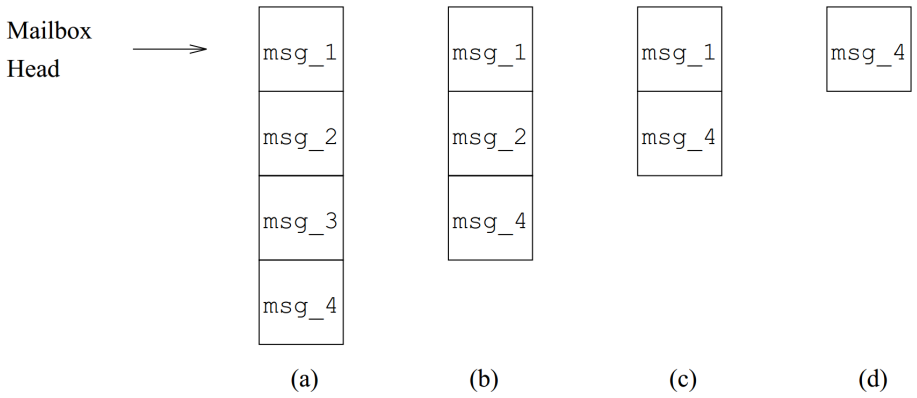
Η λήψη μηνυμάτων επιτυγχάνεται χρησιμοποιώντας την εντολή **receive**, η οποία έχει την ακόλουθη σύνταξη:

```
1  receive
2    Message1 [when Guard1] ->
3      Actions1;
4    Message2 [when Guard2] ->
5      Actions2;
6    ...
7  end
```

Listing 2.7: Η σύνταξη της εντολής receive.

Κάθε διεργασία έχει ένα γραμματοκιβώτιο (mailbox) όπου όλα τα μηνύματα που αποστέλλονται στη διεργασία αποθηκεύονται με τη σειρά που έφθασαν. Το **receive** ελέγχει όλα τα πρότυπα `Message1`, `Message2`, ... και προσπαθεί να τα ταιριάζει με τα μηνύματα στο γραμματοκιβώτιο της διεργασίας. Όταν βρεθεί ένα αντίστοιχο μήνυμα `MessageN` και ο αντίστοιχος φρουρός ικανοποιηθεί, το μήνυμα αφαιρείται από το γραμματοκιβώτιο και στη συνέχεια αποτιμούνται τα `ActionsN`. Τα μηνύματα που παραμένουν στο γραμματοκιβώτιο και δεν έχουν επιλεγεί, θα παραμείνουν στο γραμματοκιβώτιο με την ίδια σειρά με την παράδοσή τους και θα αντιστοιχιστούν με το επόμενο **receive**. Η αποτίμηση του **receive** θα μπλοκάρει μέχρι να βρεθεί ένα αντίστοιχο μήνυμα στο γραμματοκιβώτιο.

Είναι δυνατή η εφαρμογή ενός μηχανισμού επιλεκτικής λήψης (selective receive) στην Erlang, ωστόσο, καθώς τα μηνύματα που δεν αντιστοιχίζονται παραμένουν στο γραμματοκιβώτιο, είναι ευθύνη της εφαρμογής να βεβαιωθεί ότι το σύστημα δεν γεμίζει με αξεπέραστα μηνύματα. Ακολουθεί ένα παράδειγμα [Vird96] που δείχνει τη σειρά λήψης μηνυμάτων στην Erlang.



Σχήμα 2.1: Υποδοχή μηνυμάτων.

Ένα γραμματοκιβώτιο διεργασίας αρχικά περιέχει τέσσερα μηνύματα (Σχήμα 2.1 (a)) `msg_1`, `msg_2`, `msg_3`, `msg_4` με αυτή τη σειρά. Στη συνέχεια αποτιμάται το ακόλουθο **receive**:

```
1  receive
2    msg_3 ->
3      ...
4  end
```

Listing 2.8: Το πρώτο receive που αποτιμάται.

Το αποτέλεσμα είναι ότι το `msg_3` αντιστοιχεί και στη συνέχεια αφαιρείται από το γραμματοκιβώτιο, όπως φαίνεται στο Σχήμα 2.1 (b). Όταν αποτιμάται το παρακάτω receive:

```
1  receive
2    msg_4 ->
3      ...
4    msg_2 ->
5      ...
6  end
```

Listing 2.9: Δεύτερο receive.

Αυτό θα προσπαθήσει να ταιριάξει κάθε μήνυμα στο γραμματοκιβώτιο με το `msg_4` και εν συνεχεία με το `msg_2`. Αυτό έχει ως αποτέλεσμα το `msg_2` να ταιριάζει και να αφαιρείται, όπως φαίνεται στο Σχήμα 2.1 (c). Τέλος θα εκτελεστεί το:

```
receive
  AnyMessage ->
    ...
end
```

Listing 2.10: Τρίτο receive

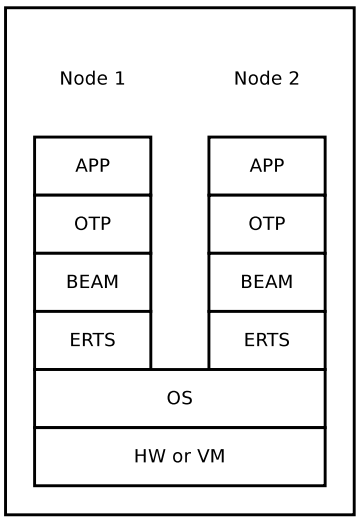
όπου το `AnyMessage` είναι μη δεσμευμένο, οδηγεί στην αντιστοίχηση και κατάργηση του `msg_1` από το γραμματοκιβώτιο, όπως φαίνεται στο Σχήμα 2.1 (d).

2.1.2 Το Σύστημα Χρόνου Εκτέλεσης της Erlang (ERTS)

Το σύστημα χρόνου εκτέλεσης της Erlang είναι πολύπλοκο και περιέχει πολλά ανεξάρτητα στοιχεία. Δεν υπάρχει επίσημος ορισμός για το τι είναι το σύστημα χρόνου εκτέλεσης της Erlang, επομένως θα επικεντρωθούμε στην *de facto* υλοποίηση του συστήματος Erlang/OTP, το οποίο αναπτύσσεται και συντηρείται από την Ericsson. Θα το ονομάσουμε ERTS (Erlang Run Time System).

Μια εφαρμογή ή ένα σύστημα Erlang είναι συνήθως ένας κόμβος που τρέχει το ERTS και την εικονική μηχανή BEAM. Σύμφωνα με το εγχειρίδιο του Erlang/OTP, ένας κόμβος είναι ένα σύστημα χρόνου εκτέλεσης που έχει ένα όνομα.

Τα επίπεδα που τρέχει μια εφαρμογή Erlang εμφανίζονται στο Σχήμα 2.2.



Σχήμα 2.2: Στοίβα ERTS.

Πρέπει να διασφαλίσουμε ότι το ERTS ικανοποιεί τις απαιτήσεις που συζητήσαμε στήν Ενότητα 2.1.1. Όπως υπογραμμίσαμε, η Erlang είναι μια γλώσσα προσανατολισμένη στον ταυτοχρονισμό και έτσι πρέπει να διατηρεί έναν μεγάλο αριθμό διεργασιών που εκτελούνται ταυτόχρονα. Για να επιτευχθεί αυτό, οι διεργασίες αυτές πρέπει να προγραμματιστούν με αποτελεσματικό τρόπο. Πιο συγκεκριμένα, θα πρέπει να πληρούνται τα δύο ακόλουθα κριτήρια: [Vird96]:

- Ο αλγόριθμος χρονοδρομολόγησης πρέπει να είναι δίκαιος, έτσι ώστε να εκτελείται κάθε διεργασία που μπορεί να εκτελεστεί. Κατά προτίμηση οι διεργασίες θα εκτελούνται με τη σειρά κατά την οποία έγιναν διαθέσιμες.
- Καμία διεργασία δεν θα επιτρέπεται να μπλοκάρει για μεγάλο χρονικό διάστημα. Κάθε διεργασία επιτρέπεται να εκτελείται για σύντομο χρονικό διάστημα (time slice), προτού χρονοδρομολογηθεί για να επιτραπεί η εκτέλεση μιας άλλης διεργασίας.

Τα χρονικά διαστήματα συνήθως ρυθμίζονται ώστε να επιτρέπουν στη διεργασία εκτέλεσης να εκτελέσει έναν σταθερό αριθμό βημάτων[2], συνήθως γύρω στις 2000, προτού χρονοδρομολογηθούν.

Η Erlang θα πρέπει να είναι κατάλληλη για εφαρμογές πραγματικού χρόνου και λόγω αυτού, οι χρόνοι απόκρισης θα πρέπει να είναι της τάξης των χιλιοστών του δευτερολέπτου. Ένας αλγόριθμος χρονοδρομολόγησης που πληροί τα παραπάνω κριτήρια θεωρείται αρκετά καλός για μια υλοποίηση της Erlang [Vird96].

Καθώς η Erlang αποκρύπτει όλη τη διαχείριση μνήμης από το χρήστη, είναι σημαντικό η αυτόματη διαχείριση μνήμης να μην παραβιάζει τα κριτήρια που αναφέρονται παραπάνω. Στην ουσία η αυτόματη διαχείριση μνήμης θα πρέπει να γίνεται κατά τρόπο ώστε να μην μπλοκάρει το σύστημα για μεγάλο χρονικό διάστημα, κατά προτίμηση για μικρότερο χρονικό διάστημα από την περίοδο χρονοδρομολόγησης μιας μόνο διεργασίας.

Πριν εμβαθύνουμε στο ERTS και τις λειτουργίες του, θα κάνουμε μια σύντομη επισκόπηση των συστατικών του και θα ορίσουμε κάποιες απαραίτητες έννοιες.

Μια διεργασία της Erlang είναι παρόμοια με μια διεργασία του λειτουργικού συστήματος. Καθέμια έχει τη δική της μνήμη (στοίβα, σωρό, γραμματοκιβώτιο) και ένα μπλοκ ελέγχου διεργασίας (PCB) με πληροφορίες για τη διεργασία. Η εκτέλεση κώδικα Erlang πραγματοποιείται στο πλαίσιο μιας διεργασίας. Οι διεργασίες επικοινωνούν μεταξύ τους μόνο με την ανταλλαγή μηνυμάτων. Αυτό ισχύει ακόμα και για διεργασίες που βρίσκονται σε διαφορετικούς κόμβους.

Ο μεταγλωττιστής της Erlang, όπως υπονοεί το όνομά του, μεταγλωττίζει τον πηγαίο κώδικα Erlang που περιέχεται στα αρχεία `.erl` σε bytecode εικονικής μηχανής για την εικονική μηχανή BEAM.

Ο χρονοδρομολογητής είναι υπεύθυνος για την επιλογή της διεργασίας που θα εκτελεστεί ανά πάσα στιγμή, όμοια με τον χρονοδρομολογητή του λειτουργικού συστήματος.

Το ERTS προσφέρει αυτόματη διαχείριση μνήμης και παρακολουθεί τη στοίβα και το σωρό κάθε διεργασίας. Ο συλλέκτης απορριμμάτων ακολουθεί έναν αλγόριθμο συλλογής απορριμμάτων βασισμένο σε αντιγραφή, που επεκτείνεται με την καταμέτρηση αναφορών για bistrings τα οποία είναι μεγαλύτερα από κάποιο μέγεθος.

Το BEAM είναι η εικονική μηχανή που χρησιμοποιείται για την εκτέλεση του Erlang bytecode. Εκτελείται σε έναν κόμβο Erlang. Το BEAM υποστηρίζει δύο επίπεδα εντολών: Γενικές εντολές και ειδικές εντολές. Καθώς δεν υπάρχει γενικός ορισμός της εικονικής μηχανής Erlang (Erlang Virtual Machine), θα θεωρήσουμε ότι το BEAM και το σύνολο εντολών του είναι επαρκές για την εκτέλεση της Erlang.

Διεργασίες Erlang

Έχουμε ήδη συζητήσει τι είναι μια διεργασία σε υψηλό επίπεδο. Στην πράξη είναι απλώς μνήμη. Μια διεργασία Erlang είναι βασικά τέσσερα τμήματα μνήμης: μία στοίβα, ένας σωρός, μια περιοχή μηνυμάτων και το μπλοκ ελέγχου διεργασίας (PCB).

- Η στοίβα χρησιμοποιείται για την αποθήκευση διευθύνσεων επιστροφής, για την μεταφορά ορισμάτων σε συναρτήσεις και για την αποθήκευση τοπικών μεταβλητών.
- Ο σωρός χρησιμοποιείται για την αποθήκευση μεγαλύτερων σύνθετων δομών, όπως πλειάδες, λίστες και χάρτες.
- Η περιοχή μηνυμάτων χρησιμοποιείται για την αποθήκευση των μηνυμάτων που αποστέλλονται στη διεργασία από άλλες διεργασίες. Είναι επίσης γνωστή ως το γραμματοκιβώτιο της διεργασίας.

[2] Για απλότητα, ως βήμα μπορεί να θεωρηθεί απλά η κάθε κλήση συνάρτησης.

- Το μπλοκ ελέγχου της διεργασίας (PCB) χρησιμοποιείται για να παρακολουθεί την κατάσταση της διεργασίας, όπως διαχειρίζεται το λειτουργικό σύστημα την κατάσταση μιας διεργασίας του.

Ως αποτέλεσμα, το PCB δεσμεύεται στατικά και περιέχει ένα σταθερό αριθμό πεδίων που ελέγχουν τη διεργασία. Η στοίβα, ο σωρός και το γραμματοκιβώτιο από την άλλη πλευρά δεσμεύονται στη μνήμη δυναμικά, καθώς το μέγεθός τους ποικίλλει κατά τη διάρκεια της εκτέλεσης.

Υπάρχει επίσης μια άλλη περιοχή μνήμης που έχουν όλες οι διεργασίες. Το Λεξικό Διεργασίας (Process Dictionary), το οποίο είναι τοπικός αποθηκευτικός χώρος με ζεύγη κλειδιών-τιμών. Επειδή πρόκειται για μια μικρή περιοχή μνήμης, συμβαίνουν συχνά συγκρούσεις λόγω της συνάρτησης κατακερματισμού, οπότε κάθε τιμή δείχνει σε μια λίστα (bucket). Το Λεξικό Διεργασίας δε χρησιμοποιείται για την αποθήκευση μεγάλου όγκου δεδομένων καθώς η υλοποίηση του δεν είναι βέλτιστη.

Χρονοδρομολόγηση

Ο χρονοδρομολογητής είναι υπεύθυνος για την επιλογή της διεργασίας που θα εκτελεστεί ανά πάσα στιγμή, όμοια με τον χρονοδρομολητή του λειτουργικού συστήματος. Μια απλή περιγραφή είναι ότι ο χρονοδρομολογητής διατηρεί δύο ουρές, μια ουρά των διεργασιών που είναι έτοιμες να εκτελεστούν και μια ουρά αναμονής των μπλοκαρισμένων διεργασιών που περιμένουν να λάβουν ένα μήνυμα. Μια διεργασία μετακινείται από την ουρά αναμονής, εάν λάβει ένα μήνυμα ή εάν συμβεί κάποιο time-out.

Ο χρονοδρομολογητής επιλέγει την πρώτη διεργασία από την έτοιμη ουρά και αφήνει το BEAM να χειριστεί την εκτέλεση του για μια χρονική περίοδο. Το BEAM προλαμβάνει τη διεργασία όταν εξαντληθεί η περίοδος και προσθέτει τη διεργασία πίσω στο τέλος της έτοιμης ουράς. Εάν η διεργασία σε κάποιο σημείο μπλοκάρει εξαιτίας μιας λήψης κατά τη διάρκεια της χρονικής περιόδου, μετακινείται στην ουρά αναμονής.

Όπως έχουμε δει, ένας χρονοδρομολογητής έχει μόνο μία διεργασία εκτέλεσης ανά πάσα στιγμή. Έτσι, ο παραλληλισμός στην Erlang επιτυγχάνεται με την ταυτόχρονη εκτέλεση περισσότερων χρονοδρομολογητών.

Συλλογή Απορριμάτων

Το ERTS προσφέρει αυτόματη διαχείριση μνήμης και παρακολουθεί τη στοίβα και τον σωρό κάθε διεργασίας. Κατά τη διάρκεια εκτέλεσης του προγράμματος, η στοίβα και ο σωρός μπορούν να αναπτυχθούν και να συρρικνωθούν ανάλογα με τις απαιτήσεις μνήμης. Η διαχείριση μνήμης βασίζεται σε μια διαδικασία αντιγραφής γενεαλογικού συλλέκτη σκουπιδιών, που έχει επεκταθεί με την καταμέτρηση αναφοράς για συγκεκριμένα δυαδικά αρχεία. Ο λόγος που η συλλογή απορριμμάτων (Garbage Collection) συμβαίνει ανά διεργασία είναι ότι, όπως περιγράφεται παραπάνω, η GC πρέπει να μπλοκάρει το σύστημα για σύντομο χρονικό διάστημα ώστε να παραμένει το σύστημα ανταποκρίσιμο.

Ο συλλέκτης απορριμμάτων ξεκινάει κάθε φορά που δεν υπάρχει ελεύθερος χώρος στον σωρό (ή στη στοίβα, καθώς μοιράζονται το ίδιο επιμερισμένο μπλοκ μνήμης). Ο συλλέκτης απορριμμάτων δεσμεύει μια νέα περιοχή μνήμης που ονομάζεται *to space*, όπως συμβαίνει και με άλλους συλλέκτες απορριμμάτων αντιγραφής. Στη συνέχεια, περνάει από τη στοίβα για να βρει όλες τις ζωντανές ρίζες, τις οποίες ακολουθεί και αντιγράφει στη νέα περιοχή σωρού που βρίσκεται στο *to space*.

Ο συλλέκτης απορριμμάτων είναι γενεαλογικός και επομένως χρησιμοποιεί μια ευριστική για να εξετάζει νεότερα δεδομένα τις περισσότερες φορές και να ελέγχει τα παλαιότερα δεδομένα μόνο περιστασιακά. Τα συνήθη μικρότερα περάσματα ονομάζονται δευτερεύουσες συλλογές, όπου μόνο νέα δεδομένα εξετάζονται για συλλογή. Αυτά

είναι συνήθως επαρκή για την ανάκτηση ελεύθερου χώρου. Τα σκουπίδια από το παλαιότερο μέρος του σωρού συλλέγονται κατά τη διάρκεια των μεγάλων συλλογών ή των πλήρων σαρώσεων, οι οποίες είναι λιγότερο συχνές.

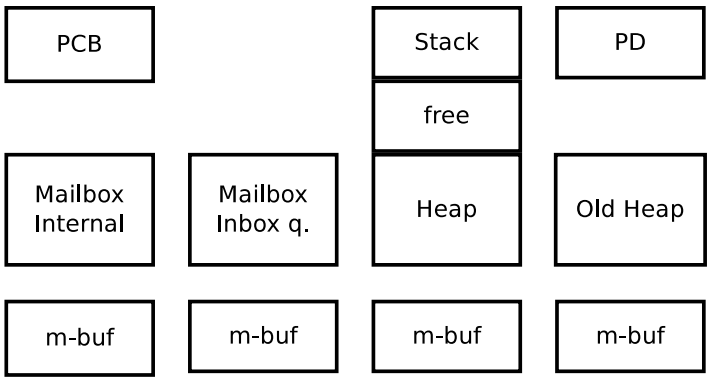
Ανταλλαγή Μηνυμάτων

Όπως επίσης περιγράφηκε παραπάνω, οι διεργασίες της Erlang επικοινωνούν μέσω ανταλλαγής μηνυμάτων. Η αποστολή ενός μηνύματος σημαίνει ότι η διεργασία αποστολής αντιγράφει ένα μήνυμα από τη δική της περιοχή μνήμης στην περιοχή μνήμης της διεργασίας λήψης.

Στις πρώτες εκδόσεις της Erlang, δεν υπήρχαν πολλοί χρονοδρομολογητές, οπότε μόνο μία διεργασία μπορούσε να εκτελεστεί κάθε φορά. Έτσι, ήταν ασφαλές για μια διεργασία αποστολέα να αντιγράψει ένα μήνυμα στο γραμματοκιβώτιο ενός λήψης. Με την ύπαρξη πολλών χρονοδρομολογητών, δεν ήταν πλέον ασφαλές να γράφει άμεσα μια διεργασία στον σωρό μιας άλλης διεργασίας, οπότε θα έπρεπε να αποκτηθεί μια κύρια κλειδαριά διεργασίας πριν από οποιαδήποτε αντιγραφή. Ωστόσο, αυτό ήταν πολύ αργό, ειδικά αν πολλές διεργασίες προσπάθησαν να στείλουν ένα μήνυμα στον ίδιο δέκτη. Ως αποτέλεσμα, εισήχθη η έννοια των θραυσμάτων σωρού έξω από την κύρια περιοχή σωρού, που ονομάζονται m-bufs. Εάν μια διεργασία αποστολέα δεν μπορεί να αποκτήσει την κύρια κλειδαριά της διεργασίας του δέκτη, αντιγράφει το μήνυμα σε ένα m-buf. Μετά την ολοκλήρωση της αντιγραφής, το μήνυμα στο m-buf συνδέεται με τη διεργασία μέσω του γραμματοκιβωτίου, προσθέτοντάς το στην ουρά μηνυμάτων του δέκτη. Ο συλλέκτης απορριμμάτων στη συνέχεια αντιγράφει τα μηνύματα στον σωρό της διεργασίας.

Προκειμένου να μειωθεί η περιττή πίεση από τη συλλογή απορριμάτων, το γραμματοκιβώτιο χωρίζεται σε δύο λίστες, η μια περιέχει μηνύματα που έχουν ήδη ελεγχθεί και η άλλη περιέχει νέα μηνύματα. Με αυτόν τον τρόπο η συλλογή απορριμάτων δεν χρειάζεται να ελέγχει τα νέα μηνύματα, καθώς βρίσκονται ακόμα στο γραμματοκιβώτιο και θα επιβιώσουν σίγουρα στη συλλογή απορριμμάτων. Τα εμφανιζόμενα μηνύματα περιέχονται στο εσωτερικό γραμματοκιβώτιο, ενώ τα μη εμφανιζόμενα μηνύματα τοποθετούνται στο γραμματοκιβώτιο της ουράς εισερχομένων.

Μια απλοποιημένη διάταξη όλων των περιοχών μνήμης μιας διεργασίας εμφανίζεται στο Σχήμα 2.3.



Σχήμα 2.3: Διάταξη μνήμης διεργασίας.

Ο Μεταγλωττιστής της Erlang

Ο μεταγλωττιστής της Erlang χρησιμοποιείται για τη δημιουργία κώδικα BEAM από τον πηγαίο κώδικα Erlang. Περιέχει πολλά ανεξάρτητα περάσματα μεταγλωττιστή και

επιτρέπει αρκετές ενδιάμεσες εξόδους κατά τη σύνταξη. Θα περιγράψουμε συνοπτικά όλα τα τυπικά περάσματα μεταγλωττιστή.

1. Προεπεξεργαστής Erlang (Erlang Pre-Processor): Το πρώτο πέρασμα μεταγλωττιστή είναι ένας tokenizer σε συνδυασμό με έναν προεπεξεργαστή. Αρχικά, επεκτείνονται οι μακροεντολές ως tokens[3]. Μία σωστή τιμή μακροεντολής δεν πρέπει απαραίτητα να είναι ένας έγκυρος όρος Erlang.

2. Συντακτικοί Μετασχηματισμοί (Parse Transformations): Επιτρέπει να τροποποιηθεί ο πηγαίος κώδικας Erlang. Οι συντακτικοί μετασχηματισμοί δουλεύουν σε ένα αφηρημένο δένδρο σύνταξης (AST) και αναμένεται να επιστρέψουν ένα νέο AST. Επιτρέπεται η επιστροφή μη έγκυρων AST, καθώς οι έλεγχοι εγκυρότητας συμβαίνουν σε μεταγενέστερο μεταβατικό πέρασμα.

3. Linter: Παράγει προειδοποιήσεις για συντακτικά σωστό αλλά "κακό" κώδικα. Για παράδειγμα, η ενεργοποίηση της σημαίας `export_all` οδηγεί σε προειδοποίηση από το πέρασμα του Linter.

4. Αποθήκευση Αφηρημένου Συντακτικού Δέντρου (Save AST): Αυτό το πέρασμα χρησιμοποιείται για την αποθήκευση ενός AST σε αρχείο εξόδου για σκοπούς εντοπισμού σφαλμάτων. Είναι ενεργοποιημένο μέσω μιας επιλογής μεταγλωττιστή. Σημειώστε ότι το αποθηκευμένο AST δεν περιέχει βελτιστοποίησεις καθώς συμβαίνουν σε μεταγενέστερα περάσματα.

5. Επέκταση (Expand): Σε αυτό το πέρασμα, οι κατασκευές γλωσσικού επιπέδου πηγαίου κώδικα επεκτείνονται σε κατασκευάσματα χαμηλότερου επιπέδου (π.χ., οι εγγραφές επεκτείνονται σε πλειάδες).

6. Core Erlang: Είναι μια αυστηρή συναρτησιακή ενδιάμεση γλώσσα. Βρίσκεται μεταξύ του πηγαίου κώδικα Erlang και του ενδιάμεσου κώδικα που βρίσκεται συνήθως στους μεταγλωττιστές [Carl04]. Είναι καθαρή και απλή και επιτρέπει λίγους τρόπους για να εκφραστούν οι ίδιες λειτουργίες. Έτσι διευκολύνει τους μετασχηματισμούς κώδικα και τις βελτιστοποιήσεις του μεταγλωττιστή. Η Core Erlang είναι τακτική, με σαφώς καθορισμένη σημασιολογία, προκειμένου να διευκολύνει τα εργαλεία που λειτουργούν σε αυτό.

7. Kernel Erlang: Πρόκειται για μια επίπεδη έκδοση της Core Erlang, όπου κάθε μεταβλητή είναι μοναδική. Η αντιστοίχιση μοτίβων στον πυρήνα Erlang μεταγλωττίζεται σε πιο βασικές λειτουργίες.

8. Κωδικας BEAM: Το τελευταίο βήμα της τυπικής σύνταξης είναι η μορφή bytecode BEAM.

