Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Σχεδίαση και υλοποίηση ενός μεταγλωττιστή κώδικα μηχανής για τη γλώσσα Erlang με χρήση της LLVM

Διπλωματική Εργασία

των

Χρήστου Σταυρακάκη, Γιάννη Τσιούρη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού

Αθήνα, Νοέμβριος 2011

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

Σχεδίαση και υλοποίηση ενός μεταγλωττιστή κώδικα μηχανής για τη γλώσσα Erlang με χρήση της LLVM

Διπλωματική Εργασία

των

Χρήστου Σταυρακάκη, Γιάννη Τσιούρη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 7η Νοεμβρίου, 2011.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Άρης Κοζύρης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Νοέμβριος 2011

....

Χρήστος Σταυρακάκης **Γιάννης Τσιούρης**

Διπλωματούχοι Ηλεκτρολόγοι Μηχανικοί και Μηχανικοί Υπολογιστών Ε.Μ.Π.

Copyright © – All rights reserved Χρήστος Σταυρακάκης, Γιάννης Τσιούρης, 2011.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Στις μέρες μας, οι υπάρχοντες μεταγλωττιστές ανοιχτού κώδικα χρησιμοποιούν απαρχαιωμένες τεχνικές παραγωγής κώδικα μηχανής και δεν μοιράζονται μεταξύ τους κώδικα υλοποίησης. Επιπλέον, αποτελούνται από πολύπλοκο κώδικα υλοποίησης, γεγονός που καθιστά δύσκολο το να μελετηθούν και ακόμα πιο δύσκολο το να αλλάξουν. Η Low Level Virtual Machine (LLVM) είναι μια βιβλιοθήκη τελευταίας τεχνολογίας για μεταγλωττιστές. Παρέχει ένα σύνολο από επαναχρησιμοποιήσιμα εργαλεία που υλοποιούν τις καλύτερες τεχνικές μεταγλώττισης και στατικής ανάλυσης, εστιάζοντας στην επίδοση του παραγόμενου κώδικα και στο χρόνο μεταγλώττισης. Ο απώτερος στόχος της LLVM είναι να παρέχει ανεξάρτητα εξαρτήματα για την κατασκευή υψηλής ποιότητας μεταγλωττιστών για πολλές διαφορετικές γλώσσες.

Η διπλωματική αυτή περιγράφει την παρούσα αρχιτεκτονική, τις βασικές σχεδιαστικές αποφάσεις και κάποιες λεπτομέρειες υλοποίησης ενός νέο οπίσθιου τμήματος για τον HiPE, το μεταγλωττιστή κώδικα μηχανής του συστήματος της Erlang/OTP, που χρησιμοποιεί την υποδομή της LLVM. Διαπιστώθηκε πως μία από τις ενδιάμεσες γλώσσες του HiPE, η Register Transfer Language (RTL), έχει πολύ απλή και προφανή απεικόνιση στη συμβολική γλώσσα της LLVM. Παρόλα αυτά, υπήρχαν κάποια λεπτά σημεία, όπως η σύμβαση κλήσης συναρτήσεων, ο μηχανισμός χειρισμού εξαιρέσεων και η συλλογή σκουπιδιών, που απαιτούσαν ειδικούς χειρισμούς για να διατηρηθεί η συμβατότητα με το Application Binary Interface (ABI) του Συστήματος Χρόνου-Εκτέλεσης της Erlang (Erlang Run-Time System) και κατ' επέκταση η ενσωμάτωση της δουλειάς μας να γίνει διατηρώντας την υπάρχουσα αρχιτεκτονική της Εικονικής Μηχανής. Γι' αυτούς τους λόγους χρειάστηκε να αλλάξουμε το Εξάρτημα Παραγωγής Κώδικα της LLVM, εφαρμόζοντας τους κανόνες που επέβαλλε το ABI στον παραγόμενο κώδικα.