9. Κώδικας Μηχανής (Native Code): Αυτό είναι ένα προαιρετικό τελευταίο βήμα της μεταγλώττισης, όταν ενεργοποιείται το HiPE [Joha00], ο μεταγλωττιστής κώδικα μηχανής της Erlang. Σε αυτό το βήμα δημιουργείται και αποθηκεύεται κώδικας μηχανής μαζί με τον κώδικα BEAM στο αρχείο `.beam`. Αυτό το βήμα περιέχει πολλά μικρότερα βήματα που θα περιγραφούν περαιτέρω στην Ενότητα 2.1.4.

[3] Σημειώστε ότι η επέκταση δεν είναι απλή αντικατάσταση συμβολοσειράς, καθώς η μακροεντολή θα επεκταθεί πάντα ως ξεχωριστό διακριτικό και δεν μπορεί να συνδεθεί με ένα υπάρχον διακριτικό.

Φόρτωση Κώδικα

Στη συντριπτική πλειονότητα των γλωσσών προγραμματισμού η αλλαγή του κώδικα που εκτελείται σε ένα σύστημα είναι μια απλή διαδικασία. Το σύστημα σταματάει, αλλάζει ο κώδικας και στη συνέχεια επανεκκινείται το σύστημα.

Ωστόσο, η Erlang έχει σχεδιαστεί για την υλοποίηση συστημάτων με απαιτήσεις πραγματικού χρόνου, τα οποία συχνά δεν επιτρέπεται να έχουν κανένα χρόνο διακοπής. Ως αποτέλεσμα, πρέπει να περιέχουν ένα μηχανισμό που επιτρέπει την απρόσκοπτη αλλαγή κώδικα χωρίς την ανάγκη επανεκκίνησης. Σε ένα σειριακό σύστημα αυτό θα μπορούσε εύκολα να γίνει με μια σύντομη παύση, αλλά σε ένα σύστημα με πολλές ταυτόχρονες διεργασίες ο μηχανισμός θα πρέπει να είναι πιο περίπλοκος. Αυτός ο μηχανισμός, γνωστός ως φόρτωση καυτού κώδικα (hot-code loading), παρέχεται από το ERTS και η λειτουργικότητά του περιγράφεται παρακάτω.

Το ERTS επιτρέπει την ταυτόχρονη φόρτωση δύο εκδόσεων κώδικα για κάθε module στο σύστημα [Arms03]. Από εδώ και πέρα θα ονομάσουμε αυτές τις δύο εκδόσεις *παλιά* και *τρέχουσα*. Την πρώτη φορά που ένα άρθρωμα φορτώνεται στο σύστημα, ο κώδικάς του θεωρείται *παλιός*. Όταν φορτώνεται μια νέα έκδοση του αρθρώματος, ο προηγουμένως φορτωμένος κώδικας γίνεται *παλιός* και ο κώδικας της νέας έκδοσης θεωρείται ο *τρέχων*.

Είναι σημαντικό να σημειωθεί ότι, για να υποστηριχθεί η φόρτωση καυτού κώδικα, η γλώσσα κάνει μια σημασιολογική διάκριση μεταξύ των τοπικών κλήσεων συναρτήσεων, οι οποίες έχουν τη μορφή `f(...)` και τις αποκαλούμενες *απομακρυσμένες κλήσεις* που έχουν τη μορφή `m:f(...)` και πρέπει να αναζητήσουν την πιο πρόσφατα φορτωμένη έκδοση του αρθρώματος `m`.

Το πλεονέκτημα της ύπαρξης δύο συνυπάρχουσων εκδόσεων κώδικα είναι ότι η αλλαγή κώδικα συμβαίνει βαθμιαία με τρόπο ανά διεργασία. Κάθε διεργασία αλλάζει από *παλίο σε τρέχων* κώδικα όταν εκτελεί μια απομακρυσμένη κλήση. Για λόγους σαφήνειας παρουσιάζουμε το ακόλουθο παράδειγμα:

```
1  -module(m).
2  -export([loop/0]).
3
4  loop() ->
5    receive
6      code_switch ->
7        %% An external function call that could lead to a code change
8        m:loop();
9      Otherwise ->
10       %% A local function call
11       loop()
12   end.
```

Listing 2.11: Ένας βρόχος που επιτρέπει φόρτωση καυτού κώδικα.

Σε αυτό το παράδειγμα, μια διεργασία που εκτελεί το παραπάνω `loop` σε μια *παλιά* έκδοση κώδικα θα μεταβεί στον *τρέχων* κώδικα μόνο αν λάβει το μήνυμα `code_switch`.

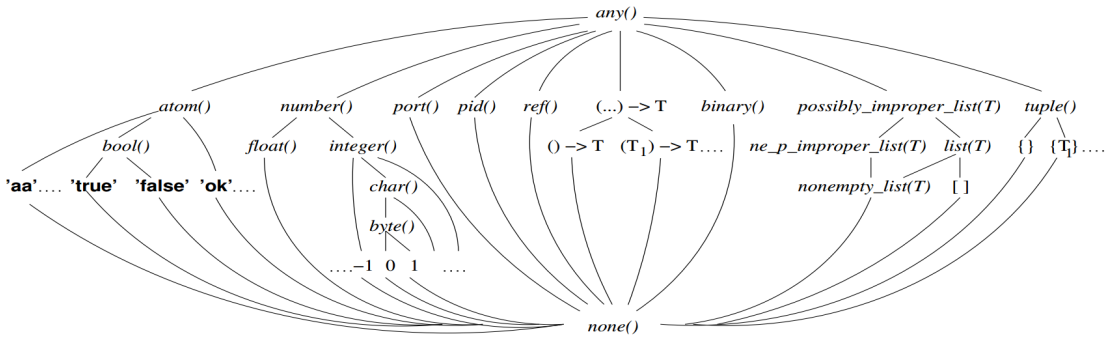
Όταν όλες οι διεργασίες έχουν μεταβεί στον *τρέχων* κώδικα, ο *παλιός* κώδικας αποδεσμεύεται από τη μνήμη. Ωστόσο, το σύστημα περιορίζεται σε δύο ταυτόχρονες εκδόσεις κώδικα. Επομένως, αν φορτωθεί μια νεότερη έκδοση ενός αρθρώματος, ενώ οι διεργασίες εξακολουθούν να εκτελούν τον *παλιό* κώδικα αυτού του αρθρώματος, αυτές οι διεργασίες θανατώνονται αμέσως και ο *παλιός* κώδικας καθαρίζεται. Στη συνέχεια, ο νεότερος κώδικας γίνεται *τρέχων* και ο προηγούμενος *τρέχων* κώδικας γίνεται *παλιός*. Εξαιτίας αυτού, είναι απαραίτητο οι προγραμματιστές της Erlang να είναι εξαιρετικά προσεκτικοί κατά τη φόρτωση νέου κώδικα, έτσι ώστε να μη συμβαίνει κάποιος ανεπιθύμητος τερματισμός διεργασιών.

2.1.3 Το Σύστημα Τύπων της Erlang

Η Erlang είναι μια γλώσσα με δυναμικό αλλά αυστηρό σύστημα τύπων. Αυστηρό επειδή ένας όρος δεν μπορεί να αλλάξει σιωπηλά σε έναν όρο άλλου τύπου. Ένα παράδειγμα σιωπηλής αλλαγής στην C (μια γλώσσα με αδύναμους τύπους) θα ήταν η χρήση μιας τιμής τύπου `char` αντί μιας τιμής τύπου `int`. Οι όροι Erlang μπορούν να μετατραπούν σε όρους διαφορετικών τύπων μόνο ρητά με τη χρήση κατάλληλων εντολών. Το σύστημα τύπων της Erlang είναι δυναμικό επειδή οι τύποι και η ορθότητα τύπων ελέγχονται κατά το χρόνο εκτέλεσης με προσθήκη ελέγχων τύπου (type tests). Ο έλεγχος τύπων κατά το χρόνο εκτέλεσης απαιτεί επίσης έναν τρόπο σύνδεσης των όρων με τους τύπους κατά την εκτέλεση του προγράμματος. Στην Erlang αυτό γίνεται με το συνδυασμό της τιμής κάθε όρου με μια ετικέτα (μια ακολουθία δυαδικών ψηφίων) που δείχνει τον τύπο της.

Οι τύποι στο Erlang περιγράφουν σύνολα όρων. Οι προκαθορισμένοι τύποι υπάρχουν για τους περισσότερους τυπικούς όρους Erlang, όπως **`integer()`** και **`atom()`**, και για όλους τους μεμονωμένους όρους, όπως ο ακέραιος `42`. Όλοι οι άλλοι τύποι είναι κατασκευασμένοι ως ενώσεις αυτών των προκαθορισμένων τύπων.

Το σύνολο των τύπων της Erlang είναι κλειστό κάτω από μια σχέση υποτύπων $T_A \sqsubseteq T_B$ που ισχύει αν το σύνολο των όρων T_B είναι ένα υπερσύνολο του T_A. Επομένως, το σύνολο των τύπων Erlang περιέχει επίσης έναν ανώτερο τύπο (`any()`) και έναν κατώτερο (`none()`). Το πλήρες πλέγμα υποτύπων [Lind05] για όλους τους προκαθορισμένους τύπους της Erlang εμφανίζεται στο Σχήμα 2.4.



Σχήμα 2.4: Πλέγμα τύπων της Erlang.

Σε μια ένωση τύπων μεταξύ ενός τύπου και ενός από τους υποτύπους του, ο υποτύπος απορροφάται από τον υπερτύπο. Για παράδειγμα, η ένωση **`atom()`** `|` `'bar'` `|` **`integer()`** `|` `42` είναι η ίδια με την ένωση **`atom()`** `|` **`integer()`**.

Τύποι και Προδιαγραφές Συναρτήσεων

Παρά το γεγονός ότι η Erlang είναι μια γλώσσα με δυναμικό σύστημα τύπων, είναι πιθανό ο προγραμματιστής να δηλώσει ρητά πληροφορίες σχετικά με τους τύπους ορισμάτων συναρτήσεων, τις τιμές επιστροφής τους και τα πεδία εγγραφής. Αυτές οι πληροφορίες τύπου έχουν τις εξής χρήσεις:

- Για την τεκμηρίωση των διεπαφών συναρτήσεων και για τη διευκρίνιση των προθέσεων των συναρτήσεων και των προγραμματιστών.
- Για τη διευκόλυνση των εργαλείων ανίχνευσης σφαλμάτων στατικής ανάλυσης, όπως το Dialyzer [Lind04].
- Για τη δημιουργία εγχειριδίων διαφόρων μορφών χρησιμοποιώντας εργαλεία δημιουργίας εγχειριδίων, όπως το EDoc [Eric].

Οι οδηγίες και η τεκμηρίωση των συναρτήσεων είναι συνήθως γραμμένη σε σχόλια προγράμματος. Ωστόσο, τα σχόλια συχνά τείνουν να μην ανανεώνονται όσο συχνά ανανεώνεται ο κώδικας με αποτέλεσμα να υστερούν. Αυτός είναι ο κύριος λόγος για τη χρήση προδιαγραφών συναρτήσεων που, όταν συνδυάζονται με ένα εργαλείο ανίχνευσης σφαλμάτων που ελέγχει αν οι προδιαγραφές ταιριάζουν με την υλοποίηση, δεν θα μένουν πίσω σε σχέση με την υλοποίηση.

Το Σύστημα Ετικετών των Τιμών της Erlang

Όπως έχει επίσης αναφερθεί παραπάνω, στην αναπαράσταση μνήμης ενός όρου, ορισμένα bits είναι δεσμευμένα για μια ετικέτα τύπου. Οι όροι της Erlang διαχωρίζονται σε όρους άμεσους και όρους (immediate terms) κλεισμένους σε κουτί (boxed terms). Οι άμεσοι όροι μπορούν να χωρέσουν σε μια λέξη μηχανής, ενώ οι boxed αποτελούνται από δύο μέρη, έναν δείκτη με ετικέτες και έναν αριθμό λέξεων που είναι αποθηκευμένοι στο σωρό της διεργασίας. Οι λέξεις που αποθηκεύονται στον σωρό περιέχουν μια επικεφαλίδα και ένα σώμα.

Η βασική ιδέα είναι ότι τα λιγότερο σημαντικά κομμάτια μιας λέξης χρησιμοποιούνται ως ετικέτες. Καθώς οι περισσότερες σύγχρονες αρχιτεκτονικές CPU ευθυγραμμίζουν λέξεις 32 ή 64 bit, οι δείκτες έχουν τουλάχιστον δύο αχρησιμοποίητα bits. Αυτά χρησιμοποιούνται ως ετικέτα, αλλά δεν αρκούν για να αντιπροσωπεύσουν όλους τους τύπους της Erlang. Επομένως, χρησιμοποιούνται περισσότερα bits ανάλογα με τις ανάγκες.

Success Typings

Ένας αλγόριθμος συμπερασμάτων τύπων βασισμένος στα Success Typings [Lind06] χρησιμοποιείται για την Erlang προκειμένου να εντοπίσει πιθανά σφάλματα [Lind04], να σχολιάσει προγράμματα με τις προδιαγραφές λειτουργίας [Lind05] και να πραγματοποιήσει βελτιστοποιήσεις [Joha00]. Ο κύριος στόχος αυτού του αλγορίθμου είναι να αποκαλύψει όσο το δυνατόν περισσότερες πληροφορίες τύπων χωρίς να υποτιμάει ποτέ κάποιον τύπο. Ένας ορισμός για τα Success Typings παρουσιάζεται παρακάτω [Lind06],

Definition 1 (Success Typing). *Το success typing μιας συνάρτησης f είναι ένας χαρακτηρισμός τύπου, $(\bar{\alpha}) \to \beta$, έτσι ώστε οποτεδήποτε μια εφαρμογή $f(\bar{p})$ αποτιμάται σε μια τιμή v, τότε $v \in \beta$ και $\bar{p} \in \bar{\alpha}$.*

Έτσι, η ουσία ενός success typing $\bar{\alpha} \to \beta$ για μια συνάρτηση f είναι ότι αν τα ορίσματα της συνάρτησης ανήκουν στο πεδίο ορισμού της, η εφαρμογή μπορεί να επιτύχει, αλλά αν δεν ανήκουν η κλήση της συνάρτησης σίγουρα θα αποτύχει.

Ένα σημαντικό σημείο που πρέπει να αναφερθεί είναι ότι αυτό το σύστημα τύπου δεν υποστηρίζει παραμετρικό πολυμορφισμό και ως εκ τούτου δεν εξάγει καμία πληροφορία σχετικά με τη σχέση των ορισμάτων με το αποτέλεσμά. Ένα παράδειγμα φαίνεται παρακάτω:

```
1  %% A simple function that tags its argument
2  tag_1(N) when is_atom(N) -> {atom, N};
3  tag_1(N) when is_float(N) -> {float, N};
4  tag_1(N) when is_integer(N) -> {integer, N}.
5
6  %% Its inferred type would be
7  tag_1/1 :: (atom() | number()) -> {'atom', atom()}
8                                  | {'float', float()}
9                                  | {'integer', integer()}.
```

Listing 2.12: An example of a success typing that misses the relation of the arguments to the result.

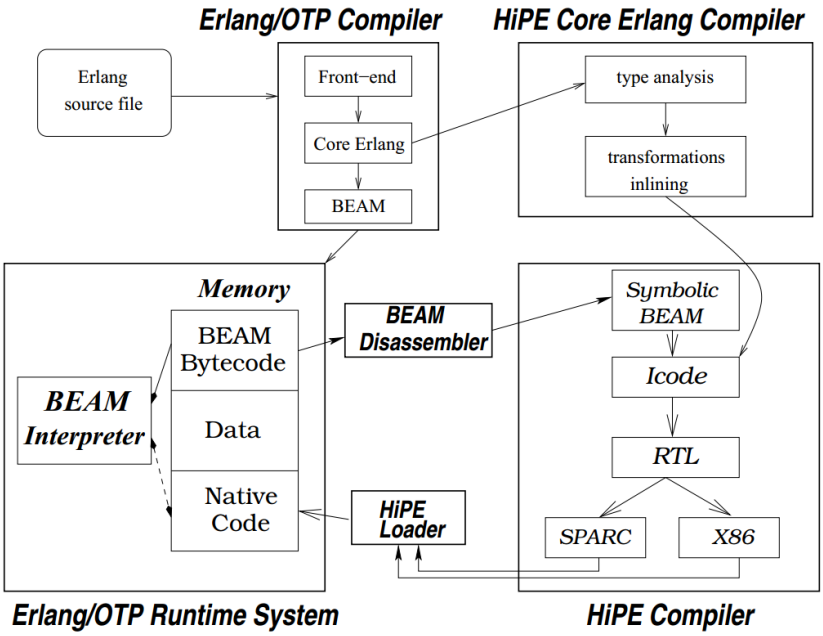
Σε αυτό το παράδειγμα το γεγονός ότι το όρισμα εισόδου είναι το ίδιο με το δεύτερο στοιχείο της πλειάδας εξόδου έχει χαθεί.

2.1.4 HiPE

Το HiPE είναι ένας μεταγλωττιστής κώδικα μηχανής για την Erlang. Προσφέρει ευέλικτη ενσωμάτωση μεταξύ bytecode και κώδικα μηχανής. Ο κύριος στόχος του είναι να επιτρέψει την μεταγλώττιση συγκεκριμένων τμημάτων της εφαρμογής, διατηρώντας ταυτόχρονα τα υπόλοιπα, προκειμένου να επιτευχθεί η καλύτερη ισορροπία μεταξύ απόδοσης και φορητότητας.

Πριν από την ανάπτυξη του HiPE όλες οι υλοποιήσεις της Erlang βασίστηκαν σε εξομοιωτές εικονικών μηχανών. Αυτό βελτίωνε τη φορητότητα αλλά επέβαλε κυρώσεις επιδόσεων. Επίσης, οι εξομοιωτές bytecode συνήθως έχουν ως αποτέλεσμα μικρότερο μέγεθος κώδικα από τους μεταγλωττιστές κώδικα μηχανής. Ενώ σε πολλούς τομείς, το μέγεθος του κώδικα αντικειμένων καθίσταται όλο και λιγότερο πρόβλημα, υπάρχουν περιοχές (όπως τα ενσωματωμένα συστήματα) όπου εξακολουθεί να αποτελεί πιθανό πρόβλημα.

Μια επισκόπηση ενός συστήματος Erlang/OTP εκτεταμένου με HiPE φαίνεται στο Σχήμα 2.5 [Sago03]. Η μικρότερη μονάδα μεταγλώττισης στο HiPE είναι ένα άρθρωμα.



Σχήμα 2.5: Δομή του συστήματος Erlang/OTP με υποστήριξη του HiPE.

Ενδιάμεσες Αναπαραστάσεις

Το HiPE έχει τέσσερις ενδιάμεσες αναπαραστάσεις κώδικα, [Joha00].

- Μια εσωτερική αναπαράσταση του BEAM bytecode.
- Μια ενδιάμεση γλώσσα υψηλότερου επιπέδου, η οποία ονομάζεται ICode.
- Μια γενική γλώσσα μεταβίβασης καταχωρητών, η οποία ονομάζεται RTL.
- Μια γλώσσα μηχανής για διάφορες αρχιτεκτονικές (όπως SPARC, x86, κ.λπ.).

Το ICode, το RTL και οι γλώσσες μηχανής αντιπροσωπεύονται εσωτερικά ως γράφοι ροής ελέγχου βασικών μπλοκ.

Το ICode βασίζεται σε μια εικονική μηχανή με καταχωρητές για την Erlang. Υποστηρίζει έναν άπειρο αριθμό καταχωρητών που χρησιμοποιούνται για την αποθήκευση ορισμάτων και προσωρινών δεδομένων. Όλες οι τιμές στο ICode είναι οι όροι της Erlang. Επιπλέον, η στοίβα κλήσεων είναι σιωπηρή και διατηρεί τους καταχωρητές. Ορισμένες τυπικές βελτιστοποιήσεις εφαρμόζονται στο ICode, όπως διάδοση αντιγραφής και σταθερών τιμών. Στη συνέχεια πραγματοποιείται αφαίρεση του νεκρού κώδικα. Το Icode μεταφράζεται σε RTL, και κατά τη διάρκεια αυτής της διαδικασίας αφαιρείται ο απρόσιτος κώδικας, καθώς μεταφράζονται σε RTL μόνο τα προσβάσιμα βασικά μπλοκ.

Η διαχείριση της στοίβας κλήσεων και η αποθήκευση και αποκατάσταση των καταχωρητών γύρω από κλήσεις γίνονται ρητή στο RTL. Στη συνέχεια εφαρμόζονται ξανά οι τυπικές βελτιστοποιήσεις που εφαρμόστηκαν στο ICode. Στη συνέχεια, ο κώδικας RTL μεταφράζεται σε κώδικα μηχανής και οι καταχωρητές αποδίδονται χρησιμοποιώντας έναν αλγόριθμο χρωματισμού γραφημάτων παρόμοιο με αυτόν του Briggs et al. [Brig94]. Τέλος, οι συμβολικές αναφορές σε atoms και συναρτήσεις αντικαθίστανται από τις πραγματικές τους τιμές στο τρέχον σύστημα, η μνήμη διατίθεται για τον κώδικα και στη συνέχεια ο κώδικας συνδέεται με το σύστημα.

Η διαίρεση σε τρεις ξεχωριστές ενδιάμεσες αναπαραστάσεις απλοποιεί τον πειραματισμό καινούργιων βελτιστοποίησεων και την ανάπτυξη του μεταγλωττιστή. Ωστόσο, πολλές βασικές βελτιστοποιήσεις, όπως η συνεχής διάδοση και αναδίπλωση, συμβαίνουν σε όλες τις ενδιάμεσες αναπαραστάσεις, δυνητικά επιβραδύνοντας τη μεταγλώττιση [Joha03].

Αλλαγές Λειτουργίας

Οι αλλαγές λειτουργίας συμβαίνουν όταν ο έλεγχος μεταβαίνει από κώδικα μηχανής σε bytecode ή αντίστροφα. Το HiPE σχεδιάστηκε με τέτοιο τρόπο ώστε να μην επιβαρύνεται η εκτέλεση εάν δεν υπάρχουν αλλαγές λειτουργίας. Εξαιτίας αυτού, οι αλλαγές λειτουργίας επιτυγχάνονται με τρόπο που επιβάρυνει την εκτέλεση, και για αυτό το λόγω είναι πολύ σημαντικό οι υπεύθυνοι ανάπτυξης εφαρμογών σε Erlang να είναι προσεκτικοί ώστε να μην γίνονται αναίτιες αλλαγές λειτουργίας κατά τη διάρκεια εκτέλεσης του προγράμματος.

Βελτιστοποιήσεις: Διάδοση Τύπων

Η Erlang είναι μια γλώσσα με δυναμικό σύστημα τύπων και αυτό γενικά επιτρέπει τον ταχύτερο πειραματισμό στα αρχικά στάδια ανάπτυξης. Ωστόσο, αυτό οδηγεί επίσης στην εισαγωγή πολλών ελέγχων τύπου κατά την εκτέλεση του κώδικα για να είναι σίγουρο ότι οι λειτουργίες εκτελούνται στους κατάλληλους τύπους δεδομένων. Ένα παράδειγμα μιας λειτουργίας που εκτελείται σε τύπους χωρίς νόημα μπορεί να είναι η διαίρεση ενός δεκαδικου από μια λίστα.

Αυτο αντιμετωπίστηκε αρχικά με την εφαρμογή ενός περάσματος τοπικής διάδοσης τύπων, το οποίο ανακαλύπτει τις πληροφορίες τύπων (για μια συνάρτηση) και διαδίδει αυτές τις πληροφορίες για την εξάλειψη των περιττών ελέγχων τύπου [Sago03]. Αυτές οι πληροφορίες τύπων χρησιμοποιούνται επίσης για τη μετατροπή κάποιων πολυμορφικών γενικών συναρτήσεων σε εξειδικευμένες που αποδίδουν καλύτερα.

Ωστόσο, αν δεν γνωρίζουμε τίποτα για τα ορίσματα των συναρτήσεων, μπορούν να ανακαλυφθούν μόνο ελάχιστες πληροφορίες τύπου. Εξαιτίας αυτού, το HiPE διαθέτει επίσης ένα στατικό αναλυτή τύπων ο οποίος επεξεργάζεται ολόκληρες αρθρώματα. Ο αναλυτής είναι σε θέση να συνάγει πιο συγκεκριμένες πληροφορίες τύπου, ειδικά όταν το άρθρωμα είναι σχετικά κλειστό στον εξωτερικό κόσμο, πράγμα που σημαίνει ότι

δεν εξάγονται πολλές συναρτήσεις. Αυτό επιτρέπει στο HiPE να κάνει αυστηρότερες υποθέσεις σχετικά με τα ορίσματα των συναρτήσεων που δεν εξάγονται, οι οποίες στη συνέχεια οδηγούν σε εξάλειψη περισσότερων ελέγχων τύπου.

Είναι σημαντικό να σημειωθεί ότι αυτός ο στατικός αναλυτής τύπων δεν είναι ένας ελεγκτής τύπων και δεν παράγει προειδοποιήσεις για κακές λειτουργίες. Επίσης, ο χρήστης δεν μπορεί να αλληλεπιδράσει με αυτόν με κανέναν τρόπο.

Βελτιστοποιήσεις: Χειρισμός Δεκαδικών

Οι ατομικές τιμές στην Erlang αντιπροσωπεύονται ως λέξεις 32 ή 64 bit με ετικέτα στο σύστημα χρόνου εκτέλεσης. Εάν μια τιμή με ετικέτα είναι υπερβολικά μεγάλη για να χωρέσει σε μία λέξη μηχανής, η τιμή αποθηκεύεται σε ένα κουτί και αποθηκεύεται στον σωρό. Εξαιτίας αυτού, οι αριθμοί κινητής υποδιαστολής συνήθως περικλείονται σε κουτιά, οπότε κάθε φορά που πρέπει να γίνει μια πράξη με τέτοιους αριθμούς, τα ορίσματα πρέπει να βγουν από τα κουτιά και μετά τα αποτελέσματα πρέπει να μπουν πάλι πίσω.

Για να αποφευχθεί αυτή η επιβάρυνση, το BEAM υποστηρίζει ειδικές οδηγίες κινητής υποδιαστολής που λειτουργούν απευθείας σε τιμές που δεν φέρουν ετικέτα, έτσι ώστε να αποφεύγονται πολλές λειτουργίες κυτιοποίησης. Ωστόσο, ο κώδικας BEAM ερμηνεύεται έτσι ώστε η αριθμητική επίπλευσης να μην εκμεταλλεύεται την FPU της αρχιτεκτονικής στόχου.

Το HiPE επεκτείνει το BEAM χαρτογραφώντας τις τιμές του κινητού σημείου στην FPU και διατηρώντας τις εκεί για όσο το δυνατόν περισσότερο, μειώνοντας τις γενικές δαπάνες των πράξεων κινητής υποδιαστολής. Σε συνδυασμό με τη διάδοση τύπων που περιγράφηκε παραπάνω, το HiPE μπορεί να χειριστεί πολύ αποτελεσματικά τις πράξεις κινητής υποδιαστολής με τον ελάχιστο αριθμό ελέγχων τύπων [Lind03].

2.2 JiT Μεταγλώττιση

Οι διαχειριζόμενες γλώσσες, όπως η Java [Arno00a] και η C# [Hejl06], υποστηρίζουν το μοντέλο "compile-once, run-anywhere" για την παραγωγή και τη διανομή του κώδικα. Αυτό επιτρέπει τη δημιουργία προγραμμάτων που είναι φορητά και μπορούν να εκτελεστούν σε οποιαδήποτε συσκευή είναι εξοπλισμένη με την αντίστοιχη εικονική μηχανή. Εξαιτίας αυτού, η μορφή του παραγόμενου προγράμματος πρέπει να είναι ανεξάρτητη από μια συγκεκριμένη αρχιτεκτονική και έτσι οι γλώσσες αυτές ερμηνεύονται συχνά πριν από την εκτέλεση του προγράμματος. Αυτό με τη σειρά του οδηγεί σε επιβάρυνση απόδοσης σε σύγκριση με τον παραγόμενο κώδικα μηχανής. Αυτή η επιβάρυνση οδήγησε τη διερεύνηση της "πάνω-στην-ώρα" (just-in-time) μεταγλώττισης (μεταγλώττιση κατά την εκτέλεση του προγράμματος), η οποία είναι ένας συνδυασμός των δύο.

Υπάρχουν δύο κύριες κατηγορίες μεταγλωττιστών JiT, χωρισμένες ανάλογα με το βασικό μπλοκ μεταγλώττισης: οι JiT μεταγλωττιστές συναρτήσεων και οι JiT μεταγλωττιστές εντοπισμού.

2.2.1 JiT Μεταγλωττιστές Συναρτήσεων

Οι JiT μεταγλωττιστές συναρτήσεων (μεθόδων) καταγράφουν συναρτήσεις κατά τη διάρκεια της εκτέλεσης και στη συνέχεια να τις μεταγλωττίζουν και να τις βελτιστοποιούν με βάση κάποιον αλγόριθμο απόφασης. Οι πρώτοι μεταγλωττιστές JiT μεταγλώττιζαν και βελτιστοποιούσαν μια συνάρτηση την πρώτη φορά που καλούνταν [Deut84, Cham91]. Ωστόσο, αυτό αποδείχθηκε αναποτελεσματικό, καθώς υπήρχαν πολλές παύσεις εκτέλεσης για την μεταγλώττιση κάθε συνάρτησης. Αυτός είναι ο λόγος για τον οποίο οι πε-

ρισσότεροι μεταγλωττιστές JIT χρησιμοποιούν ένα μοντέλο εξομοίωσης [Hans74], όπου κάθε συνάρτηση εξομειώνεται αρχικά και στη συνέχεια βελτιστοποιείται από τον μεταγλωττιστή JiT όταν κριθεί απαραίτητο.

Ένα βασικό χαρακτηριστικό των JiT μεταγλωττιστών συναρτήσεων είναι η επιλεκτική μεταγλώττιση, η οποία βασίζεται στην παρατήρηση ότι τα περισσότερα προγράμματα ξοδεύουν τη μεγάλη πλειοψηφία του χρόνου εκτέλεσης σε ένα μικρό μόνο τμήμα του κώδικά τους. Η επιλεκτική μεταγλώττιση χρησιμοποιεί καταγραφή του χρόνου εκτέλεσης για να καθορίσει ποιες συναρτήσεις παίρνουν το μεγαλύτερο μέρος του χρόνου εκτέλεσης (hot functions), για να μεταγλωττίσουν και να βελτιστοποιήσουν μόνο αυτές και να περιορίσουν την επιβάρυνση από την μεταγλώττιση περιττών συναρτήσεων [Holz94, Suga00, Cier00].

Δυστυχώς, η εύρεση των αυτών των συναρτήσεων για μεταγλώττιση απαιτεί μελλοντικές πληροφορίες εκτέλεσης του προγράμματος και δεν μπορεί να προβλεφθεί, έτσι οι περισσότερες στρατηγικές κάνουν την υπόθεση ότι οι τρέχουσες "καυτές" συναρτήσεις θα παραμείνουν καυτές στο μέλλον. Οι πιο δημοφιλείς προσεγγίσεις καταγραφής βασίζονται σε κάποια μορφή μετρητή, είτε μετρώντας τον αριθμό των κλήσεων συναρτήσεων [Hans74, Kotz08] είτε περιοδικά διακόπτοντας την εκτέλεση, ελέγχοντας ποια λειτουργία εκτελείται εκείνη την ώρα [Arno05] και ενημερώνοντας έναν μετρητή. Και οι δύο μέθοδοι θεωρούν μια συνάρτηση καυτή όταν ο μετρητής της έχει υπερβεί μια καθορισμένη τιμή κατωφλίου. Η επιλογή αυτής της τιμής κατωφλίου είναι πολύ σημαντική για την απόδοση του μεταγλωττιστή, επειδή αν ο καθορισμός της είναι πολύ χαμηλός, θα μπορούσε να οδηγήσει σε πολύ επιθετική μεταγλώττιση, η οποία θα υποβαθμίσει την απόδοση εκτέλεσης, ενώ μια πολύ ψηλή τιμή θα μπορούσε να οδηγήσει σε έναν πολύ συντηρητικό μεταγλωττιστή JiT που δεν βελτιστοποιεί καθόλου τις μεθόδους.

Ένα πολύ σημαντικό χαρακτηριστικό των JiT μεταγλωττιστών συναρτήσεων είναι η δυνατότητα χρήσης πληροφοριών χρόνου εκτέλεσης για την προσαρμογή των βελτιστοποιήσεων σε μια συγκεκριμένη εκτέλεση [Kist03, Burg97, Burg98, Gran99], η οποία έχει αποδειχθεί ότι αποφέρει σημαντικές βελτιώσεις στην απόδοση.

Μία από αυτές τις βασικές βελτιστοποιήσεις αναπτύχθηκε για την πρώτη γενιά του JiT μεταγλωττιστή της γλώσσας Self και ονομάζεται προσαρμογή (customization) [Cham89]. Η κύρια ιδέα ήταν ότι αντί της δυναμικής μεταγλώττισης μιας συνάρτησης σε κώδικα μηχανής που λειτουργεί για οποιαδήποτε κλήση, ο JiT μεταγλωττιστής θα παράγει μια εξειδικευμένη έκδοση της συνάρτησης με βάση το συγκεκριμένο πλαίσιο. Οι πληροφορίες τύπου είναι η μεγαλύτερη πηγή πληροφοριών χρόνου εκτέλεσης που συγκεντρώθηκαν και χρησιμοποιήθηκαν για την παραγωγή εξειδικευμένων εκδόσεων συναρτήσεων. Αυτή η βελτιστοποίηση ήταν ιδιαίτερα επωφελής για την Self, όπου κάθε συνάρτηση είναι δυναμική και μεταβλητή, και έτσι δεν υπάρχουν διαθέσιμες στατικές πληροφορίες για τον μεταγλωττιστή [Ayco03].

Η ενσωμάτωση συναρτήσεων βασισμένη σε δεδομένα καταγραφής (profile-driven inlining) έχει μελετηθεί ευρέως [AdlT03, Cier00, Haze03] και έχει ενσωματωθεί σε πολλούς μεταγλωττιστές JiT. Αυτό οφείλεται στο γεγονός ότι η στατική πρόβλεψη του αποτελέσματος της ενσωμάτωσης μιας κλήσης συνάρτησης είναι μια δαπανηρή υπολογιστική εργασία. Παρόλα αυτά, έχοντας πρόσβαση σε πληροφορίες χρόνου εκτέλεσης, όπως ο χρόνος εκτέλεσης που δαπανάται σε κάθε συνάρτηση και ο αριθμός των φορών που έχει κληθεί κάθε συνάρτηση από κάθε άλλη, η ενσωμάτωση θα μπορούσε να οδηγήσει στην παροχή των ίδιων πλεονεκτημάτων απόδοσης με πολύ απλούστερους αλγόριθμους απόφασης. Έχει φανεί ότι οι μηχανισμοί λήψης αποφάσεων που βασίζονται σε καταγραφή δεδομένων του χρόνου εκτέλεσης ξεπερνούν εκείνους που βασίζονται μόνο στα στατικά δεδομένα [Arno02, Suga02].

2.2.2 JiT Μεταγλωττιστές Ιχνηλάτησης

Οι JiT μεταγλωττιστές ιχνηλάτησης (Tracing JiT) καθορίζουν συχνά εκτελούμενα ίχνη (καυτά μονοπάτια) στα τρέχοντα προγράμματα και εκπέμπουν βελτιστοποιημένο κώδικα μηχανής για αυτά τα ίχνη. Ένας JiT μεταγλωττιστής ιχνηλάτησης παρατηρεί την εκτέλεση μέχρι να αναγνωριστεί μια ακολουθία εντολών ("ίχνος"). Στη συνέχεια, δημιουργεί μια βελτιστοποιημένη έκδοση αυτού του ίχνους. Οι επακόλουθες συναντήσεις της διεύθυνσης εισόδου του καυτού ίχνους κατά την ερμηνεία οδηγούν στο άλμα στην κορυφή της βελτιστοποιημένης έκδοσης του ίχνους. Όταν ο έλεγχος εξέρχεται από το βελτιστοποιημένο ίχνος, το JiT συνεχίζει να ερμηνεύει κανονικά.

Είναι σημαντικό να κατανοήσουμε πώς ένας JiT μεταγλωττιστής ιχνηλάτησης επιλέγει ποια ίχνη θα βελτιστοποιήσει. Ο κύριος συλλογισμός είναι ότι ένας Tracing JiT στοχεύει περισσότερο στην προβλεψιμότητα και λιγότερο στην ακρίβεια. Έτσι, για παράδειγμα, ένα ίχνος που είναι ζεστό για ένα σύντομο χρονικό διάστημα, αλλά συνολικά δεν συμβάλλει πολύ στον χρόνο εκτέλεσης του προγράμματος, μπορεί να είναι σημαντικό για έναν Tracing JiT μεταγλωττιστή.

Για να κατανοήσουμε πώς λειτουργεί ένας JiT μεταγλωττιστής ιχνηλάτησης, θα περιγράψουμε τη λειτουργικότητα του Dynamo [Bala00] που ήταν ένας από τους πρώτους Tracing JiT μεταγλωττιστές. Ο Dynamo διατηρεί ένα μετρητή για ορισμένα ειδικά σημεία του προγράμματος, που ονομάζονται έναρξη-ιχνηλάτησης, όπως οι διευθύνσεις-στόχοι των προς τα πίσω διακλαδώσεων που είναι πολύ πιθανό να είναι στην αρχή κάποιου βρόχου. Κάθε μετρητής αυξάνεται όταν η εκτέλεση περνά από τη διεύθυνση στόχου. Εάν ένας μετρητής υπερβεί ένα προκαθορισμένο όριο, ο Tracing JiT καταγράφει όλες τις οδηγίες που ξεκινούν από τη διεύθυνση που σχετίζεται με τον μετρητή. Η εγγραφή σταματά όταν πληρούνται ορισμένες ειδικές συνθήκες και το αποθηκευμένο ίχνος αποθηκεύεται για να βελτιστοποιηθεί αργότερα. Αυτό το σχήμα επιλογής ιχνών ονομάζεται πιο πρόσφατα εκτελεσμένη ουρά (MRET) και η διορατικότητα πίσω από αυτό είναι ότι όταν μια εντολή γίνεται "καυτή", είναι πιθανό ότι η επόμενη ακολουθία οδηγιών θα είναι επίσης "καυτή" . Επομένως, αντί να καταγράφει όλους τους κλάδους μετά την αρχή της ανίχνευσης, που θα οδηγούσε σε μια μεγάλη μνήμη, θα καταγράφει μόνο την ακολουθία των οδηγιών που ακολουθούν την αρχή του ίχνους στη συγκεκριμένη εκτέλεση.

Είναι σημαντικό να σημειώσουμε ότι το ίχνος, όντας σειριακό, αντιπροσωπεύει μόνο ένα από τα πολλά πιθανά μονοπάτια σε όλο τον κώδικα. Για να εξασφαλιστεί η ορθότητα, το ίχνος περιέχει έναν έλεγχο σε κάθε πιθανό σημείο όπου η διαδρομή θα μπορούσε να ακολουθήσει άλλη κατεύθυνση.

Σε αντίθεση με το Dynamo, οι Gal et al. αποφάσισαν να επεκτείνουν την απλή ακολουθιακή ανίχνευση με την έννοια των δέντρων ιχνών [Gal09], τα οποία υποτίθεται ότι προσφέρουν οφέλη απόδοσης καθώς βελτιώνουν την απλή ροή ελέγχου στις διαδρομές εκτέλεσης.

Στην περίπτωση του PyPy, οι Bolz et al. σχεδίασαν έναν TJiT μεταγλωττιστή που, αντί να ιχνηλατεί το πρόγραμμα, ιχνηλατεί τον ίδιο τον διερμηνέα [Bolz09], με στόχο την επέκταση του PyPy ως Tracing JiT μεταγλωττιστή πολλών δυναμικών γλωσσών, υλοποιώντας τον διερμηνέα τους σε RPython[4].

2.3 Άλλοι JiT Μεταγλωττιστές για την Erlang

Η JiT μεταγλώττιση έχει διερευνηθεί στο πλαίσιο της Erlang αρκετές φορές στο παρελθόν, τόσο παλαιότερα όσο και πιο πρόσφατα. Για παράδειγμα, τόσο ο Jerico [Joha96], ο πρώτος μεταγλωττιστής κώδικα μηχανής για την Erlang (βασισμένος στο JAM) γύρω στο 1996, όσο και οι πρώιμες εκδόσεις του HiPE περιείχαν κάποια υποστήριξη για JiT

[4] Ένα υποσύνολο της Python που υποστηρίζει το PyPy.

μεταγλώττιση. Ωστόσο, αυτή η υποστήριξη δεν έγινε ποτέ σταθερή σε σημείο που θα μπορούσε να χρησιμοποιηθεί στην παραγωγή.

Πιο πρόσφατα, έχουν γίνει δύο προσπάθειες για την ανάπτυξη TJiT μεταγλωττιστών για την Erlang, το BEAMJIT [Drej14] και το Pyrlang [Huan16].

2.3.1 BEAMJIT

Το BEAMJIT [Drej14] είναι ένα Tracing JiT σύστημα χρόνου εκτέλεσης με δυνατότητες μεταγλώττισης για τη Erlang. Το BEAMJIT χρησιμοποιεί μια στρατηγική ιχνηλάτησης για να αποφασίσει ποιες ακολουθίες κώδικα θα μεταγλωττιστούν σε κώδικα μηχανής και το LLVM [Latt04] για τη βελτιστοποίηση και την εκπομπή κωδίκα μηχανής. Επεκτείνει την κλασσική υλοποίηση του BEAM με υποστήριξη για την καταγραφή, την ιχνηλάτηση, και τη μεταγλώττιση κώδικα και την υποστήριξη της εναλλαγής μεταξύ αυτών των τριών τρόπων εκτέλεσης (καταγραφή, ιχνηλάτηση και εκτέλεση κώδικα μηχανής).

Όσον αφορά την απόδοση, το 2014, το BEAMJIT κατάφερε να μειώσει τον χρόνο εκτέλεσης ορισμένων μικρών προγραμμάτων της Erlang κατά 25–40% σε σύγκριση με το BEAM, αλλά υπήρξαν επίσης πολλά άλλα προγράμματα όπου απέδιδε χειρότερα από το BEAM [Drej14]. Επιπλέον, στην ίδια δημοσίευση αναφέρεται ότι «το HiPE παρέχει μια τόσο μεγάλη βελτίωση επιδόσεων σε σύγκριση με το BEAM που η σύγκριση με το BEAMJIT θα ήταν μη ενδιαφέρουσα» [Drej14, Ενότητα 5]. Τη χρονική στιγμή που γράφεται αυτή η διπλωματική (Μάιος 2018), το BEAMJIT δεν είναι ακόμη μια πλήρης εφαρμογή υλοποίηση της Erlang. Για παράδειγμα, δεν υποστηρίζει ακόμα δεκαδικούς αριθμούς. Παρόλο, που οι εργασίες συνεχίζονται για την επέκταση του BEAMJIT, η συνολική του απόδοση, η οποία έχει βελτιωθεί, δεν ξεπερνά ακόμα την απόδοση του HiPE.[5] Το BEAMJIT δεν είναι διαθέσιμο και για αυτόν το λόγο δε μπορέσαμε να συγκρίνουμε τον μεταγλωττιστή μας μαζί του.

2.3.2 Pyrlang

Η δεύτερη προσπάθεια, το Pyrlang [Huan16], είναι μια εναλλακτική εικονική μηχανή για το bytecode του BEAM, ο οποίος χρησιμοποιεί τον Tracing JiT μεταγλωττιστή της RPython [Bolz09] για να βελτιώσει τη σειριακή απόδοση των προγραμμάτων Erlang. Αυτός ο Tracing JiT μεταγλωττιστής εντοπίζει και βελτιστοποιεί το διερμηνέα αντί το ίδιο το προγράμμα. Η ίδια ιδέα έχει προηγουμένως εφαρμοστεί και στη γλώσσα Racket, με τη μορφή του Pyket [Baum15]. Στην δημοσίευση του Pyrlang [Huan16] αναφέρεται ότι το Pyrlang επιτυγχάνει μέση απόδοση η οποία είναι 38.3% γρηγορότερη από το BEAM και 25.2% βραδύτερη από την HiPE, σε μια ομάδα μικρών σειριακών προγραμμάτων. Επί του παρόντος, το Pyrlang είναι ένα ερευνητικό πρωτότυπο και δεν έχει ακόμη ολοκληρωθεί η υλοποίησή του, ούτε και για το σειριακό κομμάτι της γλώσσας. Από την άλλη πλευρά, σε αντίθεση με το BEAMJIT, το Pyrlang είναι διαθέσιμο και θα συγκρίνουμε το JIT μεταγλωττιστή μας άμεσα με αυτό στο Κεφάλαιο 5.

[5] Lukas Larsson, ιδιωτική επικοινωνία, Μάιος 2018.

Κεφάλαιο 3

HiPErJiT

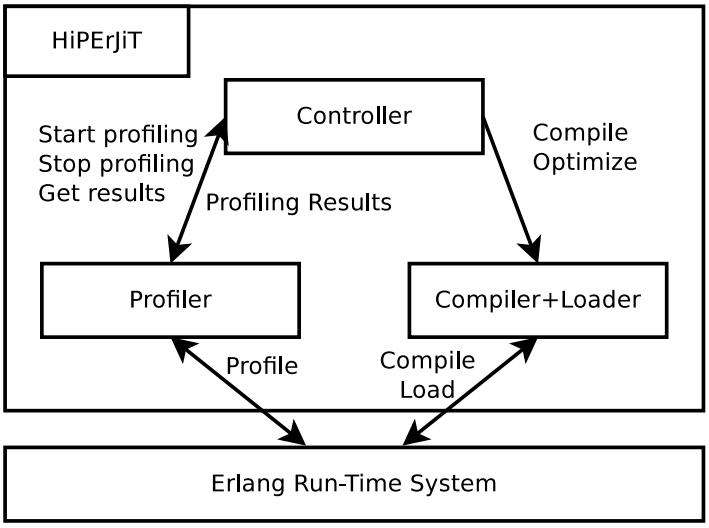
Η λειτουργικότητα του HiPErJiT μπορεί να περιγραφεί σύντομα ως εξής. Καταγράφει μονάδες κώδικα μου εκτελούνται, διατηρώντας δεδομένα χρόνου εκτέλεσης, όπως το χρόνο εκτέλεσης και γράφους κλήσεων. Στη συνέχεια αποφασίζει ποιες μονάδες θα μεταγλωττιστούν και θα βελτιστοποιηθούν με βάση τα δεδομένα που συλλέχθηκαν. Τέλος, μεταγλωττίζει και φορτώνει τις JiT-μεταγλωττισμένες μονάδες στο σύστημα χρόνου εκτέλεσης. Κάθε μία από αυτές τις εργασίες αντιμετωπίζεται από ένα ξεχωριστό τμήμα του HiPErJiT.

Ελεγκτής (Controller) Ο ελεγκτής, είναι η κεντρική μονάδα ελέγχου που αποφασίζει ποιες μονάδες θα πρέπει να καταγραφούν και ποιες θα πρέπει να βελτιστοποιηθούν με βάση τα δεδομένα που συλλέγονται κατά το χρόνο εκτέλεσης.

Καταγραφέας (Profiler) Ο καταγραφέας είναι ένα ενδιάμεσο στρώμα μεταξύ του ελεγκτή και των καταγραφέων του ERTS. Συλλέγει πληροφορίες σχετικά κατά τη διάρκεια της εκτέλεσης των προγραμμάτων, την οργανώνει και τη στέλνει πίσω στον ελεγκτή για περαιτέρω επεξεργασία.

Μεταγλωττιστής+Φορτωτής (Compiler+Loader) Ο μεταγλωττιστής+φορτωτής είναι ένα ενδιάμεσο στρώμα μεταξύ του ελεγκτή και του HiPE. Μεταγλωττίζει τις μονάδες που επιλέγει ο ελεγκτής και στη συνέχεια τις φορτώνει στο σύστημα χρόνου εκτέλεσης.

Η αρχιτεκτονική του μεταγλωττιστή HiPErJiT φαίνεται στο Σχήμα 3.1.



Σχήμα 3.1: Η αρχιτεκτονική του HiPErJiT.

3.1 Ελεγκτής

Ο ελεγκτής, όπως αναφέρθηκε παραπάνω, είναι η θεμελιώδης μονάδα του HiPErJiT. Επιλέγει τις μονάδες που καταγράφονται και χρησιμοποιεί τα δεδομένα καταγραφής για να αποφασίσει ποιες μονάδες θα μεταγλωττιστούν και θα βελτιστοποιηθούν. Είναι σημαντικό να μην απαιτείται καμία αλληλεπίδραση με τον χρήστη για τη λήψη αποφάσεων. Ο σχεδιασμός μας επεκτείνει πολλές ιδέες από το Jalapeño JVM [Arno00b].

Παραδοσιακά, πολλοί JiT μεταγλωττιστές χρησιμοποιούν μια μέθοδο συχνότητας κλήσεων για να οδηγήσουν τη μεταγλώττιση [Ayco03]. Αυτή η μέθοδος διατηρεί ένα μετρητή για κάθε συνάρτηση και τον αυξάνει κάθε φορά που καλείται η συνάρτηση. Όταν ο μετρητής ξεπεράσει ένα καθορισμένο όριο, ενεργοποιείται η JiT μεταγλώττιση. Η προσέγγιση αυτή, αν και έχει πολύ χαμηλό υπολογιστικό κόστος, δεν παρέχει ακριβείς πληροφορίες σχετικά με την εκτέλεση του προγράμματος.

Αντ' αυτού, ο HiPErJiT λαμβάνει την απόφασή του βάσει μιας απλής ανάλυσης κόστους-οφέλους. Η μεταγλώττιση για μια ενότητα ενεργοποιείται όταν ο προβλεπόμενος μελλοντικός χρόνος εκτέλεσης για αυτή, όντας μεταγλωττισμένη, σε συνδυασμό με τον χρόνο μεταγλώττισης, θα είναι μικρότερος από τον μελλοντικό χρόνο εκτέλεσης όταν ερμηνεύεται:

$$FutureExecTime_c + CompTime < FutureExecTime_i$$

Φυσικά, δεν είναι δυνατόν να προβλεφθεί επακριβώς ο μελλοντικός χρόνος εκτέλεσης μιας μονάδας ή ο χρόνος που απαιτείται για τη μεταγλώττιση της, έτσι πρέπει να γίνουν κάποιες εκτιμήσεις. Πρώτα από όλα, πρέπει να υπολογίσουμε τη μελλοντική διάρκεια εκτέλεσης. Τα αποτελέσματα μιας μελέτης σχετικά με τη διάρκεια ζωής διεργασιών UNIX [Harc97] δείχνουν ότι ο συνολικός χρόνος εκτέλεσης των διεργασιών UNIX ακολουθεί μια κατανομή Pareto (heavytail). Ο μέσος χρόνος αναμονής αυτής της διανομής είναι ανάλογος με τον χρόνο που έχει περάσει ήδη. Έτσι, αν υποτεθεί ότι ισχύει η αναλογία μεταξύ μιας μονάδας και μιας διεργασίας UNIX, θεωρούμε ότι ο μελλοντικός χρόνος εκτέλεσης μιας μονάδας είναι ίσος με τον χρόνο εκτέλεσης της μέχρι τώρα $FutureExecTime = ExecTime$. Επιπλέον, θεωρούμε ότι ο μεταγλωττισμένος κώδικας έχει σταθερή αύξηση ταχύτητας προς τον ερμηνευμένο κώδικα, επομένως $ExecTime_c * Speedup_c = ExecTime_i$. Τέλος, θεωρούμε ότι ο χρόνος σύνταξης για μια ενότητα εξαρτάται γραμμικά από το μέγεθός της, έτσι $CompTime = C * Size$. Με λίγα λόγια, η συνθήκη που ελέγχεται για τη μεταγλώττιση μιας μονάδας είναι η εξής:

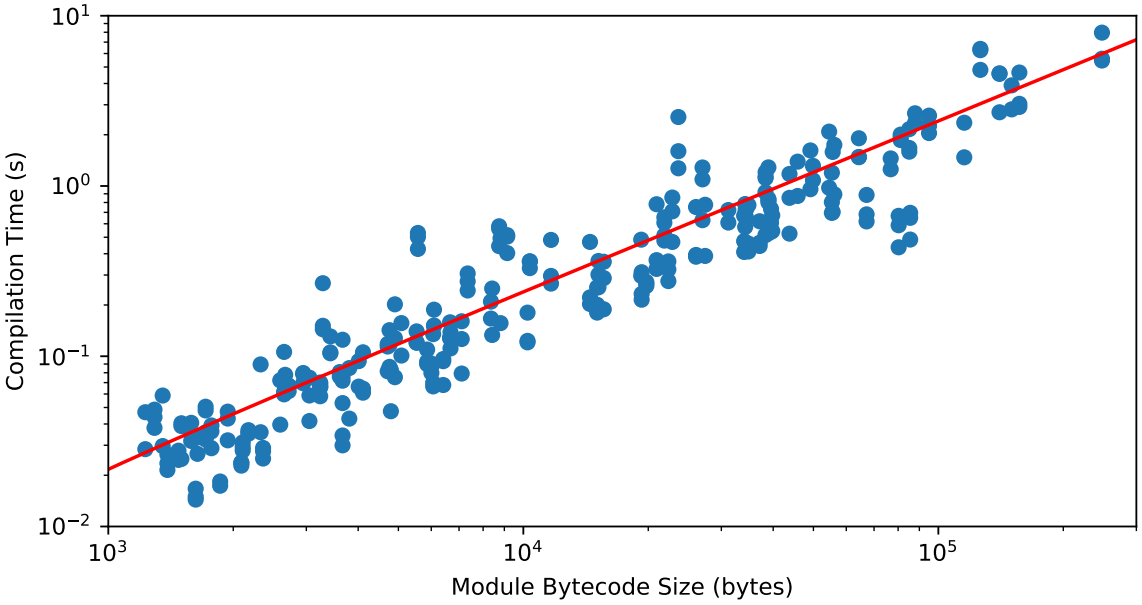
$$\frac{ExecTime_i}{Speedup_c} + C * Size < ExecTime_i$$

Εάν αυτή η συνθήκη ισχύει, η ενότητα αξίζει να μεταγλωττιστεί.

Πραγματοποιήσαμε δύο πειράματα για να βρούμε τις κατάλληλες τιμές για τις παραμέτρους $Speedup_c$ και C. Στο πρώτο, εκτελέσαμε ένα σύνολο εφαρμογών πριν και μετά τη μεταγλώττιση των ενοτήτων τους σε εγγενή κώδικα. Η μέση επιτάχυνση που μετρήσαμε ήταν 2 και την χρησιμοποιήσαμε ως εκτίμηση για την παράμετρο $Speedup_c$. Στο δεύτερο πείραμα, μεταγλωττίσαμε ένα σύνολο μονάδων διαφόρων μεγεθών, μετρήσαμε τον χρόνο μεταγλώττισης τους και βρήκαμε τη βέλτιστη γραμμή που περιγράφει αυτά τα δεδομένα χρησιμοποιώντας την μέθοδο των ελάχιστων τετραγώνων (βλ. Σχήμα 3.2).

Η γραμμή έχει κλίση $2.5e^{-5}$. Αξίζει να σημειωθεί ότι, στην πραγματικότητα, ο χρόνος μεταγλώττισης δεν εξαρτάται μόνο από το μέγεθος της μονάδας, αλλά από πολλούς άλλους παράγοντες (π.χ. τον αριθμό των διακλάδωσεων, το πόσες είναι οι εξαγόμενες συναρτήσεις, κ.λπ.). Ωστόσο, θεωρούμε ότι αυτή η εκτίμηση είναι επαρκής για τους σκοπούς μας.

Τέλος, το HiPErJiT χρησιμοποιεί ένα σχήμα που ανατροφοδότησης για να βελτιώσει την ακρίβεια της απόφασης για μεταγλώττιση όταν είναι δυνατόν. Ο HiPErJiT μετράει



Σχήμα 3.2: Χρόνοι μεταγλώττισης των μονάδων σε σχέση με το μέγεθος τους.

τον χρόνο μεταγλώττισης κάθε μονάδας που μεταγλωττίζεται σε κώδικα μηχανής και τον αποθηκεύει σε μια μόνιμη αποθήκη με ζεύγη κλειδιού-τιμής, όπου το κλειδί είναι μια πλειάδα του ονόματος της μονάδας και της τιμής κατακερματισμού MD5 του bytecode της. Εάν η ίδια έκδοση της εν λόγω μονάδας ζητηθεί για τη μεταγλώττιση κάποια επόμενη φορά, ο HiPErJiT θα χρησιμοποιήσει την αποθηκευμένη μέτρηση χρόνου σύνταξης στη θέση $CompTime$.

3.2 Καταγραφέας

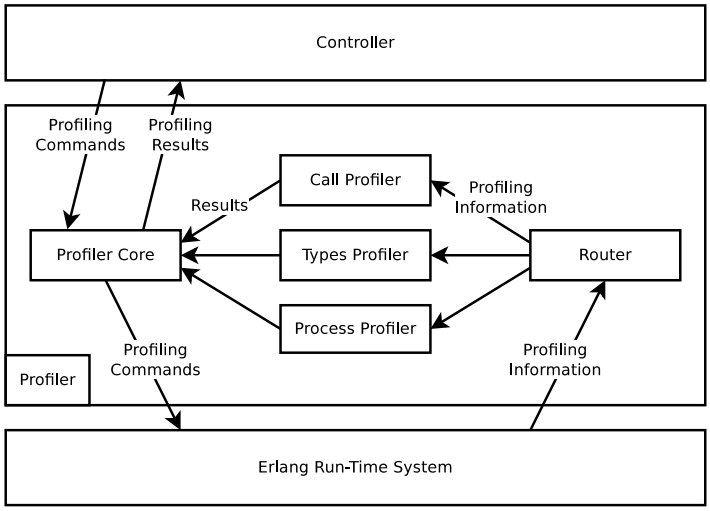
Ο καταγραφέας είναι υπεύθυνος για την αποτελεσματική καταγραφή του εκτελεσθέντος κώδικα. Η αρχιτεκτονική του, η οποία φαίνεται στο Σχήμα 3.3, αποτελείται από:

- Τον πυρήνα του καταγραφέα, ο οποίος λαμβάνει τις εντολές καταγραφής από τον ελεγκτή και τις μεταφέρει στο ERTS. Λαμβάνει επίσης τα αποτελέσματα καταγραφής από τους μεμονωμένους καταγραφείς και τα μεταφέρει πίσω στον ελεγκτή.
- Το δρομολογητή, ο οποίος λαμβάνει όλα τα μηνύματα καταγραφής από το ERTS και τα δρομολογεί στον σωστό καταγραφέα.
- Τους μεμονωμένους καταγραφείς, που χειρίζονται τα μηνύματα καταγραφής, τα συγκεντρώνουν και μεταφέρουν τα αποτελέσματα στον πυρήνα του καταγραφέα. Κάθε μεμονωμένος καταγραφέας επεξεργάζεται ένα διαφορετικό υποσύνολο των δεδομένων εκτέλεσης, δηλαδή είτε κλήσεις συναρτήσεων, είτε χρόνο εκτέλεσης, είτε τύπους παραμέτρων ή διάρκεια ζωής των διεργασιών.

Σχεδιάσαμε τον καταγραφέα με τρόπο που διευκολύνει την προσθήκη και την αφαίρεση μεμονωμένων καταγραφέων.

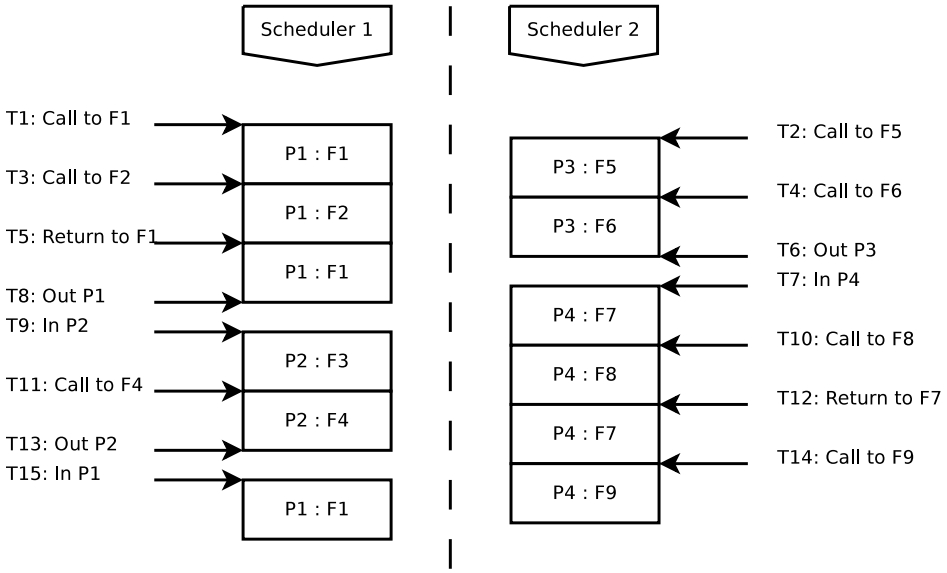
3.2.1 Καταγραφή Χρόνου Εκτέλεσης

Χρησιμοποιήσαμε αρχικά τη λειτουργία καταγραφής χρόνου εκτέλεσης του ERTS. Ωστόσο, λόγω του υψηλού υπολογιστικού κόστους, αποφασίσαμε να καταγράψουμε τον



Σχήμα 3.3: Αρχιτεκτονική του καταγραφέα και τα τμήματα του.

χρόνο εκτέλεσης χρησιμοποιώντας μια διαφορετική μέθοδο, καταγράφοντας όλες τις κλήσεις συναρτήσεων, τις επιστροφές συναρτήσεων και τις ενέργειες χρονοδρομολόγησης κάθε διεργασίας. Για να εξηγήσουμε καλύτερα τη μέθοδο μας, ένα παράδειγμα εκτέλεσης δύο χρονοδρομολογητών φαίνεται στο Σχήμα 3.4.



Σχήμα 3.4: Τα μηνύματα ιχνών που αποστέλλονται κατά τη διάρκεια μιας περιόδου εκτέλεσης δύο χρονοδρομολογητών.

Κάθε ορθογώνιο απεικονίζει ένα χρονικό κομμάτι στον χρονοδρομολογητή και αναγνωρίζεται από τη διεργασία και τη συνάρτηση που εκτελείται. Τα Tis είναι χρονοσφραγίδες, τα Pis είναι αναγνωριστικά διεργασίων, και τα Fis είναι ονόματα συναρτήσεων. Τα In και Out σημαίνουν ότι μια διεργασία αρχίζει ή σταματάει να εκτελείται εκείνη τη στιγμή. Κάθε βέλος αντιπροσωπεύει ένα μήνυμα το οποίο ο καταγραφέας λαμβάνει από το ERTS. Τα μηνύματα έχουν τη μορφή: $\langle Action, Timestamp, Process, Function\rangle$, όπου $Action$ μπορεί να είναι οποιαδήποτε από τις ακόλουθες τιμές: *call*, *return_to*, *sched_in* και *sched_out*.

Για κάθε ακολουθία μηνυμάτων που αναφέρονται στην ίδια διαδικασία P, δηλαδή της μορφής $[(\textit{sched\_in}, T_1, P, F_1), (A_2, T_2, P, F_2), \ldots, (A_{n-1}, T_{n-1}, P, F_{n-1}), (\textit{sched\_out}, T_n, P, F_n)]$,

μπορούμε να υπολογίσουμε το χρόνο που δαπανάται σε κάθε συνάρτηση βρίσκοντας τη διαφορά των χρονοσφραγίδων σε κάθε ζεύγος παρακείμενων μηνυμάτων, $[(T_2 - T_1, F_1), (T_3 - T_2, F_2), \ldots, (T_n - T_{n-1}, F_{n-1})]$. Το άθροισμα όλων των διαφορών για κάθε συνάρτηση δίνει το συνολικό χρόνο εκτέλεσης που δαπανάται σε κάθε συνάρτηση.

Η παραπάνω μέθοδος είναι λιγότερο απαιτητική και δίνει εξίσου ακριβή αποτελέσματα. Επίσης, μας επέτρεψε να παρακολουθήσουμε τον αριθμό των κλήσεων μεταξύ ενός ζεύγους συναρτήσεων, το οποίο χρησιμοποιείται αργότερα για την βελτιστοποίηση ενσωμάτωσης (βλ. Ενότητα 4.2). Ωστόσο, στα παράλληλα προγράμματα, ο ρυθμός με τον οποίο αποστέλλονται μηνύματα καταγραφής μπορεί να αυξηθεί ανεξέλεγκτα, κατακλύζοντας έτσι το γραμματοκιβώτιό του καταγραφέα, οδηγώντας σε υψηλή κατανάλωση μνήμης και προβλήματα ανταπόκρισης.

Αυτό το πρόβλημα είναι στην πραγματικότητα παρόμοιο με το πρόβλημα της ανταπόκρισης των συστημάτων ροής δεδομένων και έχει μελετηθεί αρκετά [Reis05, Babc04]. Τα συστήματα ροής δεδομένων συχνά πρέπει να αντιμετωπίζουν ριπές δεδομένων που δεν μπορούν να αντιμετωπιστούν από τους διαθέσιμους πόρους του συστήματος. Συνήθως το επιτυγχάνουν με μια μορφή προσαρμοστικής μείωσης φόρτου που πετάει δεδομένα πριν εισέλθουν στο σύστημα.

Με βάση τα παραπάνω, αποφασίσαμε να απορρίψουμε τα μηνύματα καταγραφής αν το φορτίο είναι πολύ υψηλό για να διατηρηθεί η ανταπόκριση σε ένα σταθερό επίπεδο. Επεκτείναμε τον καταγραφέα έτσι ώστε κάθε R μηνύματα ελέγχει ολόκληρο το γραμματοκιβώτιο για να βρει ένα αίτημα για αποτελέσματα από τον ελεγκτή. Αν δεν βρεθεί συνεχίζει να χειρίζεται μηνύματα καταγραφής. Χρησιμοποιώντας αυτήν την τεχνική καταφέραμε να έχουμε ένα "ανταποκρίσιμο" καταγραφέα ακόμη και όταν του στέλνονταν περισσότερα μηνύματα από αυτά που μπορούσε να χειριστεί. Όλα τα μη επεξεργασμένα μηνύματα στο γραμματοκιβώτιο μετά το αίτημα για αποτελέσματα αποσύρονται. Αυτό προφανώς οδηγεί σε απώλεια πληροφοριας, αλλά αυτό δεν πρέπει να μας αποθαρρύνει, καθώς ο μεταγλωττιστής JIT χρειάζεται τα αποτελέσματα της καταγραφής χρόνου εκτέλεσης για να αποφασίσει ποιές μονάδες θα μεταγλωττίσει και επομένως οι επιδόσεις είναι προτιμότερες από την ακρίβεια.

Μετά από αυτές τις βελτιώσεις, η επιβάρυνση καταγραφής ήταν χαμηλότερη από τον κανονικό μηχανισμό εντοπισμού χρόνου εκτέλεσης, τα αποτελέσματα ανίχνευσης ήταν επαρκώς ακριβή και το σύστημα ανταποκρινόταν ικανοποιητικά σε κυρίως σειριακά προγράμματα. Ωστόσο, αυτό δεν ίσχυε και για τα πιο περίπλοκα προγράμματα πολλαπλών διεργασιών και αυτό μας οδήγησε στην ακόλουθη υλοποίηση.

Το σύστημά μας έχει μια πολύ σημαντική διαφορά με τα συστήματα ροής δεδομένων που μελετώνται στη βιβλιογραφία. Η διαφορά είναι ότι στην περίπτωσή μας ο παραγωγός δεδομένων (το ERTS) είναι μέρος του συστήματος και επίσης ότι επιβαρύνει την απόδοση του συστήματος. Έτσι, ενώ η απόρριψη φορτίου καθιστά το σύστημά μας πιο αποδοτικό, δε μειώνει την επιβάρυνση που επιβάλλει ο παραγωγός δεδομένων, ο οποίος εξακολουθεί να δημιουργεί και να αποστέλλει μηνύματα καταγραφής τα οποία τελικά θα απορριφθούν. Ως αποτέλεσμα, τροποποιήσαμε ελαφρώς τον μηχανισμό μας, ώστε να σταματά εντελώς τον παραγωγό, αντί να απλώς απορρίπτει τα μηνύματά του. Εισήγαμε έναν μηχανισμό ανίχνευσης που ελέγχει εάν το γραμματοκιβώτιο υπερβαίνει κάποιο μέγιστο μέγεθος (πράγμα που σημαίνει ότι περισσότερα μηνύματα αποστέλλονται στο σύστημά μας από αυτά που μπορεί να διαχειριστεί) και αν αυτό ισχύει, σταματάει την αποστολή μηνυμάτων καταγραφής.

Αυτή η αλλαγή προσέφερε μεγάλα οφέλη απόδοσης, τόσο στον χρόνο εκτέλεσης όσο και στη χρήση της μνήμης. Το όφελος χρόνου εκτέλεσης των συστημάτων που συζητούνται σε αυτήν την ενότητα παρουσιάζεται στο Σχήμα 3.5. Σημειώστε ότι οι χρόνοι εκτέλεσης για τον εντοπισμό του προσαρμοσμένου χρόνου στο dialyzer_plt και στο orbit_par_seq δεν εμφανίζονται επειδή αυτές οι εκτελέσεις εξάντλησαν τη διαθέσιμη μνήμη και συνετρίβη-

Σχήμα 3.5: Ο χρόνος εκτέλεσης ορισμένων προγραμμάτων με διαφορετικές μεθόδους καταγραφής χρόνου εκτέλεσης.

σαν. Αυτό οφείλεται κυρίως στο γεγονός ότι αυτά τα προγράμματα δημιουργούν πολλές διεργασίες και στη συνέχεια υπερχειλίζουν τον καταγραφέα με μηνύματα.

Τα αποτελέσματα καταγραφής της μεθόδου μας είναι παρόμοια (αλλά όχι ίσα) με αυτά που αποκτήθηκαν από τον τυπικό μηχανισμό καταγραφής χρόνου εκτέλεσης. Μικρές διαφορές μπορούν να αποδοθούν στους ακόλουθους λόγους:

- Δεν είναι δυνατό να καταγραφούν ταυτόχρονα τόσο οι τοπικές όσο και οι απομακρυσμένες κλήσεις με τον μηχανισμό καταγραφής, έτσι εντοπίζονται μόνο τοπικές κλήσεις. Αυτό οδηγεί σε ανακρίβειες, ειδικά σε περιπτώσεις μεγάλων αλυσίδων απομακρυσμένων κλήσεων, καθώς η μέθοδος μας θεωρεί ότι όλο αυτό το χρονικό διάστημα δαπανάται στην τελευταία τοπική κλήση συνάρτησης.

- Η άλλη πηγή ανακρίβειας είναι ότι ο καταγραφέας παύει περιοδικά, για να επβαρύνει λιγότερο το χρόνο εκτέλεσης, οδηγώντας έτσι σε εσφαλμένες υπολογισμούς της κατάστασης εκτέλεσης. Αυτό είναι ιδιαίτερα εμφανές όταν ο αριθμός των διεργασιών είναι πολύ μεγαλύτερος από τον αριθμό των χρονοδρομολογητών.

Η μείωση της επιβάρυνσης από τον μηχανισμό μας δικαιολογεί τη χρήση του στο πλαίσιο του HiPErJiT παρά τις μικρές ανακρίβειες στα αποτελέσματα καταγραφής.

Μελλοντικά, θα θέλαμε να βελτιώσουμε τη μέθοδο μας έτσι ώστε να γίνει τόσο σταθερή και ακριβής όσο η τυποποιημένη. Προκειμένου να επιτευχθεί αυτό, σχεδιάζουμε να την επεκτείνουμε ώστε να εντοπίσουμε ταυτόχρονα τοπικές και εξωτερικές κλήσεις. Επιπλέον, θα εργαστούμε για την ακρίβεια των αποτελεσμάτων, διασφαλίζοντας ότι η κατάσταση καταγραφής είναι συνεπής κατά την παύση και την επανεκκίνηση.

3.2.2 Καταγραφή Τύπων

Ο καταγραφέας είναι επίσης υπεύθυνος για καταγραφή τύπων. Όπως θα δούμε, το HiPErJiT περιέχει ένα πέρασμα μεταγλωττιστή που χρησιμοποιεί πληροφορίες τύπων για τα ορίσματα των συναρτήσεων προκειμένου να δημιουργήσει εξειδικευμένες εκδόσεις συναρτήσεων. Αυτές οι πληροφορίες τύπου εξάγονται από τις προδιαγραφές τύπου στον πηγαίο κώδικα Erlang (αν υπάρχουν) και από τις πληροφορίες τύπων που συλλέγονται και συγκεντρώνονται όπως περιγράφεται παρακάτω κατά την εκτέλεση των προγραμμάτων.

Για τις μονάδες που ο καταγραφέας έχει επιλέξει για καταγραφή, καταγράφονται για κάποιο χρονικό διάστημα και τα ορίσματα των κλήσεων συναρτήσεων τους. Η λειτουργικότητα καταγραφής του ERTS επιστρέφει τις πλήρεις τιμές των ορισμάτων, έτσι μπορούν να προσδιοριστούν οι τύποι τους. Ωστόσο, οι συναρτήσεις μπορούν να κληθούν με περίπλοκες δομές δεδομένων ως ορίσματα και ο καθορισμός των τύπων τους απαιτεί την πλήρη διάσχιση τους. Καθώς αυτό θα μπορούσε να οδηγήσει σε σημαντική επιβάρυνση της απόδοσης, ο καταγραφέας προσεγγίζει τους τύπους τους με τη χρήση της μεθόδου του περιορισμένου ορίου βάθους. Με άλλα λόγια, χρησιμοποιείται το *depth-k abstraction type*. Κάθε τύπος T αντιπροσωπεύεται ως δέντρο με φύλλα που είναι είτε μεμονωμένοι τύποι (singletons) και εσωτερικοί κόμβοι που είναι κατασκευαστές τύπων με ένα ή περισσότερα ορίσματα. Το βάθος κάθε κόμβου ορίζεται ως η απόσταση από τη κορυφή του δέντρου. Ένας βάθους-k τύπος T_k είναι ένα δέντρο όπου κάθε κόμβος με βάθος $\geq k$ είναι κλαδευμένος και υπερ-προσεγγισμένος με τον τύπο κορυφής (`any()`). Για παράδειγμα, στη γλώσσα τύπων της Erlang [Lind05], η οποία υποστηρίζει ενώσεις singleton τύπων, ο όρος Erlang `{foo,{bar,[{a,17},{a,42}]}}` έχει τύπο `{'foo',{'bar',list({'a',17|42})}}`, όπου `'A'` δηλώνει το μεμονωμένο τύπο ατόμου `A`. Οι βάθους-1 και βάθους-2 τύποι του παραπάνω όρου είναι `{'foo',{any(),any()}}` και `{'foo',{'bar',list(any())}}`.

Οι ακόλουθες δύο ιδιότητες πρέπει να ισχύουν για κάθε βάθους-k τύπο:

1. $\forall k.k \geq 0 \Rightarrow T \sqsubseteq T_k$ όπου $\sqsubseteq$ είναι η σχέση υποτύπου

2. $\forall t.t \neq T \Rightarrow \exists k.\forall i.i \geq k \Rightarrow t \not\sqsubseteq T_i$

Η πρώτη ιδιότητα εγγυάται την ορθότητα ενώ η δεύτερη μας επιτρέπει να βελτιώσουμε την ακρίβεια προσέγγισης επιλέγοντας ένα μεγαλύτερο k, επιτρέποντάς μας έτσι να ανταλλάξουμε επιδόσεις με ακρίβεια ή αντίστροφα.

Ένα άλλο πρόβλημα που αντιμετωπίσαμε είναι ότι οι συλλογές της Erlang μπορούν να περιέχουν στοιχεία διαφορετικού τύπου. Η διάσχιση τους για να βρεθεί ο πλήρης τύπος τους θα μπορούσε επίσης να δημιουργήσει ανεπιθύμητη επιβάρυνση. Ως αποτέλεσμα, αποφασίσαμε να εκτιμήσουμε αισιόδοξα τους τύπους στοιχείων των συλλογών. Αν και οι συλλογές Erlang μπορούν να περιέχουν στοιχεία πολλών διαφορετικών τύπων, αυτό συμβαίνει σπάνια στην πράξη. Στα περισσότερα προγράμματα, οι συλλογές συνήθως περιέχουν στοιχεία του ίδιου τύπου. Αυτό, σε συνδυασμό με το γεγονός ότι το HiPErJiT χρησιμοποιεί τις πληροφορίες καταγραφής τύπων για να εξειδικεύσει κάποιες συναρτήσεις για τους τύπους συγκεκριμένων ορισμάτων, μας δίνει την ευκαιρία να είμαστε πιο αισιόδοξοι εκτιμώντας τους τύπους τιμών. Ως εκ τούτου, αποφασίσαμε να προσεγγίσουμε τους τύπους συλλογών (επί του παρόντος λίστες και χάρτες) εξετάζοντας μόνο ένα μικρό υποσύνολο των στοιχείων τους.

Λαμβάνοντας υπόψη όλα τα παραπάνω, ο καταγραφέας τύπων είναι σε θέση να εκτιμήσει αποτελεσματικά τους τύπους τιμών Erlang. Ένα σημείο που αξίζει να αναφερθεί είναι ότι αρχικά εφαρμόσαμε αυτή τη λειτουργικότητα βάσει του τυπικού μηχανισμού καταγραφής του ERTS που καταγράφει τα ορίσματα και τις τιμές επιστροφής κάθε κλήσης συνάρτησης. Αυτό είχε αρνητικό αντίκτυπο στην απόδοση επειδή, όπως εξηγήθηκε

και στην Ενότητα 2.1.1, τα μηνύματα στην Erlang αντιγράφονται, οπότε στην πραγματικότητα όλα τα ορίσματα κλήσεων συναρτήσεων αντιγράφονται καθώς εκτελείται το πρόγραμμα, οδηγώντας σε επιβάρυνση της μνήμης και του χρόνου εκτέλεσης. Αντιμετώσαμε αυτό το πρόβλημα υλοποιώντας αυτή τη λειτουργία ως Erlang NIF (Native Implemented Function) σε C, πριν από την αποστολή μηνυμάτων καταγραφής, αποφεύγοντας έτσι την άσκοπη αντιγραφή τιμών.

Αυτό που απομένει είναι ένας τρόπος να γενικεύονται και να συγκεντρώνονται οι πληροφορίες τύπου που αποκτήθηκαν μέσω της καταγραφής πολλών κλήσεων συναρτήσεων. Όπως περιγράφεται στην Ενότητα 2.1.3, οι τύποι στην Erlang αντιπροσωπεύονται εσωτερικά χρησιμοποιώντας ένα σύστημα υποτύπων, σχηματίζοντας έτσι ένα πλέγμα τύπων. Εξαιτίας αυτού, υπάρχει μια λειτουργία supremum που μπορεί να χρησιμοποιηθεί για τη συγκέντρωση πληροφοριών τύπου που έχουν αποκτηθεί μέσω διαφορετικών καταγραφών.

3.2.3 Καταγράφωντας την Περίοδο Ζωής των Διεργασιών

Έχουμε καταβάλει σημαντική προσπάθεια για να διασφαλίσουμε ότι η επιβάρυνση του HiPErJiT δεν αυξάνεται με τον αριθμό των παράλληλων διεργασιών. Αρχικά, η απόδοση ήταν κακή κατά την εκτέλεση ταυτόχρονων εφαρμογών με μεγάλο αριθμό ($\gg 100$) διεργασιών. Κατά τη καταγραφή μιας διεργασίας, κάθε κλήση συνάρτησης οδηγεί στην αποστολή ενός μηνύματος στον καταγραφέα. Το πρόβλημα προκύπτει όταν πολλές διεργασίες εκτελούνται ταυτόχρονα, όπου απαιτεί πολύ χρόνο εκτέλεσης από τη διεργασία του καταγραφέα για να χειριστεί όλα τα μηνύματα που αποστέλλονται σε αυτήν. Επιπλέον, η κατανάλωση μνήμης του καταγραφέα εκτοξεύεται όταν τα μηνύματα φτάνουν με υψηλότερο ρυθμό από αυτόν με τον οποίο καταναλώνονται.

Για να αποφύγουμε τέτοια προβλήματα, το HiPErJiT χρησιμοποιεί τη μέθοδο της *πιθανοτικής καταγραφής*. Η ιδέα βασίζεται στην ακόλουθη παρατήρηση. Οι μαζικά ταυτόχρονες εφαρμογές συνήθως αποτελούνται από πολλές διεργασίες αδέρφια που έχουν την ίδια λειτουργικότητα. Εξαιτίας αυτού, είναι λογικό να γίνεται δειγματοληψία των εκατοντάδων (ή ακόμη και χιλιάδων) διεργασιών και να καταγράφονται μόνο αυτές για να λαμβάνονται πληροφορίες για ολόκληρο το σύστημα.

Το δείγμα αυτό όμως δεν πρέπει να λαμβάνεται τυχαία, αλλά πρέπει να ακολουθεί την αρχή που περιγράφεται παραπάνω. Διατηρούμε ένα δέντρο διεργασιών παρακολουθώντας τη διάρκεια ζωής όλων των διεργασιών. Οι κόμβοι του δέντρου διεργασίας είναι του ακόλουθου τύπου:

```
-type ptnode() :: {pid(), mfa(), [ptnode()]}.
```

Το `pid()` είναι το αναγνωριστικό διεργασίας, το `mfa()` είναι η αρχική συνάρτηση της διεργασίας και το `[ptnode()]` είναι μια λίστα με τις διεργασίες παιδιά της, που είναι άδειο αν η διεργασία είναι φύλλο. Η ουσία του δέντρου διεργασιών είναι ότι τα υποδέντρα που είναι αδέρφια και μοιράζονται την ίδια δομή και εκτελούν την ίδια αρχική συνάρτηση έχουν συνήθως την ίδια λειτουργικότητα. Επομένως, μπορούμε να καταγράψουμε μόνο ένα υποσύνολο αυτών των υποδέντρων και να έχουμε σχετικά ακριβή αποτελέσματα. Ωστόσο, η εύρεση υποδέντρων που μοιράζονται την ίδια δομή μπορεί να γίνει υπολογιστικά απαιτητική. Αντ' αυτού, αποφασίσαμε να βρούμε τις διεργασίες φύλλα που δημιουργήθηκαν χρησιμοποιώντας το ίδιο `mfa` και να τις ομαδοποιήσουμε. Έτσι, αντί να καταγράψουμε όλες τις διεργασίες κάθε ομάδας, καταγράφουμε μόνο ένα τυχαίο δείγμα τους. Η στρατηγική δειγματοληψίας που χρησιμοποιήσαμε είναι πολύ απλή αλλά εξακολουθεί να δίνει ικανοποιητικά αποτελέσματα. Όταν οι διεργασίες μιας ομάδας είναι περισσότερες από ένα καθορισμένο όριο (επί του παρόντος 50), το δείγμα είναι ένα μόνο ποσοστό (επί του παρόντος 10%) αυτών. Τα αποτελέσματα καταγραφής για αυτά

τα υποσύνολα στη συνέχεια κλιμακώνονται ανάλογα (δηλαδή πολλαπλασιάζονται με 10) για να αντιπροσωπεύουν καλύτερα τα πλήρη αποτελέσματα.

Είναι σημαντικό να σημειωθεί ότι η εφαρμογή της πιθανοτικής καταγραφής βασίζεται στη λειτουργικότητα καταγραφής του ERTS. Πιστεύουμε ότι η απόδοση του καταγραφέα θα μπορούσε να βελτιωθεί σημαντικά με την υλοποίηση του καταγραφέα διεργασιών στο ERTS.

3.3 Μεταγλωττιστής+Φορτωτής

Αυτό είναι το στοιχείο που είναι υπεύθυνο για την μεταγλώττιση και τη φόρτωση των μονάδων κώδικα. Λαμβάνει τα δεδομένα καταγραφής (δηλ. πληροφορίες τύπων και δεδομένα κλήσεων συναρτήσεων) από τον ελεγκτή, τα μορφοποιεί και καλεί μια εκτεταμένη έκδοση του μεταγλωττιστή HiPE για την μεταγλώττιση της μονάδας και τη φόρτωσή της. Ο μεταγλωττιστής HiPE επεκτείνεται με δύο πρόσθετες βελτιστοποιήσεις που καθοδηγούνται από τα δεδομένα καταγραφής. Αυτές οι βελτιστοποιήσεις περιγράφονται στο Κεφάλαιο 4.

Υπάρχουν πολλές προκλήσεις που πρέπει να αντιμετωπίσει το HiPErJiT, προκειμένου να μεταγλωττίσει και να φορτώσει αποτελεσματικά τον κώδικα Erlang. Αρχικά υπάρχει το *hot code loading*: η απαίτηση να είναι δυνατή η αντικατάσταση μονάδων, ατομικά, ενώ το σύστημα λειτουργεί και χωρίς να επιβάλλεται μακρά διακοπή της λειτουργίας του. Το δεύτερο χαρακτηριστικό, το οποίο συνδέεται στενά με το hot code loading, είναι ότι η γλώσσα κάνει μια σημασιολογική διάκριση μεταξύ των τοπικών κλήσεων και των *απομακρυσμένων κλήσεων*, όπως επίσης περιγράφεται στην Ενότητα 2.1.2.

Αυτά τα δύο χαρακτηριστικά, σε συνδυασμό με το γεγονός ότι η Erlang είναι πρωτίστως μια ταυτόχρονη γλώσσα στην οποία ένας μεγάλος αριθμός διεργασιών μπορεί να εκτελεί κώδικα από διαφορετικές μονάδες ταυτοχρόνως, επιβάλλει ουσιαστικά ότι η μεταγλώττιση συμβαίνει ανά μονάδα, χωρίς ευκαιρίες για βελτιστοποιήσεις μεταξύ διαφορετικών μονάδων. Η εναλλακτική λύση, δηλαδή η εκτέλεση βελτιστοποιήσεων μεταξύ διαφόρων μονάδων, απαιτεί το σύστημα χρόνου εκτέλεσης να είναι σε θέση να αναιρέσει γρήγορα τις βελτιστοποιήσεις στον κώδικα ορισμένων μονάδων που βασίζονται σε πληροφορίες από άλλες μονάδες, όποτε αλλάζουν αυτές οι μονάδες. Επειδή τέτοιες αλλαγές μπορούν να φτάσουν αυθαίρετα βαθιά, αυτή η εναλλακτική λύση δεν είναι πολύ ελκυστική από θέμα υλοποίησης και διαχείρισης του συστήματος. Επιπλέον, απαιτεί ότι το HiPErJiT ενεργοποιεί αυτόματα τη διαδικασία καταγραφής για μια μονάδα όταν αυτή επαναφορτωθεί.

Το σύστημα χρόνου εκτέλεσης του Erlang/OTP υποστηρίζει τη φόρτωση καυτού κώδικα με ένα συγκεκριμένο, αναμφισβήτητα αρκετά *ad-hoc* τρόπο. Επιτρέπει την ταυτόχρονη φόρτωση *μέχρι δύο* εκδόσεων κάθε μονάδας (m_{old} και $m_{current}$) και ανακατευθύνει όλες τις απομακρυσμένες κλήσεις στο $m_{current}$, το πιο πρόσφατο από τα δύο. Κάθε φορά που πρόκειται να φορτωθεί η m_{new}, μια νέα έκδοση του module m, όλες οι διαδικασίες που εξακολουθούν να εκτελούν κώδικα του m_{old} τερματίζονται απότομα, το $m_{current}$ γίνεται το νέο m_{old} και το m_{new} γίνεται $m_{current}$. Αυτός ο αρκετά περίπλοκος μηχανισμός εφαρμόζεται από τον φορτωτή κώδικα με υποστήριξη από το ERTS, το οποίο έχει τον έλεγχο όλων των διεργασιών.

Είναι σημαντικό να σημειωθεί ότι ο τρέχων μηχανισμός φόρτωσης κώδικα εισάγει μια μάλλον ασαφή αλλά ακόμη δυνατή "κούρσα δεδομένων" (data race) μεταξύ του HiPErJiT και του χρήστη. Εάν ο χρήστης προσπαθήσει να φορτώσει μια νέα έκδοση μιας μονάδας αφού η HiPErJiT ξεκίνησε να μεταγλωττίζει την τρέχουσα, αλλά πριν ολοκληρώσει τη φόρτωσή της, ο HiPErJiT μπορεί να φορτώσει JiT μεταγλωττισμένο κώδικα για μια παλαιότερη έκδοση της μονάδας, αφού ο χρήστης έχει φορτώσει την καινούργια. Επιπλέον, εάν το HiPErJiT προσπαθήσει να φορτώσει μια έκδοση της μονάδας m που έχει μετα-

γλωττιστεί όταν υπάρχουν διεργασίες που εκτελούν κώδικα m_{old}, αυτό θα μπορούσε να οδηγήσει σε θανάτωση αυτών των διεργασιών. Έτσι, ο HiPErJiT φορτώνει μια μονάδα m ως εξής. Κατ 'αρχάς, αποκτά ένα κλείδωμα μονάδας, που δεν επιτρέπει σε οποιαδήποτε άλλη διεργασία να επαναφορτώσει αυτή τη συγκεκριμένη μονάδα κατά τη διάρκεια μιας χρονικής περιόδου. Στη συνέχεια, καλεί το HiPE να μεταγλωττίσει τη μονάδα αε κώδικα μηχανής. Μετά την ολοκλήρωση της μεταγλώττισης, το HiPErJiT ελέγχει επανειλημμένα αν οποιαδήποτε διεργασία εκτελεί κώδικα m_{old}. Όταν καμία διαδικασία δεν εκτελεί κώδικα m_{old}, φορτώνεται η έκδοση που έχει μεταγλωττιστεί από το HiPErJiT. Τέλος, το HiPErJiT απελευθερώνει το κλείδωμα έτσι ώστε να μπορούν να φορτωθούν και πάλι οι νέες εκδόσεις της μονάδας m. Με αυτό τον τρόπο, το HiPErJiT δεν οδηγεχί σε θάνατο καμίας διεργασίας οταν φορτώνει κάποια μονάδα.

Φυσικά, το παραπάνω σχήμα δεν προσφέρει καμία εγγύηση προόδου, καθώς είναι δυνατό μια διεργασία να εκτελέσει κώδικα m_{old} για αόριστο χρονικό διάστημα. Σε αυτή την περίπτωση, ο χρήστης δε θα ήταν σε θέση να επαναφορτώσει την μονάδα (π.χ., μετά τη διόρθωση κάποιου σφάλματος). Προκειμένου να αποφευχθεί αυτό, το HiPErJiT σταματά να προσπαθεί να φορτώσει μια JiT μεταγλωττισμένη μονάδα μετά από ένα χρονικό όριο που έχει οριστεί από το χρήστη (η προεπιλεγμένη τιμή είναι 10 δευτερόλεπτα), απελευθερώνοντας έτσι το κλείδωμα της μονάδας.

Ας κλείσουμε αυτό το κεφάλαιο με μια σύντομη σημείωση σχετικά με την απόμεταγλώττιση. Οι περισσότεροι JiT μεταγλωττιστές υποστηρίζουν την απομεταγλώττιση, η οποία συνήθως ενεργοποιείται όταν ένα κομμάτι JiT μεταγλωττισμένου κώδικα δεν εκτελείται αρκετά συχνά πια. Το κύριο όφελος από αυτό είναι ότι, αφού ο κώδικας μηχανής καταλαμβάνει περισσότερο χώρο από τον bytecode, η απομεταγλώττιση συχνά μειώνει την κατανάλωση μνήμης του συστήματος. Ωστόσο, δεδομένου ότι το μέγεθος του κώδικα δεν αποτελεί μεγάλη ανησυχία στα σημερινά μηχανήματα και δεδομένου ότι η απομεταγλώττιση μπορεί να αυξήσει τον αριθμό των αλλαγών λειτουργίας (οι οποίες είναι γνωστό ότι προκαλούν σοβαρή επιβάρυνση), το HiPErJiT δεν υποστηρίζει την απομεταγλώττιση αυτή τη στιγμή.

Κεφάλαιο 4

Βελτιστοποιήσεις Βασισμένες σε Δεδομένα Χρόνου Εκτέλεσης

Ένα πλεονέκτημα των JiT μεταγλωττιστών σε σχέση με τους κλασσικούς στατικούς μεταγλωττιστές είναι ότι έχουν πρόσβαση σε δεδομένα χρόνου εκτέλεσης που αποκτώνται κατά την εκτέλεση ενός προγράμματος. Αυτό δημιουργεί ευκαιρίες για πιο επιθετικές βελτιστοποιήσεις, που δεν μπορούν να γίνουν με δεδομένα χρόνου μεταγλώττισης. Οι βελτιστοποιήσεις με βάση τα δεδομένα χρόνου εκτέλεσης έχουν μελετηθεί εκτεταμένα, ειδικά στο πλαίσιο των δυναμικών γλωσσών [Holz94, Bala00, Arno02, Arno05, Bolz09, Kedl13, Kedl14, Ren16]. Σε αυτό το κεφάλαιο, περιγράφουμε δύο βελτιστοποιήσεις που εκτελεί το HiPErJiT, με βάση τις συλλεγμένες πληροφορίες καταγραφής, πέρα από τις υπόλοιπες βελτιστοποιήσεις του HiPE: εξειδίκευση τύπων και ενσωμάτωση συναρτήσεων.

4.1 Εξειδίκευση Τύπων

Σε δυναμικές γλώσσες προγραμματισμού, κατά τη μεταγλώττιση βρίσκονται συνήθως πολύ λίγες πληροφορίες τύπων. Οι τύποι τιμών καθορίζονται κατά το χρόνο εκτέλεσης και λόγω αυτού, οι μεταγλωττιστές δυναμικών γλωσσών δημιουργούν κώδικα που χειρίζεται όλους τους πιθανούς τύπους τιμών, προσθέτοντας ελέγχους τύπων για να διασφαλιστεί ότι οι λειτουργίες εκτελούνται σε όρους σωστών τύπων. Επίσης, όλες οι τιμές επεκτείνονται με ετικέτες, πράγμα που σημαίνει ότι η παράστασή τους περιέχει πληροφορίες σχετικά με τον τύπο τους. Ο συνδυασμός αυτών των δύο χαρακτηριστικών οδηγεί σε μεταγλωττιστές που παράγουν λιγότερο αποτελεσματικό κώδικα από εκείνους για γλώσσες με στατικό σύστημα τύπων.

Αυτό το πρόβλημα έχει αντιμετωπίστει κατά καιρούς είτε με στατική ανάλυση τύπων [Lind03, Sago03], με ανατροφοδότηση τύπων [Holz94] ή με συνδυασμό και των δύο [Kedl13]. Η ανατροφοδότηση τύπων είναι ουσιαστικά ένας μηχανισμός που συλλέγει πληροφορίες τύπων από κλήσεις κατά τη διάρκεια του χρόνου εκτέλεσης και χρησιμοποιεί αυτές τις πληροφορίες για να δημιουργήσει εξειδικευμένες εκδόσεις των συναρτήσεων για τους συνηθέστερα χρησιμοποιούμενους τύπους κατά την επόμενη μεταγλώττιση του προγράμματος. Από την άλλη πλευρά, η στατική ανάλυση τύπων προσπαθεί να εξαγάγει πληροφορίες τύπων από τον κώδικα με συντηρητικό τρόπο, προκειμένου να την απλοποιήσει (π.χ., να εξαλείψει ορισμένους περιττούς ελέγχους τύπων). Στην περίπτωση του HiPErJiT, χρησιμοποιούμε έναν συνδυασμό αυτών των δύο μεθόδων.

4.1.1 Ανατροφοδότηση Τύπων

Όπως περιγράφηκε παραπάνω, η ανατροφοδότηση τύπων αποτελείται από δύο βασικά συστατικά:

- Συλλογή πληροφοριών τύπων χρόνου εκτέλεσης.
- Δημιουργία εξειδικευμένο κώδικα με βάση τις παραπάνω πληροφορίες τύπου.

Συλλογή Πληροφοριών Τύπων

Θα περιγράψουμε πρώτα πώς ο HiPErJiT συλλέγει τις πληροφορίες τύπων κατά τη διάρκεια της εκτέλεσης του προγράμματος. Πρώτα απ' όλα, ο HiPErJiT περιέχει ένα στοιχείο καταγραφής τύπων, όπως περιγράφεται στην Ενότητα 3.2, το οποίο συγκεντρώνει τις πληροφορίες τύπων για τα ορίσματα κλήσεων συναρτήσεων κατά τη διάρκεια εκτέλεσης. Συνεπώς, αυτή είναι η κύρια μέθοδος συλλογής πληροφοριών τύπων. Ωστόσο, όπως περιγράφεται στο Κεφάλαιο 2, η Erlang επιτρέπει επίσης στους χρήστες να διακοσμούν προγράμματα με πληροφορίες τύπου για λειτουργίες που χρησιμοποιούν προδιαγραφές τύπου. Δεδομένου ότι η Erlang είναι μια γλώσσα με δυναμικό σύστημα τύπων, αυτές οι προδιαγραφές δεν χρησιμοποιούνται από τον μεταγλωττιστή, αλλά από εργαλεία που εκτελούν στατικές αναλύσεις για να ανιχνεύσουν πιθανά σφάλματα [Lind04]. Οι προδιαγραφές τύπων δεν πρέπει να θεωρούνται σωστές από έναν μεταγλωττιστή, διότι σε περίπτωση που είναι λάθος το αποτέλεσμα μπορεί να είναι εσφαλμένος ή αργός κώδικας. Εντούτοις, σε ένα περιβάλλον ενός JiT μεταγλωττιστή, αυτές οι προδιαγραφές μπορούν να χρησιμοποιηθούν ως συμπλήρωμα στις πληροφορίες τύπων που συγκεντρώθηκαν κατά την εκτέλεση.

Λαμβάνοντας υπόψη όλα τα παραπάνω, υλοποιήσαμε ένα μηχανισμό που συνδυάζει πληροφορίες τύπων χρόνου εκτέλεσης με προδιαγραφές τύπων. Οι προδιαγραφές τύπων είναι ως επί το πλείστον χρήσιμες για συναρτήσεις που δεν έχουν αρκετές πληροφορίες τύπων από το χρόνο εκτέλεσης. Μια τελευταία σημείωση που πρέπει να ληφθεί υπόψη είναι ότι μια ασυνέπεια μεταξύ των προδιαγραφών τύπων και των καταγεγραμμένων πληροφοριών τύπων σημαίνει ότι η προδιαγραφή δεν είναι σωστή. Σε αυτή την περίπτωση το HiPErJiT ακυρώνει την προδιαγραφή και δεν τη χρησιμοποιεί καθόλου, αλλά, προς το παρόν, δε δίνει καμία προειδοποίηση στο χρήστη.

Αισιόδοξη Μεταγλώττιση Τύπων

Οι πληροφορίες τύπων χρόνου εκτέλεσης μπορεί να είναι κατά προσέγγιση ή μπορεί να μην ισχύουν για όλες τις επόμενες κλήσεις. Επομένως, η αισιόδοξη μεταγλώττιση τύπων προσθέτει ελέγχους τύπων που ελέγχουν αν τα ορίσματα έχουν τους κατάλληλους (με βάση την καταγραφή) τύπους. Σκοπός της είναι να δημιουργήσει εξειδικευμένες (αισιόδοξες) εκδόσεις συναρτήσεις, των οποίων τα ορίσματα είναι γνωστών τύπων.

Η αισιόδοξη μεταγλώττιση τύπων είναι το πρώτο πέρασμα του μεταγλωττιστή που εκτελείται στην Core Erlang, καθώς βελτιώνει το όφελος από τις επόμενες βελτιστοποιήσεις. Η υλοποίηση της είναι απλή. Για κάθε συνάρτηση `f` όπου οι πληροφορίες τύπων που συλλέγονται είναι μη τετριμμένες:

- Η συνάρτηση αντιγράφεται σε μια βελτιστοποιημένη έκδοση `f$opt` και μια απλή έκδοση `f$std` της συνάρτησης.

- Δημιουργείται μια συνάρτηση επικεφαλίδα που περιέχει όλους τους ελέγχους τύπων. Αυτή η συνάρτηση εκτελεί όλους τους απαραίτητους ελέγχους τύπων για κάθε όρισμα, για να εξασφαλίσει ότι ικανοποιούν τυχόν υποθέσεις στις οποίες μπορεί να βασιστεί η εξειδίκευση τύπων. Αν περάσουν όλοι οι έλεγχοι, η επικεφαλίδα καλεί την `f$opt`, αλλιώς καλεί την `f$std`.

- Η λειτουργία επικεφαλίδας ενσωματώνεται σε κάθε τοπική κλήση της `f`. Αυτό διασφαλίζει ότι οι έλεγχοι τύπων συμβαίνουν στην πλευρά του καλούντος, βελτιώνοντας έτσι το όφελος από τη διαδικασία αναλύσεων τύπων που συμβαίνει αργότερα.

4.1.2 Ανάλυση Τύπων

Η αισιόδοξη συλλογή τύπων από μόνη της δεν προσφέρει οφέλη απόδοσης. Απλά αντιγράφει τον κώδικα και οδηγεί στην εκτέλεση περισσότερων ελέγχων τύπων. Τα οφέλη της προκύπτουν από το συνδυασμό της με την ανάλυση τύπων.

Η ανάλυση τύπου είναι ένα πέρασμα βελτιστοποίησης που εκτελείται στο Icode, το οποίο συνάγει πληροφορίες τύπων για κάθε σημείο του προγράμματος και στη συνέχεια απλοποιεί τον κώδικα βάσει αυτών των πληροφοριών. Αφαιρεί τους περιττούς ελέγχους τύπων με βάση τις υπονοούμενες πληροφορίες τύπου. Επίσης απλοποιεί τις πράξεις δεκαδικών αριθμών. Μια σύντομη περιγραφή του αλγορίθμου ανάλυσης τύπων [Lind05] έχει ως εξής:

1. Κατασκευάζει το γράφο κλήσεων για όλες τις συναρτήσεις μιας μονάδας και το ταξινομεί τοπολογικά με βάση τις εξαρτήσεις μεταξύ των ισχυρά συνδεδεμένων στοιχείων (SCCs).

2. Αναλύει τα SCC από κάτω προς τα πάνω χρησιμοποιώντας έναν αλγόριθμο συμπερασμού τύπων που βασίζεται σε περιορισμούς για να βρει τα πιο γενικά success typings [Lind06] κάτω από τους τρέχοντες περιορισμούς.

3. Αναλύει τα SCC από πάνω προς τα κάτω χρησιμοποιώντας μια ανάλυση ροής δεδομένων για να μεταδώσει τις πληροφορίες τύπων από τις τοπικές κλήσεις στις κληθείσες συναρτήσεις.

4. Προσθέτει νέους περιορισμούς για τους τύπους, με βάση τις πληροφορίες που διαδόθηκαν από το προηγούμενο βήμα.

5. Εάν δεν έχει επιτευχθεί ένα σημείο σταθεροποίησης, επιστρέφει στο βήμα 2.

Οι αρχικοί περιορισμοί παράγονται κυρίως χρησιμοποιώντας τις πληροφορίες τύπων των ενσωματωμένων λειτουργιών της Erlang (BIFs) και των λειτουργιών από την τυπική βιβλιοθήκη που είναι γνωστές κατά την ανάλυση. Οι συνθήκες φρουροί και τα αντίστοιχα πρότυπα χρησιμοποιούνται επίσης για τη δημιουργία περιορισμών τύπων.

Παρουσιάζουμε ένα απλό παράδειγμα που παρουσιάζει τον αλγόριθμο και τη σημασία του συνδυασμού του περάσματος από κάτω με το πέρασμα από πάνω [Lind05].

```
1 -module(reverse).
2 -export([reverse/1]).
3
4 reverse(L) -> reverse(L, []).
5
6 reverse([H|T], Acc) -> reverse(T, [H|Acc]);
7 reverse([], Acc) -> Acc.
```

Listing 4.1: Μια μονάδα που προσφέρει μια συνάρτηση αντιστροφής για λίστες.

Καθώς τα SCC του γράφου κλήσεων αναλύονται από κάτω προς τα πάνω, η συνάρτηση `reverse/2` πρέπει πρώτα να αναλυθεί, αφού εξαρτάται μόνο από τον εαυτό της. Η εκτέλεση του συμπερασμού τύπων αποδίδει τα ακόλουθα success typings (με τη σειρά με την οποία αναλύονται οι συναρτήσεις):

```
reverse/2 :: (list(),any()) -> any()
reverse/1 :: (list()) -> any()
```

Αυτά τα success typings είναι σωστά, αλλά υπερεκτιμούν τον τύπο του δεύτερου ορίσματος της συνάρτησης `reverse/2`. Με την εφαρμογή της ανάλυσης ροής δεδομένων για τοπικές συναρτήσεις, ο τύπος του δεύτερου ορίσματος της `reverse/2` περιορίζεται σε

`list()` καθώς το `reverse/2` καλείται μόνο από το `reverse/1` με μια λίστα ως δεύτερο όρισμα. Το τελικό success typing είναι το:

```
reverse/2 :: (list(),list()) -> list()
reverse/1 :: (list()) -> list()
```

Αυτό το παράδειγμα ήταν αρκετά απλό και έτσι επιτυγχάνεται ένα σημείο σταθεροποίησης μετά από ένα μόνο πέρασμα του αλγορίθμου. Σε άλλα παραδείγματα, ο αλγόριθμος μπορεί να κάνει περισσότερα περάσματα μέχρι να βρει κάποιο σταθερό σημείο για τους τύπους που συμπεραίνει.

Μετά την ανάλυση τύπων, πραγματοποιούνται βελτιστοποιήσεις κώδικα. Πρώτον, όλοι οι περιττοί έλεγχοι τύπου ή άλλοι έλεγχοι απομακρύνονται πλήρως. Στη συνέχεια, ορισμένες εντολές, όπως οι πράξεις δεκαδικών, απλοποιούνται βάσει των διαθέσιμων πληροφοριών τύπων. Τέλος, η ροή ελέγχου απλοποιείται και αφαιρείται ο νεκρός κώδικας.

Είναι σημαντικό να σημειωθεί ότι η ανάλυση και η διάδοση τύπων περιορίζεται στα όρια της μονάδας. Δεν υπάρχουν υποθέσεις σχετικά με τα επιχειρήματα των εξαγόμενων συναρτήσεων, καθώς αυτές οι συναρτήσεις μπορούν να καλούνται από μονάδες που δεν υπάρχουν ακόμη στο σύστημα ή δεν είναι διαθέσιμες για ανάλυση. Έτσι, οι μονάδες που εξάγουν μόνο λίγες συναρτήσεις επωφελούνται περισσότερο από αυτή την ανάλυση καθώς παράγονται περισσότεροι περιορισμοί για τις τοπικές τους συναρτήσεις και χρησιμοποιούνται για την εξειδίκευση.

Ένα Παράδειγμα

Παρουσιάζουμε ένα απλό παράδειγμα που απεικονίζει το όφελος της εξειδίκευσης τύπων. Ο Κώδικας 4.2 περιέχει μια συνάρτηση που υπολογίζει τη δύναμη δύο τιμών, όπου η βάση είναι ένας οποιοσδήποτε αριθμός (ακέραιος ή δεκαδικός) και ο εκθέτης είναι ένας ακέραιος αριθμός.

```
1 -spec power(number(), integer(), number()) -> number().
2 power(_V1, 0, V3) -> V3;
3 power(V1, V2, V3) -> power(V1, V2-1, V1*V3).
```

Listing 4.2: Ο πηγαίος κώδικας μιας απλής συνάρτησης ύψωσης σε δύναμη.

Χωρίς την αισιόδοξη συλλογή τύπων, το HiPE δημιουργεί το Icode που εμφανίζεται στον Κώδικα 4.3.

```
1  power/3(v1, v2, v3) ->
2  12:
3      v5 := v3
4      v4 := v2
5      goto 1
6  1:
7      _ := redtest()    (primop)
8      if is_{integer,0}(v4) then goto 3 else goto 10
9  3:
10     return(v5)
11 10:
12     v8 := '-'(v4, 1)  (primop)
13     v9 := '*'(v1, v5) (primop)
14     v5 := v9
15     v4 := v8
16     goto 1
```

Listing 4.3: Παραγώμενο Icode για την συνάρτηση power.

Αν θεωρούμε ότι αυτή η συνάρτηση καλείται κυρίως με δεκαδικούς αριθμούς ως το πρώτο όρισμα, τότε το HiPE με αισιόδοξη μεταγλώττιση τύπων παράγει το Icode στον Κώδικα 4.4. Σημειώστε ότι `power$std` έχει το ίδιο Icode με το `power` χωρίς αισιόδοξη μεταγλώττιση τύπων.

```
power/3(v1, v2, v3) ->
16:
    _ := redtest()    (primop)
    if is_float(v1) then goto 3 else goto 14
3:
    if is_integer(v2) then goto 5 else goto 14
5:
    if is_float(v3) then goto 7 else goto 14
7:
    power$opt/3(v1, v2, v3)
14:
    power$std/3(v1, v2, v3)

power$opt/3(v1, v2, v3) ->
24:
    v5 := v3
    v4 := v2
    goto 1
1:
    _ := redtest() (primop)
    if is_{integer,0}(v4) then goto 3 else goto 20
3:
    return(v5)
20:
    v8 := '-'(v4, 1)  (primop)
    _ := gc_test<3>() (primop) -> goto 28, #fail 28
28:
    fv10 := unsafe_untag_float(v5) (primop)
    fv11 := unsafe_untag_float(v1) (primop)
    _ := fclearerror()             (primop)
    fv12 := fp_mul(fv11, fv10)     (primop)
    _ := fcheckerror()             (primop)
    v5 := unsafe_tag_float(fv12)   (primop)
    v4 := v8
    goto 1
```

Listing 4.4: Παραγόμενο Icode για την συνάρτηση power με αισιόδοξη μεταγλώττιση τύπων.

Όπως φαίνεται, η αισιόδοξη μεταγλώττιση τύπων μαζί με την ανάλυση τύπων βρήκε ότι τόσο οι `v1` όσο και οι `v3` είναι πάντα δεκαδικοί αριθμοί. Στη συνέχεια, ο κώδικας τροποποιήθηκε έτσι ώστε να βγαίνει η ετικέτα τους χωρίς έλεγχο σε κάθε επανάληψη και να πολλαπλασιάζονται χρησιμοποιώντας μια εντολή εξειδικευμένη για δεκαδικούς, ενώ χωρίς την αισιόδοξη μεταγλώττιση τύπων πολλαπλασιάζονται με τη συνήθη γενική συνάρτηση πολλαπλασιασμού, η οποία ελέγχει τους τύπους και των δύο ορισμάτων και τους βγάζει την ετικέτα πριν κάνει την πράξη.

4.1.3 Συζήτηση

Συνδυάζοντας την αισιόδοξη μεταγλώττιση τύπων, που εκτελεί το HiPErJiT, με το πέρασμα ανάλυσης τύπων του HiPE, φαίνεται να μειώνεται ο χρόνος εκτέλεσης των περισσότερων προγραμμάτων κατά ένα μικρό ποσοστό. Το σημαντικότερο όφελος, ωστόσο, είναι ότι επιτρέπει την περαιτέρω βελτιστοποίηση του αισιόδοξου κλάδου του κώδικα.

Ιδιαίτερα όταν συνδυάζεται με ενσωμάτωση συναρτήσεων, η αισιόδοξη μεταγλώττιση τύπων αποδίδει πολύτιμες βελτιώσεις στην απόδοση.

Ωστόσο, πρέπει να γίνουν ακόμα πολλά για να βελτιωθεί η αισιόδοξη συλλογή των τύπων, καθώς έχουμε αγγίξει μόνο την επιφάνεια των δυνατοτήτων της. Ορισμένες πιθανές κατευθύνσεις μελλοντικής εργασίας περιγράφονται συνοπτικά παρακάτω.

Πρώτα απ 'όλα, υπάρχει η ανάγκη να διασφαλιστεί δυναμικά ότι ο κώδικας περνάει από την αισιόδοξη συνάρτηση πράγματι συχνότερα από την αρχική. Η αισιόδοξη μεταγλώττιση τύπων βασίζεται σε πληροφορίες τύπου που συγκεντρώνονται μέσω της καταγραφής, επομένως είναι πιθανό κάποιες παραδοχές που κάνει να είναι λανθασμένες. Σε περίπτωση που διαπιστωθεί ότι η αισιόδοξη συνάρτηση εκτελείται σπάνια (ή και ποτέ), η συνάρτηση θα πρέπει να επαναμεταγλωττιστεί χωρίς την αισιόδοξη μεταγλώττιση τύπων.

Ένα άλλο πρόβλημα είναι ότι ορισμένοι έλεγχοι τύπων εκτελούνται πολλές φορές στη μη-αισιόδοξο συνάρτηση. Ένα παράδειγμα όπου υπάρχει το πρόβλημα αυτό ακολουθεί παρακάτω.

```
1  %% A type test added by our optimistic type compilation
2  foo/1(v1) ->
3  1:
4    if is_nil(v1) then goto 2 else goto 3
5  2:
6    foo$opt/1(v1)
7  3:
8    foo$std/1(v1)
9
10 %% The standard function
11 foo$std/1(v1) ->
12 1:
13   if is_nil(v1) then goto 2 else ...
14 2:
15   return v1
16 ...
17
18 %% The optimistic function
19 foo$opt/1(v1) ->
20   return v1
```

Listing 4.5: Ένα παράδειγμα περιττών ελέγχων τύπου.

Στο παραπάνω παράδειγμα, το πέρασμα ανάλυσης τύπων του HiPE απομάκρυνε τον έλεγχο τύπου `is_nil` στο `foo$opt` καθώς ο `v1` έχει ήδη περάσει έναν έλεγχο τύπου `is_nil`. Ωστόσο, ο έλεγχος τύπου στην `foo$std` δεν θα αφαιρεθεί παρόλο που είναι περιττός. Αυτό συμβαίνει επειδή η ανάλυση τύπου δεν ασχολείται με συμπληρωματικούς τύπους. Στο μέλλον, θα θέλαμε να πειραματιστούμε με συμπληρωματικούς τύπους και ενδεχομένως να επεκτείνουμε το πέρασμα ανάλυσης τύπων για να τους υποστηρίξουμε.

Τέλος, θα θέλαμε να πειραματιστούμε με πιο επιθετικές βελτιστοποιήσεις τύπων στις αισιόδοξες συναρτήσεις.

4.2 Ενσωμάτωση Συναρτήσεων

Η ενσωμάτωση συναρτήσεων (function inlining) είναι η διαδικασία αντικατάστασης μιας κλήσης συνάρτησης με το σώμα της καλούμενης συνάρτησης. Αυτό βελτιώνει την απόδοση με δύο τρόπους. Κατ' αρχάς, μετριάζει το κόστος της κλήσης. Δεύτερον, και πιο σημαντικό, επιτρέπει περισσότερες βελτιστοποιήσεις να πραγματοποιηθούν αργότερα, καθώς οι περισσότερες βελτιστοποιήσεις συνήθως δε διασχίζουν τα σύνορα συναρτήσεων. Αυτός είναι ο λόγος για τον οποίο η ενσωμάτωση πραγματοποιείται συνήθως

στις πρώτες φάσεις της μεταγλώττισης, έτσι ώστε οι μεταγενέστερες φάσεις να γίνονται πιο αποτελεσματικές.

Ωστόσο, η επιθετική ενσωμάτωση έχει αρκετά πιθανά μειονεκτήματα, τόσο στον χρόνο μεταγλώττισης όσο και στην αύξηση του μεγέθους του κώδικα (που με τη σειρά του μπορεί επίσης να οδηγήσει σε υψηλότερους χρόνους εκτέλεσης λόγω του caching). Ωστόσο, το μέγεθος του κώδικα είναι λιγότερο ανησυχητικό στις σημερινές μηχανές, εκτός φυσικά σε τομείς εφαρμογών όπου η μνήμη δεν είναι άφθονη (π.χ. σε IoT ή σε ενσωματωμένα συστήματα).

Προκειμένου να επιτευχθεί η μέγιστη απόδοση από την ενσωμάτωση, ο μεταγλωττιστής πρέπει να πετύχει μια λεπτή ισορροπία μεταξύ της ενσωμάτωσης κάθε κλήσης συνάρτησης και της μη ενσωμάτωσης. Επομένως, το πιο σημαντικό ζήτημα είναι η επιλογή των κλήσεων συναρτήσεων που αξίζει να ενσωματωθούν. Έχει γίνει πολυ δουλειά στον συγκεκριμένο τομέα για το πως να παρθεί αυτή η απόφαση στατικά [Sant95, Serr97, Wadd97, Peyt02], καθώς και με τη χρήση πληροφοριών χρόνου εκτέλεσης στο πλαίσιο των JiT μεταγλωττιστών [Ayer97, Gran99, Suga02, Haze03]. Το HiPErJiT παίρνει όλες τις αποφάσεις ενσωμάτωσης χρησιμοποιώντας πληροφορίες χρόνου εκτέλεσης, αλλά διατηρεί επίσης τον αλγόριθμό του αρκετά ελαφρύ ώστε να μην επιβάλλει μεγάλη επιβάρυνση.

4.2.1 Απόφαση Ενσωμάτωσης

Αποφασίσαμε να χρησιμοποιήσουμε έναν μηχανισμό απόφασης που χρησιμοποιεί πληροφορίες χρόνου εκτέλεσης για να αποφασίσει ποιες συναρτήσεις να ενσωματώσει. Ο μηχανισμός μας δανείζεται δύο ιδέες από προηγούμενες δουλειές, τη χρήση της συχνότητας κλήσης [Ayer97] και των γράφων κλήσεων [Suga02] για να καθοδηγήσει την απόφαση ενσωμάτωσης. Θυμηθείτε ότι το HiPErJiT μεταγλωττίζει ολόκληρες μονάδες τη φορά, και έτσι οι αποφάσεις λαμβάνονται βάσει πληροφοριών από όλες τις λειτουργίες μιας μονάδας. Λόγω της υποστήριξης φόρτωσης καυτού κώδικα, δεν πραγματοποιείται ενσωμάτωση μεταξύ συναρτήσεων σε διαφορετικές μονάδες.

Η εύρεση των πιο αποδοτικών κλήσεων συναρτήσεων για ενσωμάτωση, αλλά και η εύρεση της πιο αποτελεσματικής σειράς ενσωμάτωσης είναι ένα βαρύ υπολογιστικό καθήκον και έτσι αποφασίσαμε να χρησιμοποιήσουμε μια ευριστική για να αποφασίσουμε ποια κλήση να ενσωματώσουμε και πότε. Τα δεδομένα συχνότητας κλήσεων που χρησιμοποιούνται είναι ο αριθμός των κλήσεων μεταξύ κάθε ζεύγους συνάρτησεων, όπως επίσης περιγράφεται στην Ενότητα 3.2.

Πριν περιγράψουμε λεπτομερώς τον μηχανισμό μας, περιγράφουμε συνοπτικά την κύρια ιδέα και το σκεπτικό πίσω από το σχεδιασμό του.

Οι αποφάσεις βασίζονται στην υπόθεση ότι οι τόποι κλήσεων που καλούνται περισσότερο είναι οι καλύτεροι υποψήφιοι για ενσωμάτωση. Εξαιτίας αυτού ο μηχανισμός μας επιλέγει άπληστα να ενσωματώσει το ζεύγος συναρτήσεων (F_1, F_2) με τις περισσότερες κλήσεις από F_1 σε F_2.

Η άπληστη ενσωμάτωση μπορεί να επιβαρύνει κάπως την απόδοση καθώς κάποιες κλήσεις ενδέχεται να γίνονται πολλές φορές. Για παράδειγμα, ας εξετάσουμε την περίπτωση που ο μηχανισμός μας αποφασίζει να ενσωματώσει τα ακόλουθα ζεύγη με αυτή τη σειρά $[(F_1, F_2), (F_2, F_3), (F_1, F_3)]$. Σε αυτήν την περίπτωση η συνάρτηση F_3 θα ενσωματωθεί δύο φορές στη συνάρτηση F_2, μία φορά στην F_2 και μία φορά στο ενσωματωμένο αντίγραφο της F_2 στην F_1. Αυτό θα μπορούσε να είχε αποφευχθεί εάν η F_3 είχε αρχικά ενσωματωθεί στην F_2 και στη συνέχεια η F_2 είχε ενσωματωθεί στην F_1. Ωστόσο, η ανάλυση που απαιτείται για την επίτευξη αυτού του αποτελέσματος δεν είναι καθόλου αποδοτική. Επομένως, η απόφαση ενσωμάτωσης γίνεται άπληστα, παρά την περιστασιακή ενσωμάτωση συναρτήσεων περισσότερες από μία φορές.

Φυσικά, η ενσωμάτωση πρέπει να περιορίζεται, ώστε να μην συμβαίνει για κάθε κλήση συνάρτησης στο πρόγραμμα. Αυτό επιτυγχάνεται περιορίζοντας το μέγιστο μέγεθος κώ-

δικα κάθε ενότητας. Μικρές μονάδες επιτρέπεται να μεγαλώνουν μέχρι δύο φορές το μέγεθος τους, ενώ μεγαλύτερες επιτρέπεται να μεγαλώνουν μόνο κατά 10%.

4.2.2 Υλοποίηση

Αλγόριθμος Απόφασης Ενσωμάτωσης

Η ενσωμάτωση της διαδικασίας λήψης αποφάσεων γίνεται με επαναληπτικό τρόπο έως ότου δεν υπάρχουν πλέον κλήσεις για ενσωμάτωση ή έως ότου η μονάδα γίνει μεγαλύτερη από ένα καθορισμένο όριο.

Η κατάσταση δρα στις ακόλουθες δομές δεδομένων.

- Μια ουρά προτεραιότητας που περιέχει ζεύγη συναρτήσεων (F_1, F_2) και τον αριθμό κλήσεων $N_{F_2}^{F_1}$ από την F_1 στην F_2 για κάθε τέτοιο ζευγάρι. Αυτή η ουρά προτεραιότητας υποστηρίζει τρεις βασικές λειτουργίες:
  - Find-Maximum: που επιστρέφει το ζεύγος με το μεγαλύτερο αριθμό κλήσεων.
  - Delete-Maximum: που επιστρέφει και διαγράφει το ζεύγος με το μεγαλύτερο αριθμό κλήσεων.
  - Update: που ανανεώνει ένα ζεύγος με νέο αριθμό κλήσεων.
- Ένα σύνολο που περιέχει όλες τις κλήσεις (F_1, F_2) που έχουν ήδη ενσωματωθεί. Αυτό συμβαίνει για να αποφευχθούν βρόχοι ενσωμάτωσης. Εντούτοις, αποτρέπει και την ενσωμάτωση των αυτοαναδρομικών κλήσεων, οι οποίες θα μπορούσαν ενδεχομένως να βελτιώσουν την απόδοση. Αποφασίσαμε να διατηρήσουμε αυτόν τον απλό μηχανισμό, παρά το δυνητικά χαμένο όφελος απόδοση.
- Ένα χάρτης των συνολικών κλήσεων προς κάθε συνάρτηση που αρχικά κατασκευάστηκε για κάθε f με την προσθήκη του αριθμού κλήσεων για όλα τα ζεύγη στην ουρά προτεραιότητας $T_f = \sum_{f_i} N_f^{f_i}$.
- Χάρτης με το μέγεθος κάθε συνάρτησης, που χρησιμοποιείται για να υπολογίσει αν η μονάδα έχει γίνει μεγαλύτερη από το καθορισμένο όριο.

Ο κύριος βρόχος λειτουργεί ως εξής:

1. Delete-Maximum από την ουρά προτεραιότητας.
2. Ελέγχει ότι πληρούνται όλες οι ακόλουθες συνθήκες για το συγκεκριμένο ζεύγος λειτουργιών (F_1, F_2).
   - Να μην είναι ήδη ενσωματωμένο.
   - Να έχει περισσότερες κλήσεις από το προκαθορισμένο όριο.
   - Να μην κάνει την μονάδα περισσότερο από το προκαθορισμένο όριο..
3. Αν μία από αυτές τις συνθήκες αποτύχει, ο βρόχος συνεχίζει.
4. Σε διαφορετική περίπτωση:
   (a) Όλες οι τοπικές κλήσεις από την F_1 στην F_2 ενσωματώνονται.
   (b) Το σύνολο των ήδη ενσωματωμένων κλήσεων επεκτίνεται με το ζεύγος (F_1, F_2).
   (c) Ο χάρτης των μεγεθών ανανεώνεται για την συνατηση F_1 με το νέο μέγεθος της.

(d) Η ουρά προτεραιότητας ανανεώνεται ως εξής. Εντοπίζονται όλα τα ζεύγη (F_2, F_x) στην ουρά και επίσης όλες οι κλήσεις T_{F_2} για όλες την F_2. Ύστερα ανανεώνεται κάθε ζεύγος (F_1, F_x) στην ουρά με τον αριθμό κλήσεων $N_{F_x}^{F_1} + N_{F_2}^{F_1} * N_{F_x}^{F_2} / T_{F_2}$. Στην πράξη αυτό σημαίνει ότι όλες οι κλήσεις που γινόντουσαν στην F_2 τώρα θα γίνονται από την F_1.

Αλγόριθμος Ενσωμάτωσης

Ο αλγόριθμος ενσωμάτωσης που χρησιμοποιήσαμε είναι αρκετά συνηθισμένος, με εξαίρεση κάποιες λεπτομέρειες που αφορούν ειδικά το Icode. Προκειμένου να περιγράψουμε σωστά τον αλγόριθμο, θα καθορίσουμε πρώτα μια ελαφρώς απλούστερη μορφή του Icode, Icode-Mini, στο Σχήμα 4.1, ώστε να μπορούμε να εστιάσουμε στα βασικά συστατικά του αλγορίθμου χωρίς να χρειάζεται να ασχοληθούμε με περιττές λεπτομέρειες.

$c \in Const$	Constants
$x \in Var$	Variables
$l \in Label$	Labels
$f \in Function$	Functions
$t \in Type$	Types

$$
\begin{aligned}
v &::= c \mid x \\
i &::= \text{label } l \mid \text{if } v \text{ then } l_t \text{ else } l_f \mid \text{type } t\ v\ l_t\ l_f \mid \text{goto } l \mid \text{move } v_d\ v_s \mid \\
&\quad \text{phi } v_d\ (< l_1, v_1 >, ..., < l_n, v_n >) \mid \text{call } v_d\ f\ (v_1, ..., v_n) \mid \\
&\quad \text{enter } f\ (v_1, ..., v_n) \mid \text{return } v \mid \text{begin-try } l_1\ l_2 \mid \text{end-try} \mid \\
&\quad \text{fail } (v_1, ..., v_n)\ l
\end{aligned}
$$

Σχήμα 4.1: Ορισμός του Icode-mini.

Πρώτα απ' όλα, χρειαζόμαστε μια συνάρτηση που μετατρέπει όλες τις μεταβλητές και ετικέτες της κληθείσας συνάρτησης σε καινούργιες. Έστω $vmap$ μια συνάρτηση που αντιστοιχίζει μεταβλητές στη κληθείσα συνάρτηση σε νέες μοναδικές μεταβλητές στη καλούσα συνάρτηση. Έστω επίσης $lmap$ μια συνάρτηση που αντιστοιχίζει τις ετικέτες στην κληθείσα συνάρτηση σε νέες μοναδικές ετικέτες στη καλούσα συνάρτηση.

$$
\begin{aligned}
U(\text{label } l.is) &= \text{label } lmap(l).U(is) \\
U(\text{if } v \text{ then } l_t \text{ else } l_f.is) &= \text{if } vmap(v) \text{ then } lmap(l_t) \text{ else } lmap(l_f).U(is) \\
U(\text{type } t\ v\ l_t\ l_f.is) &= \text{type } t\ vmap(v)\ lmap(l_t)\ lmap(l_f).U(is) \\
U(\text{goto } l.is) &= \text{label } lmap(l).U(is) \\
U(\text{move } v_d\ v_s.is) &= \text{move } vmap(v_d)\ vmap(v_s).U(is) \\
U(\text{phi } v_d\ (< l_1, v_1 >, ...).is) &= \text{phi } vmap(v_d)\ (< lmap(l_1), vmap(v_1) >, ...).U(is) \\
U(\text{call } v_d\ f\ (v_1, ..., v_n).is) &= \text{call } vmap(v_d)\ f\ (vmap(v_1), ..., vmap(v_n)).U(is) \\
U(\text{enter } f\ (v_1, ..., v_n).is) &= \text{enter } f\ (vmap(v_1), ..., vmap(v_n)).U(is) \\
U(\text{return } v.is) &= \text{return } vmap(v).U(is) \\
U(\text{begin-try } l_1\ l_2.is) &= \text{begin-try } lmap(l_1)\ lmap(l_2).U(is) \\
U(\text{fail } (v_1, ..., v_n)\ l.is) &= \text{fail } (vmap(v_1), ..., vmap(v_n))\ lmap(l).U(is)
\end{aligned}
$$

Πρέπει επίσης να ορίσουμε μια συνάρτηση που μετατρέπει κάθε είσοδο σε συνατηση, σε μια κλήση σε συνάρτηση και σε μια επιστροφή. Στην πράξη η είσοδος έχει την ίδια λειτουργικότητα με μια κλήση και μια επιστροφή.

$$E(\text{enter } f\ (v_1, v_2, ..., v_k).is) = \text{call } v_e\ f\ (v_1, v_2, ..., v_k).\text{return } v_e.E(is)$$

Πρέπει επίσης να ορίσουμε μια συνάρτηση που μετατρέπει όλες τις εντολές επιστροφής σε μια μεταφορά και ένα άλμα καθώς οι επιστροφές της κληθείσας συνάρτησης δεν πρέπει να επιστρέφουν από τον καλούντα αλλά πρέπει να κρατούν μια προσωρινή μεταβλητή στον καλούντα.

$$R(\text{return } v.is, v_d, l_d) = \text{move } v_d\ v.\text{goto } l_d.is$$

Έστω ότι θα ενσωματώσουμε την f_2 στη συνάρτηση f_1. Κάθε συνάρτηση έχει ένα όνομα, ορίσματα, και ένα σώμα, το οποίο είναι μια ακολουθία εντολών. Έστω $body(f_2) = i_1.i_2.is$, $args(f_2) = a_1, a_2, ..., a_m$, $vmap$ και $lmap$ όπως ορίστηκε παραπάνω. Ο τελικός μετασχηματισμός είναι ο παρακάτω.

$$\begin{aligned} I(\text{call } v_d\ f_2\ (v_1, v_2, ..., v_m).is) =& \text{move } vmap(a_1)\ v_1 ... \text{move } vmap(a_m)\ v_m. \\ & R(E(U(body(f_2))), v_d, l_d).\text{label } l_d.I(is) \\ I(\text{enter } f_2\ (v_1, v_2, ..., v_m).is) =& \text{move } vmap(a_1)\ v_1 ... \text{move } vmap(a_m)\ v_m \\ & U(body(f_2)).I(is) \end{aligned}$$

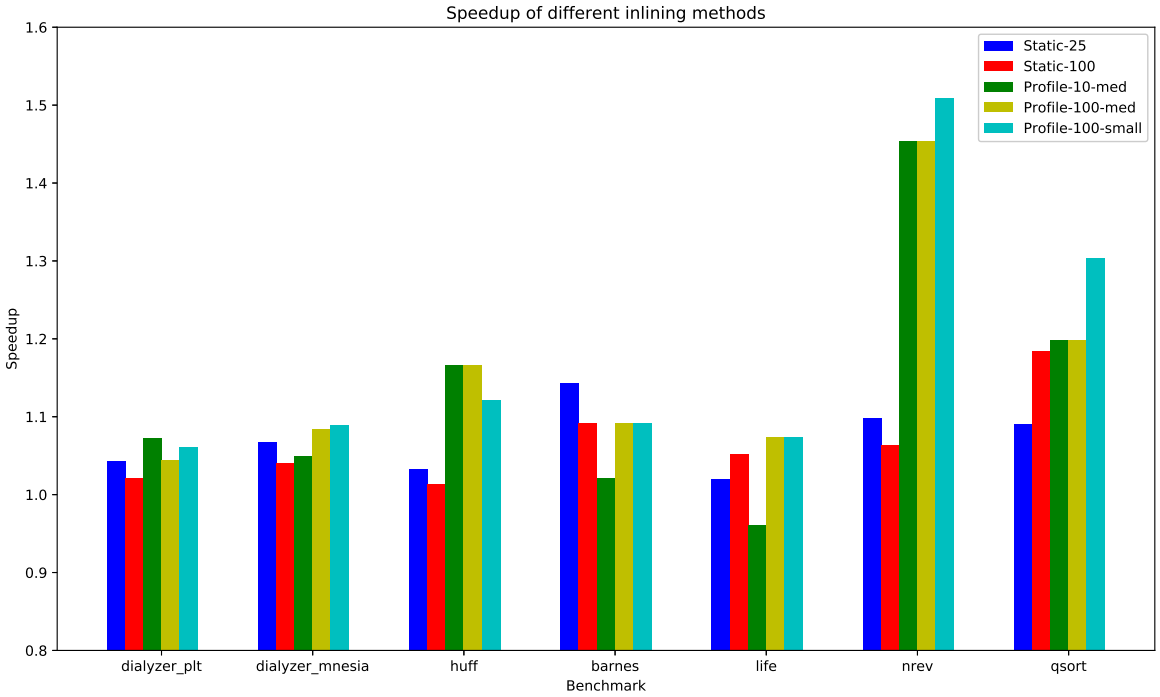
Ο μετασχηματισμός I ορίζεται μόνο για τις ενδιαφέρουσες περιπτώσεις, καθώς σε όλες τις άλλες περιπτώσεις λειτουργεί ως ταυτοτικός μετασχηματισμός.

Αξίζει να σημειωθεί ότι η Erlang χρησιμοποιεί συνεργατική χρονοδρομολόγηση, η οποία υλοποιείται διασφαλίζοντας ότι οι διεργασίες εκτελούνται για έναν αριθμό κλήσεων και στη συνέχεια παραδίδουν τη θέση τους σε κάποια άλλη. Όταν εκτελεστούν όλες οι κλήσεις μιας διεργασίας, η διεργασία αναστέλλεται και χρονοδρομολογείται για εκτέλεση μια άλλη διεργασία. Αυτή η διαδικασία υλοποιείται στο Icode από την `redtest()` (primop) στην αρχή κάθε σώματος συνάρτησης. βλ. Κώδικα 4.3. Όταν ενσωματώνεται μια κλήση συνάρτησης, η κλήση στο `redtest()` στο σώμα της κληθείσας αφαιρείται επίσης.

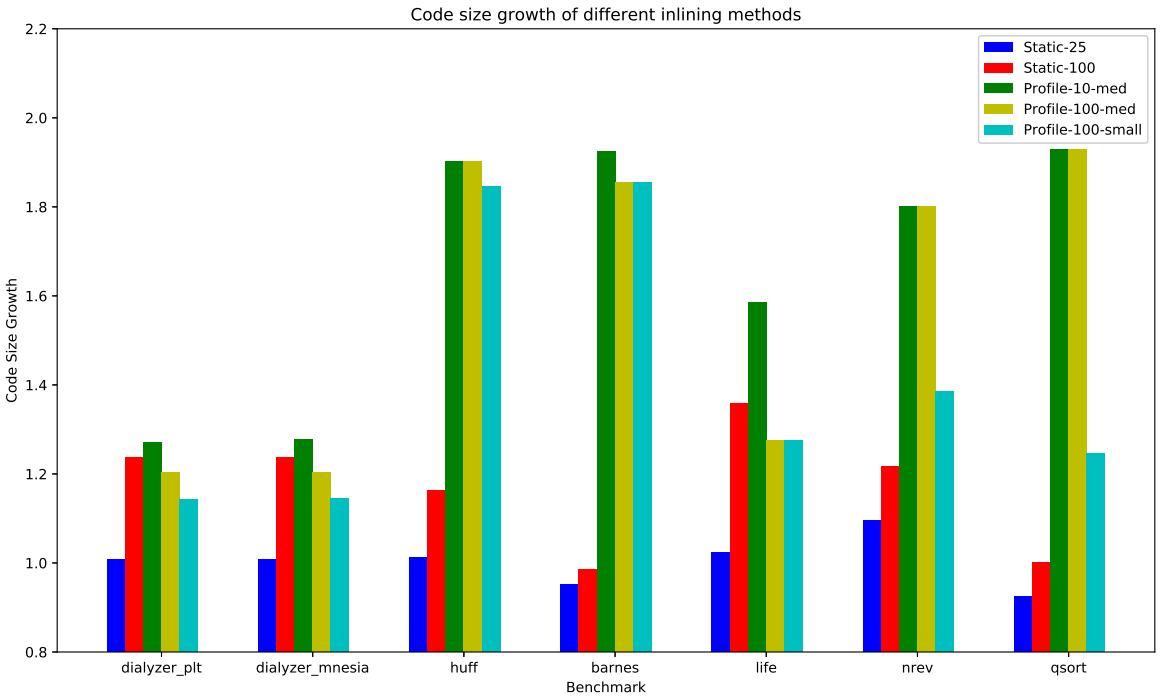
Σύγκριση με τη Στατική Ενσωμάτωση

Προκειμένου να διασφαλίσουμε ότι η ενσωμάτωση με χρήση δεδομένων από το χρόνο εκτέλεσης παρέχει πραγματικό όφελος απόδοσης, τη συγκρίνουμε με τη στατική ενσωμάτωση που προσφέρει ο μεταγλωττιστής της Erlang. Αυτός ο αλγόριθμος στατικής ενσωμάτωσης είναι μια εφαρμογή του αλγορίθμου από τους Waddell και Dybvig [Wadd97], προσαρμοσμένη στη γλώσσα Core Erlang, με μια μικρή αλλαγή. Αντί να μετονομάζει πάντα μεταβλητές χρησιμοποιεί τη στρατηγική [Peyt02] "no-shadowing". Μετρήσαμε τους χρόνους εκτέλεσης και τα παραγόμενα μεγέθη κωδικών πολλών εφαρμογών με στατική ή δυναμική ενσωμάτωση. Η επιτάχυνση των διαφορετικών μεθόδων ενσωμάτωσης σε σύγκριση με τη μεταγλώττιση χωρίς ενσωμάτωση εμφανίζεται στο Σχήμα 4.2. Η μεγέθυνση του κώδικα των διάφορων μονάδων για όλες τις μεθόδους παρουσιάζεται στο Σχήμα 4.3. Οι μέθοδοι που συγκρίναμε είναι δύο στατικές (μια πιο συντηρητική *Static-25* και μία πιο επιθετική *Static-100*) και τρεις που βασίζονται σε δεδομένα χρόνου εκτέλεσης (ο αριθμός στα ονόματα των μεθόδων υποδηλώνει τον ελάχιστο αριθμό κλήσεων για να ενσωματωθεί μια κλήση και το *small/med* περιγράφει την επιτρεπόμενη αύξηση του μεγέθους του κώδικα).

Στο Σχήμα 4.2 φαίνεται ότι όλες οι διαφορετικές μέθοδοι ενσωμάτωσης βελτιώνουν την ταχύτητα εκτέλεσης όταν χρησιμοποιούνται. Η ενσωμάτωση που βασίζεται σε δεδομένα χρόνου εκτέλεσης γενικά οδηγεί σε καλύτερους χρόνους εκτέλεσης εκτός από το



Σχήμα 4.2: Επιτάχυνση διαφορετικών μεθόδων ενσωμάτωσης.



Σχήμα 4.3: Ανάπτυξη μεγέθους κώδικα διαφορετικών μεθόδων ενσωμάτωσης.

πρόγραμμα barnes. Οι πιο συντηρητικές μέθοδοι (*Profile-100-small* και *Profile-100-med*) φαίνεται να έχουν καλύτερες επιδόσεις.

Όπως φαίνεται στο Σχήμα 4.3, δύο διατάξεις της ενσωμάτωσης με δεδομένα χρόνου εκτέλεσης οδηγούν σε σχετικά μεγάλη αύξηση του μεγέθους του κώδικα. Αυτό μπορεί να αποδοθεί κυρίως σε δύο αιτίες. Καταρχήν, η μέθοδος ενσωμάτωσης *Profile-10-med* είναι πολύ επιθετική και ενσωματώνει κάθε κλήση που έχει συμβεί περισσότερο από δέκα φορές (γεγονός που οδηγεί σε πολύ μεγάλη αύξηση κώδικα για το πρόγραμμα life). Επιπλέον, όπως εξηγείται και στην Ενότητα 4.2.1, το μέγιστο όριο αύξησης του κώδικα είναι μεταβλητό, έτσι οι μικρότερες μονάδες μπορούν να μεγαλώσουν περισσότερο από τις μεγαλύτερες μονάδες (ειδικά στις διατάξεις *med*). Επομένως, τα προγράμματα huff, barnes, nrev, qsort (που αποτελούνται από μικρές μονάδες) έχουν μεγαλύτερο όριο και μπορούν να μεγαλώσουν πολύ περισσότερο.

Είναι ενδιαφέρον να σημειωθεί ότι η διάταξη ενσωμάτωσης *Static-25* γενικά δεν οδηγεί σε αύξηση κώδικα (σε ορισμένες περιπτώσεις μειώνει το μέγεθος του κώδικα). Αυτό μπορεί να αποδοθεί στο ότι η στατική μέθοδος ενσωμάτωσης περιέχει μια ανάλυση που καταργεί τις συναρτήσεις που ενσωματώθηκαν και δεν καλούνται από πουθενά πια. Δεν έχουμε ακόμη εφαρμόσει αυτή την ανάλυση μετά από το δικό μας πέρασμα ενσωμάτωσης, αλλά αναμένουμε να μειωθεί το μέγεθος του παραγόμενου κώδικα αν την υλοποιήσουμε.

Κεφάλαιο 5

Αξιολόγηση

Σε αυτό το κεφάλαιο, αξιολογούμε την απόδοση του HiPErJiT έναντι του BEAM, το οποίο και χρησιμοποιούμε ως βάση για τη σύγκρισή μας, και έναντι δύο συστημάτων που στοχεύουν επίσης να ξεπεράσουν τις επιδόσεις του BEAM. Το πρώτο από αυτά είναι ο μεταγλωττιστής HiPE.[1] Το δεύτερο είναι το Pyrlang[2] το οποίο είναι ένα πρωτότυπο και όχι μια ολοκληρωμένη υλοποίηση, και για αυτό το λόγο αναφέρουμε τις επιδόσεις του στο υποσύνολο των προγραμμάτων που υποστηρίζει. Δεν είχαμε τη δυνατότητα να έχουμε στη διάθεσή μας μια έκδοση του BEAMJIT για να το συμπεριλάβουμε στη σύγκρισή μας, καθώς το σύστημα αυτό δεν είναι ακόμα (Μάιος 2018) διαθέσιμο στο κοινό. Ωστόσο, όπως αναφέρθηκε στην Ενότητα 2.3.1, το BEAMJIT δεν επιτυγχάνει απόδοση που να είναι ανώτερη από αυτήν του HiPE ούτως ή άλλως.

Πραγματοποιήσαμε όλα τα πειράματα σε φορητό υπολογιστή με επεξεργαστή Intel Core i7-4710HQ @ 2.50GHz και 16 GB μνήμης RAM με Ubuntu 16.04.

5.1 Επιβάρυνση Καταγραφής

Το πρώτο σύνολο μετρήσεων αφορά την επιβάρυνση που επιβάλλει η καταγραφή του HiPErJiT. Για να αποκτήσουμε μια εκτίμηση για τη χειρότερη περίπτωση, χρησιμοποιήσαμε μια τροποποιημένη έκδοση του HiPErJiT που καταγράφει προγράμματα αλλά δεν μεταγλωττίζει καμία μονάδα σε κώδικα μηχανής. Σημειώστε ότι αυτό το σενάριο είναι πολύ απαισιόδοξο για το HiPErJiT, επειδή το σύνηθες είναι ότι όταν JiT μεταγλώττιση ενεργοποιείται για ορισμένες μονάδες, ο HiPErJiT θα σταματήσει να τις καταγράφει και αυτές οι μονάδες θα εκτελούνται πιθανότατα πιο γρήγορα, όπως θα δούμε παρακάτω. Εν πάση περιπτώσει, οι μετρήσεις μας έδειξαν ότι η γενική επιβάρυνση που προκαλείται από την πιθανοτική καταγραφή είναι περίπου 10% του χρόνου εκτέλεσης. Πιο συγκεκριμένα, η επιβάρυνση που μετρήσαμε κυμαίνονταν από 5% (στις περισσότερες περιπτώσεις) έως 40% για μερικά προγράμματα με πολλές διεργασίες.

Επίσης, μετρήσαμε χωριστά την επιβάρυνση καταγραφής με και χωρίς πιθανοτική καταγραφή σε διάφορα προγράμματα με πολλές διεργασίες, για να μετρήσουμε το όφελος της πιθανοτικής καταγραφής. Η μέση γενική επιβάρυνση που επιβάλλει η τυπική καταγραφή στα παράλληλα προγράμματα είναι 19%. ενώ η μέση επιβάρυνση της πιθανοτικής καταγραφής είναι 13%.

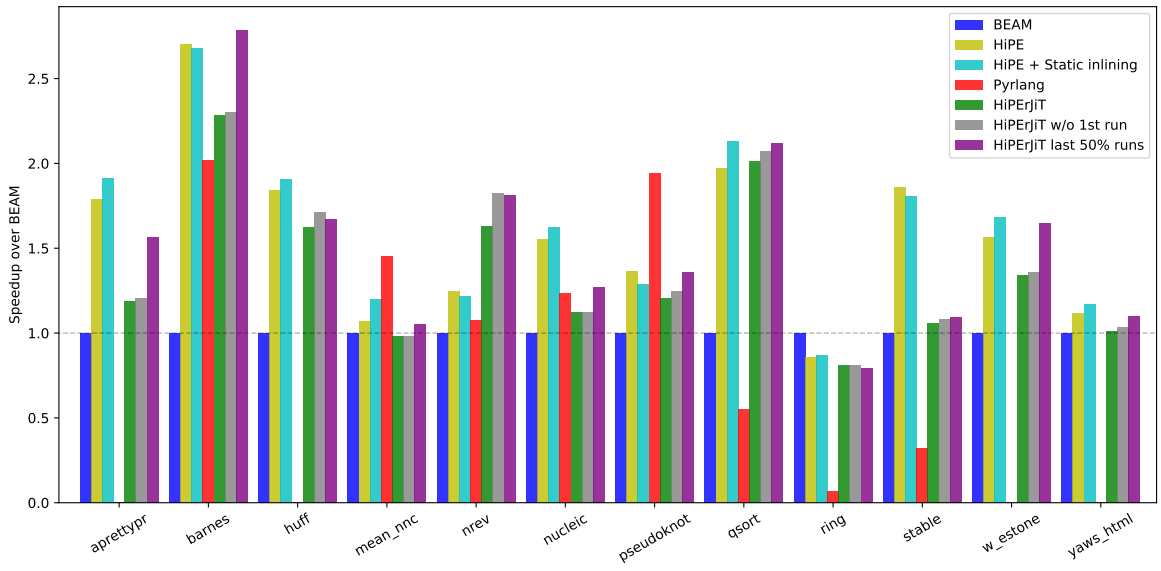
5.2 Αξιολόγηση σε Μικρά Προγράμματα

Τα προγράμματα που χρησιμοποιήσαμε προέρχονται από το ErLLVM benchmark suite[3], το οποίο έχει χρησιμοποιηθεί στο παρελθόν για την αξιολόγηση των HiPE [Joha00],

[1] Για τα BEAM και HiPE, χρησιμοποιήσαμε το 'master' branch του Erlang/OTP 21.0.

[2] Χρησιμοποιήσαμε την τελευταία έκδοση του Pyrlang (https://bitbucket.org/hrc706/pyrlang/overview) που κατασκευάστηκε με PyPy 5.0.1.

[3] Είναι διαθέσιμα στο https://github.com/cstavr/erllvm-bench



Σχήμα 5.1: Επιτάχυνση σε σχέση με το BEAM σε μικρά προγράμματα.

Πίνακας 5.1: Επιτάχυνση σε σχέση με το BEAM σε μικρά προγράμματα.

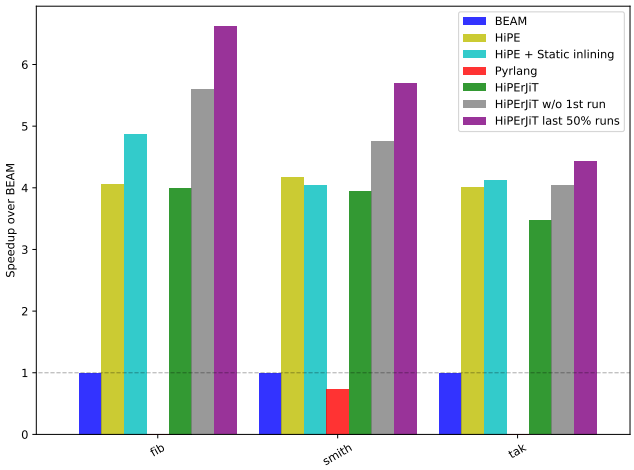
Configuration	Speedup
HiPE	2.08
HiPE + Static inlining	2.17
Pyrlang	1.05
HiPErJiT	1.85
HiPErJiT w/o 1st run	2.08
HiPErJiT last 50% runs	2.33

ErLLVM [Sago12], BEAMJIT [Drej14] και Pyrlang [Huan16]. Τα προγράμματα χωρίζονται σε δύο κατηγορίες: (1) Ένα σύνολο μικρών και σχετικά απλών, αλλά αντιπροσωπευτικών προγραμμάτων Erlang, και (2) το σύνολο των προγραμμάτων από το Computer Language Benchmarks Game (CLBG)[4] όπως ήταν στο ErLLVM benchmark suite όταν δημιουργήθηκε.

Η σύγκριση της απόδοσης ενός Just-in-Time μεταγλωττιστή με έναν τυπικό μεταγλωττιστή σε μικρά προγράμματα είναι δύσκολη, καθώς ο JiT μεταγλωττιστής πληρώνει επίσης το κόστος μεταγλώττισης κατά την εκτέλεση του προγράμματος. Επιπλέον, ένας JiT μεταγλωττιστής χρειάζεται κάποιο χρόνο για να ζεσταθεί. Για το λόγο αυτό, αναφέρουμε τρεις αριθμούς για το HiPErJiT: την επιτάχυνση που επιτυγχάνεται κατά τη διάρκεια όλων των εκτελέσεων, την επιτάχυνση που επιτυγχάνεται όταν αγνοήσουμε την πρώτη εκτέλεση του κάθε προγράμματος και την επιτάχυνση που επιτυγχάνεται στα τελευταία 50% των εκτελέσεων, όταν έχει επιτευχθεί σταθερή λειτουργία. Χρησιμοποιούμε επίσης δύο διαφορετικές διαμορφώσεις του HiPE: μια με το μέγιστο επίπεδο βελτιστοποίησης (`o3`) και μία όπου έχει γίνει static inlining (χρησιμοποιώντας την οδηγία `{inline_size, 25}` στον μεταγλωττιστή) εκτός από το `o3`.

Η επιτάχυνση των HiPErJiT, HiPE, και Pyrlang σε σύγκριση με το BEAM για τα μικρά προγράμματα εμφανίζεται στα Σχήματα 5.1 και 5.2. Τα έχουμε χωρίσει σε δύο σχήματα για λόγους καλύτερης ορατότητας, με βάση την κλίμακα του άξονα y, την επιτάχυνση σε σύγκριση με το BEAM. Σημειώστε ότι όλες οι επιταχύνσεις που αναφέρουμε είναι μέσοι

[4] Είναι διαθέσιμα στο `http://benchmarksgame.alioth.debian.org/`.



Σχήμα 5.2: Επιτάχυνση σε σχέση με το BEAM σε μικρά προγράμματα.

όροι πολλών διαφορετικών εκτελέσεων. Η συνολική μέση επιτάχυνση για κάθε διάταξη συνοψίζεται στον Πίνακα 5.1.

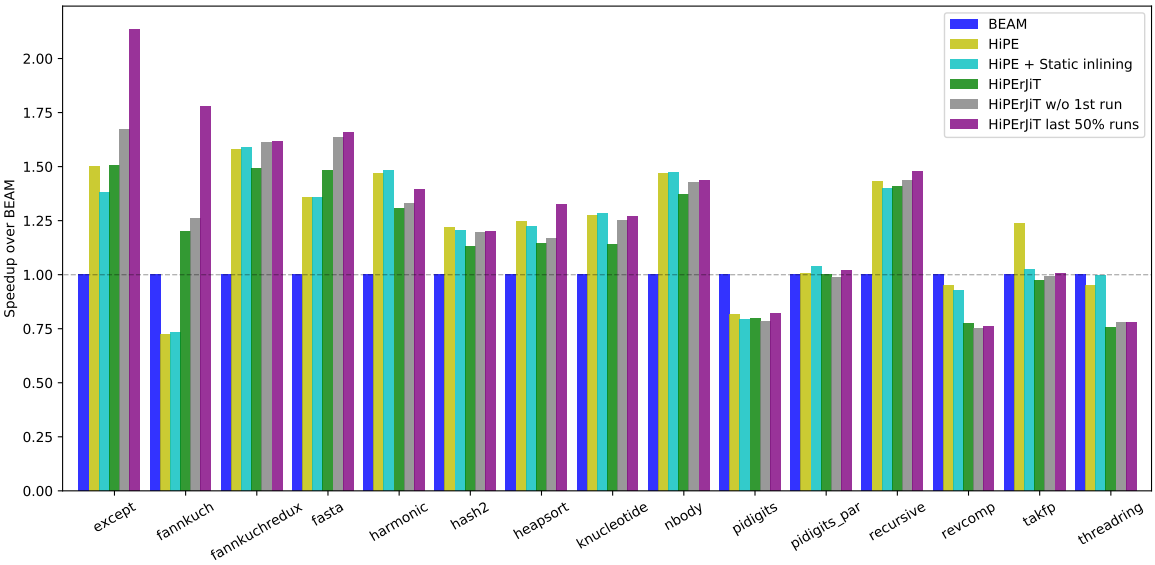
Συνολικά, η απόδοση του HiPErJiT είναι σχεδόν δύο φορές καλύτερη από το BEAM και το Pyrlang, αλλά ελαφρώς χειρότερη από το HiPE. Ωστόσο, υπάρχουν επίσης πέντε σημεία αναφοράς (barnes, nrev, fib, smith και tak) όπου το HiPERjiT, στην σταθερή λειτουργία, ξεπερνά την απόδοση του HiPE. Αυτό δείχνει έντονα ότι οι βελτιστοποιήσεις που καθοδηγούνται από δεδομένα καταγραφής που εκτελεί το HiPERjiT οδηγούν σε πιο αποδοτικό κώδικα, σε σύγκριση με εκείνον ενός τυπικού μεταγλωττιστή κώδικα μηχανής που εκτελεί στατική ενσωμάτωση (static inlining).

Από τα πέντε αυτά προγράμματα, το πιο ενδιαφέρον είναι το smith. Πρόκειται για μια εφαρμογή του αλγόριθμου Smith-Waterman για αντιστοίχιση αλληλουχιών DNA. Ο λόγος για τον οποίο το HiPERjiT προσφέρει καλύτερη επιτάχυνση σε σχέση με το HiPE είναι ότι η ενσωμάτωση με χρήση δεδομένων χρόνου εκτέλεσης ενσωματώνει τις συναρτήσεις `alpha_beta_penalty/2` και `max/2` στη `match_entry/5`, που είναι η πιο κρίσιμη λειτουργία του προγράμματος, επιτρέποντας έτσι περαιτέρω βελτιστοποιήσεις για τη βελτίωση της απόδοσης. Η στατική ενσωμάτωση από την άλλη πλευρά δεν οδηγεί στην ενσωμάτωση την `max/2` στη `match_entry/5` ακόμα κι αν κάποιος επιλέξει πολύ μεγάλες τιμές για τις παραμέτρους της μεθόδου.

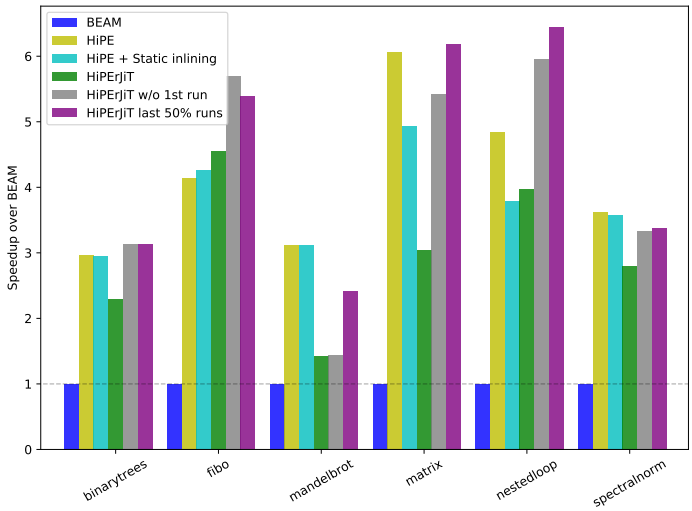
Στο πρόγραμμα ring (Σχήμα 5.2), το οποίο περιέχει πολλές ταυτόχρονες διεργασίες, το HiPERjiT έχει χειρότερη απόδοση από το HiPE και το BEAM. Πιο συγκεκριμένα, όλες οι διατάξεις είναι χειρότερα από το BEAM σε αυτό το σημείο αναφοράς. Ο κύριος λόγος είναι ότι η πλειονότητα του χρόνου εκτέλεσης δαπανάται για τη μετάδοση μηνυμάτων (περίπου 1,5 εκατομμύρια μηνύματα αποστέλλονται ανά δευτερόλεπτο), το οποίο αποτελεί μέρος της λειτουργικότητας του συστήματος χρόνου εκτέλεσης του BEAM. Τέλος, υπάρχουν πολλές δημιουργίες και καταστροφές διεργασιών (περίπου 20 χιλιάδες ανά δευτερόλεπτο), γεγονός που οδηγεί σε σημαντικές επιβαρύνσεις, καθώς το HiPERjiT διατηρεί και ενημερώνει συνεχώς το δέντρο διεργασιών.

Ένα άλλο ενδιαφέρον πρόγραμμα αναφοράς είναι το stable (επίσης Σχήμα 5.2), όπου το HiPERjiT έχει ελαφρώς καλύτερη απόδοση από το BEAM αλλά εμφανώς χειρότερη από το HiPE. Αυτό οφείλεται κυρίως στο γεγονός ότι και σε αυτό το πρόγραμμα υπάρχουν πολλές δημιουργίες και καταστροφές διεργασιών (περίπου 240 χιλιάδες ανά δευτερόλεπτο), γεγονός που οδηγεί σε σημαντική επιβάρυνση λόγω της καταγραφής.

Η επιτάχυνση των HiPERjiT και HiPE σε σύγκριση με το BEAM για τα προγράμματα CLBG φαίνεται στα Σχήματα 5.3 και 5.4. Η συνολική μέση επιτάχυνση για κάθε διαφορετική διάταξη φαίνεται στον Πίνακα 5.2.



Σχήμα 5.3: Επιτάχυνση σε σχέση με το BEAM στα προγράμματα CLBG.



Σχήμα 5.4: Επιτάχυνση σε σχέση με το BEAM στα προγράμματα CLBG.

Πίνακας 5.2: Επιτάχυνση σε σχέση με το BEAM για τα προγράμματα CLBG.

Configuration	Speedup
HiPE	2.05
HiPE + Static inlining	1.93
HiPErJiT	1.78
HiPErJiT w/o 1st run	2.14
HiPErJiT last 50% runs	2.26

Όπως και με τα μικρά προγράμματα, η απόδοση του HiPErJiT βρίσκεται κυρίως μεταξύ του BEAM και του HiPE. Όταν εξαιρείται η πρώτη εκτέλεση, το HiPErJiT υπερέχει τόσο των BEAM όσο και του HiPE σε διάφορα προγράμματα (binarytrees, except, fannkuch, fannkuchredux, fasta, fibo, nestedloop, και recursive). Τέλος, αν εξετάσουμε μόνο τη σταθερή λειτουργία του JiT μεταγλωττιστή μας, υπερέχει τόσο του HiPE όσο και του BEAM σε ένα ακόμη πρόγραμμα (matrix). Ωστόσο, η απόδοση του HiPErJiT σε ορισμένα προγράμματα (revcomp, takfp και threadring) είναι χειρότερη από αυτή του BEAM και του HiPE, επειδή οι επιβαρύνσεις από την καταγραφή δεν ξεπερνούν τα οφέλη από τη μεταγλώττιση.

5.3 Αξιολόγηση σε Ένα Μεγαλύτερο Πρόγραμμα

Εκτός από τα μικρά προγράμματα, αξιολογήσαμε επίσης τις επιδόσεις του HiPErJiT σε ένα πρόγραμμα μεγάλου μεγέθους και πολυπλοκότητας, καθώς τα αποτελέσματα μικρών ή μεσαίων προγραμμάτων ενδέχεται να μην παρέχουν πάντα μια πλήρη εικόνα για την γενική απόδοση ενός μεταγλωττιστή. Το πρόγραμμα Erlang που επιλέξαμε, το εργαλείο στατικής ανάλυσης Dialyzer [Lind04], είναι τόσο μεγάλο (περίπου 30.000 γραμμές κώδικα) όσο και πολύπλοκο και με πολλές ταυτόχρονες διεργασίες [Aron13]. Έχει επίσης δοκιμαστεί και βελτιωθεί αρκετά με τα χρόνια και κατά την εκκίνηση του μεταγλωττίζει 26 μονάδες σε κώδικα μηχανής, γιατί έχει βρεθεί από τους δημιουργούς του ότι έτσι πετυχαίνει τη μεγαλύτερη απόδοση για τις περισσότερες περιπτώσεις χρήσης. Χρησιμοποιώντας μια κατάλληλη επιλογή, ο χρήστης μπορεί να απενεργοποιήσει τη φάση μεταγλώττισης σε κώδικας μηχανής (η οποία διαρκεί περισσότερο από ένα λεπτό στο φορητό υπολογιστή που χρησιμοποιούμε) και στην πραγματικότητα αυτό ακριβώς κάνουμε για να μετρήσουμε το χρόνο εκτέλεσης της εφαρμογής στην περίπτωση του BEAM και του HiPErJiT.

Χρησιμοποιούμε το εργαλείο Dialyzer με δύο τρόπους. Ο πρώτος δημιουργεί έναν πίνακα αναζήτησης (Persistent Lookup Table) που περιέχει πληροφορίες για όλες τις μονάδες που περιλαμβάνονται στις εφαρμογές erts, compiler, crypto, hipe, kernel, stdlib και syntax_tools. Ο δεύτερος τρόπος αναλύει αυτές τις εφαρμογές για σφάλματα τύπων και άλλα προβλήματα. Οι επιταχύνσεις του HiPErJiT και του HiPE σε σύγκριση με το BEAM για τους δύο τρόπους χρήσης του Dialyzer εμφανίζονται στον Πίνακα 5.3.

Πίνακας 5.3: Επιτάχυνση σε σχέση με το BEAM για τις δύο περιπτώσεις χρήσης του Dialyzer.

	Dialyzer Speedup	
Configuration	Building PLT	Analyzing
HiPE	1.73	1.70
HiPE + Static inlining	1.75	1.72
HiPErJiT	1.51	1.30
HiPErJiT w/o 1st run	1.58	1.35
HiPErJiT last 50% runs	1.62	1.37

Τα αποτελέσματα δείχνουν ότι ο HiPErJiT επιτυγχάνει απόδοση που είναι καλύτερη από το BEAM, αλλά χειρότερη από το HiPE. Υπάρχουν διάφοροι λόγοι για αυτό. Πρώτα απ' όλα, το εργαλείο Dialyzer είναι μια πολύπλοκη εφαρμογή όπου οι συναρτήσεις από πολλές διαφορετικές μονάδες του Erlang/OTP (50–100) καλούνται με ορίσματα σημαντικού μεγέθους (π.χ. τον πηγαίο κώδικα των εφαρμογών που αναλύονται). Αυτό οδηγεί σε σημαντικές επιβαρύνσεις λόγω της καταγραφής. Δεύτερον, μερικές από τις καλούμενες μονάδες περιέχουν πολύ μεγάλες συναρτήσεις, οι οποίες συχνά παράγονται από κάποιο

μεταγλωττιστή, και η μεταγλώττιση τους διαρκεί πολύ χρόνο (ο συνολικός χρόνος μεταγλώττισης είναι περίπου 70 δευτερόλεπτα, ο οποίος αποτελεί σημαντικό μέρος του συνολικού χρόνου). Τέλος, ο HiPErJiT δε μεταγλωττίζει όλες τις μονάδες από την αρχή, πράγμα που σημαίνει ότι ένα ποσοστό του χρόνου δαπανάται εκτελώντας ερμηνευμένο κώδικα και πραγματοποιώντας αλλαγές λειτουργίας (mode switches) που είναι ακριβότερες από τις κλήσεις ίδιας λειτουργίας. Αντίθετα, ο HiPE έχει κωδικοποιημένη γνώση του "καλύτερου" συνόλου μονάδων που πρέπει να μεταγλωττιστεί σε κώδικα μηχανής πριν την έναρξη της ανάλυσης.

Κεφάλαιο 6

Επίλογος

6.1 Η Τρέχουσα Κατάσταση του HiPErJiT

Παρουσιάσαμε το HiPErJiT, ένα Just-in-Time μεταγλωττιστή, που χρησιμοποιεί δεδομένα καταγραφής χρόνου εκτέλεσης για το οικοσύστημα BEAM και ειναι βασισμένος στον μεταγλωττιστή HiPE. Προσφέρει καλύτερη απόδοση από το BEAM και συγκρίσιμη με το HiPE, σε διάφορα προγράμματα. Εκτός από την απόδοση, έχουμε φροντίσει να διατηρήσουμε σημαντικά χαρακτηριστικά για τον τομέα εφαρμογών της Erlang, όπως τη φόρτωση "καυτού" κώδικα, και έχουμε προσπαθήσει με τις σχεδιαστικές αποφάσεις μας να μεγιστοποιήσουμε τις πιθανότητες το HiPErJiT να μένει εύκολα συγχρονισμένο με τα υπόλοιπα τμήματα της υλοποίησης του συστήματος Erlang/OTP. Συγκεκριμένα, εκτός από τη χρήση του μεταγλωττιστή HiPE για τις περισσότερες βελτιστοποιήσεις, το HiPErJiT χρησιμοποιεί την υποστήριξη για ταυτοχρονισμό που παρέχει το σύστημα χρόνου εκτέλεσης της Erlang. Έτσι, μπορεί να επωφεληθεί από τυχόν βελτιώσεις που μπορεί να προκύψουν σε αυτά τα στοιχεία.

Η βασική μας ανησυχία σχετικά με το HiPErJiT είναι η επιβάρυνση που προκύπτει από την καταγραφή, ειδικά σε σύνθετες εφαρμογές που περιέχουν πολλές ταυτόχρονες διεργασίες. Προς το παρόν, δεν προκαλεί σημαντικά προβλήματα, διότι μόλις το HiPErJiT αποφασίσει να μεταγλωττίσει μια μονάδα σε κώδικα μηχανής, σταματά την καταγραφή οπότε και η επιβάρυνση μηδενίζεται σε εκείνο το σημείο. Ωστόσο, στο πλαίσιο της συνεχούς βελτιστοποίησης κατά τη διάρκεια του χρόνου εκτέλεσης, που είναι η κατεύθυνση που θέλουμε τελικά να ακολουθήσουμε, η επιβάρυνση λόγω καταγραφής θα καταστεί πιο σημαντικό ζήτημα.

6.2 Μελλοντικές Εργασίες

Παρά το γεγονός ότι η τρέχουσα υλοποίηση του HiPErJiT είναι αρκετά σταθερή και λειτουργεί ικανοποιητικά καλά, οι JiT μεταγλωττιστές είναι κυρίως αντικείμενα μηχανικής και δεν μπορούν ποτέ να θεωρηθούν εντελώς "ολοκληρωμένα". Επιπλέον, το HiPErJiT είναι απλώς το πρώτο βήμα προς τον τελικό στόχο της συνεχούς βελτιστοποίησης των προγραμμάτων κατά των χρόνο εκτέλεσης τους. Επομένως, υπάρχουν πολλές κατευθύνσεις εργασίας που θα θέλαμε να ακολουθήσουμε στο μέλλον.

- Διερεύνηση τεχνικών που μειώνουν την επιβάρυνση καταγραφής του HiPErJiT, ειδικά σε εφαρμογές που περιέχουν πολλές ταυτόχρονες διεργασίες.
- Αξιολόγηση του HiPErJiT σε πραγματικές εφαρμογές που εκτελούνται για μεγάλα χρονικά διαστήματα, καθώς τέτοια προγράμματα είναι ο ιδανικός στόχος για μεταγλωττιστές συνεχούς βελτιστοποίησης κατά τη διάρκεια του χρόνου εκτέλεσης.
- Βελτίωση της σταθερότητας και της ακρίβειας των μηχανισμών καταγραφής του HiPErJiT.

- Σχεδιασμός και υλοποίηση πιο αποτελεσματικών βελτιστοποιήσεων τύπων που θα μπορούσαν να επωφεληθούν από την εξειδίκευση τύπων και την ενσωμάτωση.
- Βελτίωση της απόδοσης της μετάδοσης μηνυμάτων χρησιμοποιώντας δεδομένα καταγραφής χρόνου εκτέλεσης.

Βιβλιογραφία

[AdlT03] Ali-Reza Adl-Tabatabai, Jay Bharadway, Dong-Yuan Chen, Anwar Ghuloum, Vijay Menon, Brian Murphy, Mauricio Serrano and Tatiana Shpeisman, "The StarJIT compiler: A dynamic compiler for managed runtime environments", *Intel Technology Journal*, vol. 07, no. 01, 2003.

[Arms03] Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, Ph.D. thesis, Royal Institute of Technology Stockholm, Sweden, 2003.

[Arno00a] Ken Arnold, James Gosling, David Holmes and David Holmes, *The Java programming language*, vol. 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[Arno00b] Matthew Arnold, Stephen Fink, David Grove, Michael Hind and Peter F Sweeney, "Adaptive Optimization in the Jalapeño JVM", in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pp. 47–65, New York, NY, USA, 2000, ACM.

[Arno02] Matthew Arnold, Michael Hind and Barbara G Ryder, "Online Feedback-directed Optimization of Java", in *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pp. 111–129, New York, NY, USA, 2002, ACM.

[Arno05] M Arnold, S J Fink, D Grove, M Hind and P F Sweeney, "A Survey of Adaptive Optimization in Virtual Machines", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, 2005.

[Aron13] Stavros Aronis and Konstantinos Sagonas, "On Using Erlang for Parallelization — Experience from Parallelizing Dialyzer", in *Trends in Functional Programming, 13th International Symposium, TFP 2012, Revised Selected Papers*, vol. 7829 of *LNCS*, pp. 295–310, Springer, 2013.

[Ayco03] John Aycock, "A Brief History of Just-in-time", *ACM Comput. Surv.*, vol. 35, no. 2, pp. 97–113, 2003.

[Ayer97] Andrew Ayers, Richard Schooler and Robert Gottlieb, "Aggressive Inlining", in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pp. 134–145, New York, NY, USA, 1997, ACM.

[Babc04] Brian Babcock, Mayur Datar and Rajeev Motwani, "Load Shedding for Aggregation Queries over Data Streams", in *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pp. 350–361, Washington, DC, USA, 2004, IEEE Computer Society.

[Bala00] Vasanth Bala, Evelyn Duesterwald and Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System", in *Proceedings of the ACM*

SIGPLAN 2000 Conference on Programming Language Design and Implementation, PLDI '00, pp. 1–12, New York, NY, USA, 2000, ACM.

[Baum15] Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G Siek and Sam Tobin-Hochstadt, "Pycket: A Tracing JIT for a Functional Language", in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pp. 22–34, New York, NY, USA, 2015, ACM.

[Bebe10] Michael Bebenita, Florian Brandner, Manuel Fahndrich, Francesco Logozzo, Wolfram Schulte, Nikolai Tillmann and Herman Venter, "SPUR: A Trace-based JIT Compiler for CIL", in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pp. 708–725, New York, NY, USA, 2010, ACM.

[Bolz09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo, "Tracing the Meta-level: PyPy's Tracing JIT Compiler", in *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pp. 18–25, New York, NY, USA, 2009, ACM.

[Brig94] Preston Briggs, Keith D. Cooper and Linda Torczon, "Improvements to Graph Coloring Register Allocation", *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 428–455, 1994.

[Burg97] Robert G Burger, *Efficient compilation and profile-driven dynamic recompilation in scheme*, Ph.D. thesis, Indiana University, 1997.

[Burg98] Robert G Burger and R Kent Dybvig, "An Infrastructure for Profile-Driven Dynamic Recompilation", in *ICCL*, 1998.

[Carl04] Richard Carlsson, Thomas Lindgren, Björn Gustavsson, Sven-Olof Nyström, Robert Virding, Erik Johansson and Mikael Pettersson, "Core Erlang 1.0.3 language specification", Technical report, Uppsala University, 2004.

[Cham89] Craig Chambers and David Ungar, "Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language", in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pp. 146–160, New York, NY, USA, 1989, ACM.

[Cham91] Craig Chambers and David Ungar, "Making Pure Object-oriented Languages Practical", in *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pp. 1–15, New York, NY, USA, 1991, ACM.

[Cier00] Michał Cierniak, Guei-Yuan Lueh and James M Stichnoth, "Practicing JUDO: Java Under Dynamic Optimizations", in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pp. 13–26, New York, NY, USA, 2000, ACM.

[Deut84] Peter L. Deutsch and Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 System", in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '84, pp. 297–302, New York, NY, USA, 1984, ACM.

[Drej14] Frej Drejhammar and Lars Rasmusson, "BEAMJIT: A Just-in-time Compiling Runtime for Erlang", in *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, Erlang '14, pp. 61–72, New York, NY, USA, 2014, ACM.

[Eric] Ericsson AB, "EDoc - Erlang documentation generator".

[Fell15] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A McCarthy and Sam Tobin-Hochstadt, "The Racket Manifesto", in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 113–128, Dagstuhl, Germany, 2015, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Gal09] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W Smith, Rick Reitmaier, Michael Bebenita, Mason Chang and Michael Franz, "Trace-based Just-in-time Type Specialization for Dynamic Languages", in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pp. 465–478, New York, NY, USA, 2009, ACM.

[Gran99] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers and Susan J Eggers, "An Evaluation of Staged Run-time Optimizations in DyC", in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pp. 293–304, New York, NY, USA, 1999, ACM.

[Gude93] David Gudeman, "Representing type information in dynamically typed languages", Technical report, University of Arizona, Department of Computer Science, 1993.

[Hans74] Gilbert Joseph Hansen, *Adaptive Systems for the Dynamic Run-time Optimization of Programs*, Ph.D. thesis, Carnegie-Mellon University, 1974.

[Harc97] Mor Harchol-Balter and Allen B Downey, "Exploiting Process Lifetime Distributions for Dynamic Load Balancing", *ACM Trans. Comput. Syst.*, vol. 15, no. 3, pp. 253–285, 1997.

[Haze03] Kim Hazelwood and David Grove, "Adaptive Online Context-sensitive Inlining", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pp. 253–264, Washington, DC, USA, 2003, IEEE Computer Society.

[Hejl06] Anders Hejlsberg, Scott Wiltamuth and Peter Golde, *C# Language Specification*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Holz94] Urs Hölzle and David Ungar, "Optimizing Dynamically-dispatched Calls with Run-time Type Feedback", in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pp. 326–336, New York, NY, USA, 1994, ACM.

[Huan15] Ruochen Huang, Hidehiko Masuhara and Tomoyuki Aotani, "Pyrlang: A High Performance Erlang Virtual Machine Based on RPython", in *Companion Proceedings of the 2015 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH Companion 2015, pp. 48–49, New York, NY, USA, 2015, ACM.

[Huan16] Ruochen Huang, Hidehiko Masuhara and Tomoyuki Aotani, "Improving Sequential Performance of Erlang Based on a Meta-tracing Just-In-Time Compiler", in *Post-Proceeding of the 17th Symposium on Trends in Functional Programming*, 2016.

[Hugh95] John Hughes, "The Design of a Pretty-printing Library", in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pp. 53–96, London, UK, 1995, Springer-Verlag.

[IEEE08] "IEEE Standard for Floating-Point Arithmetic", 2008.

[Jime07] Miguel Jimenez, Tobias Lindahl and Konstantinos Sagonas, "A Language for Specifying Type Contracts in Erlang and Its Interaction with Success Typings", in *Proceedings of the 2007 SIGPLAN Workshop on Erlang*, Erlang '07, pp. 11–17, New York, NY, USA, 2007, ACM.

[Joha96] Erik Johansson and Christer Jonsson, "Native Code Compilation for Erlang", Technical report, Computing Science Department, Uppsala University, October 1996.

[Joha00] Erik Johansson, Mikael Pettersson and Konstantinos Sagonas, "A High Performance Erlang System", in *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '00, pp. 32–43, New York, NY, USA, 2000, ACM.

[Joha03] Erik Johansson, Mikael Pettersson, Konstantinos Sagonas and Thomas Lindgren, "The development of the HiPE system: design and experience report", *International Journal on Software Tools for Technology Transfer*, vol. 4, no. 4, pp. 421–436, 2003.

[Kedl13] Madhukar N Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi and Ben Hardekopf, "Improved Type Specialization for Dynamic Scripting Languages", in *Proceedings of the 9th Symposium on Dynamic Languages*, DLS '13, pp. 37–48, New York, NY, USA, 2013, ACM.

[Kedl14] Madhukar N Kedlaya, Behnam Robatmili, Cġlin Ca\cscaval and Ben Hardekopf, "Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines", in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '14, pp. 103–114, New York, NY, USA, 2014, ACM.

[Kenn07] Andrew Kennedy, "Compiling with Continuations, Continued", in *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pp. 177–190, New York, NY, USA, 2007, Cambridge University Press.

[Kist03] Thomas Kistler and Michael Franz, "Continuous Program Optimization: A Case Study", *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[Kotz08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell and David Cox, "Design of the Java HotSpot client compiler for Java 6", *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 1, pp. 1–32, 2008.

[Kulk11] Prasad A Kulkarni, "JIT Compilation Policy for Modern Machines", in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pp. 773–788, New York, NY, USA, 2011, ACM.

[Latt04] Chris Lattner and Vikram Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation", in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pp. 75—-, Washington, DC, USA, 2004, IEEE Computer Society.

[Lind03] Tobias Lindahl and Konstantinos Sagonas, "Unboxed Compilation of Floating Point Arithmetic in a Dynamically Typed Language Environment", in *Proceedings of the 14th International Conference on Implementation of Functional Languages*, IFL'02, pp. 134–149, Berlin, Heidelberg, 2003, Springer Berlin Heidelberg.

[Lind04] Tobias Lindahl and Konstantinos Sagonas, "Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story", in Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, pp. 91–106, Berlin, Heidelberg, 2004, Springer-Verlag.

[Lind05] Tobias Lindahl and Konstantinos Sagonas, "TypEr: A Type Annotator of Erlang Code", in *Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang*, Erlang '05, pp. 17–25, New York, NY, USA, 2005, ACM.

[Lind06] Tobias Lindahl and Konstantinos Sagonas, "Practical Type Inference Based on Success Typings", in *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, pp. 167–178, New York, NY, USA, 2006, ACM.

[Luna04] Daniel Luna, Mikael Pettersson and Konstantinos Sagonas, "HiPE on AMD64", in *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang*, pp. 38–47, New York, NY, USA, September 2004, ACM.

[Pett02] Mikael Pettersson, Konstantinos Sagonas and Erik Johansson, "The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation", in *Functional and Logic Programming, 6th International Symposium, FLOPS 2002, Proceedings*, vol. 2441 of *LNCS*, pp. 228–244, Berlin, Heidelberg, September 2002, Springer.

[Peyt02] Simon Peyton Jones and Simon Marlow, "Secrets of the Glasgow Haskell Compiler Inliner", *J. Funct. Program.*, vol. 12, no. 5, pp. 393–434, 2002.

[Reis05] F Reiss and J M Hellerstein, "Data Triage: an adaptive architecture for load shedding in TelegraphCQ", in *21st International Conference on Data Engineering (ICDE'05)*, pp. 155–156, 2005.

[Ren16] Brianna M Ren and Jeffrey S Foster, "Just-in-time Static Type Checking for Dynamic Languages", in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pp. 462–476, New York, NY, USA, 2016, ACM.

[Sago03] Konstantinos Sagonas, Mikael Pettersson, Richard Carlsson, Per Gustafsson and Tobias Lindahl, "All You Wanted to Know About the HiPE Compiler: (but Might Have Been Afraid to Ask)", in *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, Erlang '03, pp. 36–42, New York, NY, USA, 2003, ACM.

[Sago12] Konstantinos Sagonas, Chris Stavrakakis and Yiannis Tsiouris, "ErLLVM: an LLVM Backend for Erlang", in *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop*, pp. 21–32, New York, NY, USA, 2012, ACM.

[Sant95] Andre Santos, *Compilation by transformation for non-strict functional languages*, Ph.D. thesis, University of Glasgow, Scotland, 1995.

[Serr97] Manuel Serrano, "Inline expansion: When and how?", in *Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP '97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, September 3–5, 1997 Proceedings*, pp. 143–157, Berlin, Heidelberg, 1997, Springer Berlin Heidelberg.

[Stee77] Guy Lewis Steele Jr., "Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO", in *Proceedings of the 1977 Annual Conference*, ACM '77, pp. 153–162, New York, NY, USA, 1977, ACM.

[Suga00] Toshio Suganuma, T Ogasawara, M Takeuchi, Toshiaki Yasue, M Kawahito, K Ishizaki, H Komatsu and Toshio Nakatani, "Overview of the IBM Java Just-in-time Compiler", *IBM Syst. J.*, vol. 39, no. 1, pp. 175–193, 2000.

[Suga02] Toshio Suganuma, Toshiaki Yasue and Toshio Nakatani, "An Empirical Study of Method In-lining for a Java Just-in-Time Compiler", in *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pp. 91–104, Berkeley, CA, USA, 2002, USENIX Association.

[Tama83] Hisao Tamaki and Taisuke Sato, "Program transformation through meta-shifting", *New Generation Computing*, vol. 1, no. 1, pp. 93–98, 1983.

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

[Wadd97] Oscar Waddell and R Kent Dybvig, "Fast and effective procedure inlining", in Pascal Van Hentenryck, editor, *Static Analysis*, pp. 35–52, Berlin, Heidelberg, 1997, Springer Berlin Heidelberg.