Στο κεφάλαιο της αξιολόγησης αναλύουμε λεπτομερώς την τρέχουσα πολυπλοκότητα και την επίδοση του νέου οπίσθιου τμήματος που υλοποιήσαμε με χρήση της LLVM για την οικογένεια επεξεργαστών AMD64. Οι χρόνοι εκτέλεσης των μετρο-προγραμμάτων που μεταγλωττίστηκαν στο δικό μας οπίσθιο τμήμα ήταν συγκρίσιμοι με εκείνους των προγραμμάτων που μεταγλωττίστηκαν στο υπάρχων οπίσθιο τμήμα του HiPE, και σημαντικά πιο μικροί από εκείνων της Εικονικής Μηχανής BEAM και της Erjang, η οποία είναι μια εικονική μηχανή για την Erlang βασισμένη στην Εικονική Μηχανή της Java (JVM). Η πολυπλοκότητα του τμήματός μας αποδείχθηκε σημαντικά μικρότερη. Ιδιαιτέρως αν λάβει κανείς υπόψιν του ότι, με σχετικά απλές επεκτάσεις, το οπίσθιο τμήμα που υλοποιήσαμε μπορεί να καλύψει όλες τις αρχιτεκτονικές επεξεργαστών που σήμερα υποστηρίζει ο HiPE. Αρκετές βελτιώσεις έχουν ήδη προγραμματιστεί ως αντικείμενο μελλοντικής εργασίας.

Λέξεις Κλειδιά

Erlang, μεταγλωττιστής HiPE, μεταγλώττιση κώδικα μηχανής, βιβλιοθήκη LLVM, οπίσθιο τμήμα, συμβολική γλώσσα υψυλού επιπέδου, μετάφραση ενδιάμεσης απεικόνισης, βελτιστοποιήσεις χρόνου μεταγλώττισης

Abstract

Existing open-source compilers are based on old code generation technology, with code bases that are difficult to learn and hard to change, and share no code between each other. The Low Level Virtual Machine (LLVM) is a state-of-the-art compiler infrastructure providing a set of reusable components that implement the best known techniques focusing on compile time and performance of the generated code. The goal of LLVM is to provide modular components for building high quality compilers for many different languages.

This thesis describes the current architecture, design decisions and implementation details of a new back end for HiPE, the native code compiler of Erlang/OTP, that targets the LLVM infrastructure. One of HiPE's intermediate representation, called Register Transfer Language (RTL), was found to have a very straightforward translation to LLVM assembly. However, there were a few subtle points, such as the calling convention, the exception handling mechanism and the garbage collection, that needed to be handled in order to retain Application Binary Interface (ABI) compatibility with the Erlang Run-Time System (ERTS) and integrate our work in the existing Virtual Machine architecture. For these reasons we patched the LLVM Code Generator and imposed the appropriate rules on the generated binary code.

In the evaluation we detail the current complexity and performance of the new LLVM back end for the AMD64 architecture. The run-time performance was found to be comparable with HiPE and signifficantly faster than BEAM virtual machine and Erjang, a virtual machine for Erlang based on the Java Virtual Machine (JVM). The complexity of the LLVM back end proved to be far simpler; especially, if you take into consideration that, with rather plain extensions, it can support all hardware architectures that HiPE currently targets. Various performance improvements are planned for future work.

Keywords

Erlang, HiPE compiler, native code compilation, LLVM framework, back end, high-level assembly, intermediate representation transformation, compile-time optimizations

Ευχαριστίες

Θα ήθελα να πω ένα μεγάλο ευχαριστώ στον Κώστη Σαγώνα για την διαρκή υποστήριξη και πολύτιμη καθοδήγηση, η οποία συνέβαλε καθοριστικά στη διαμόρφωση αυτής της διπλωματικής εργασίας, καθώς και για την εμπιστοσύνη και σεβασμό τον οποίο μου έδειξε. Επίσης, θα ήθελα να ευχαριστήσω τον Νίκο Παπασπύρου για την πολύ σημαντική βοήθεια την οποία μας προσέφερε.

Χρωστάω ένα μεγάλο ευχαριστώ στους γονείς μου για την υποστήριξη που μου έχουν προσφέρει και την εμπιστοσύνη που έδειξαν σε κάθε επιλογή μου.

Θα ήθελα ακόμα να ευχαριστήσω το φίλο και συνεργάτη Γιάννη Τσιούρη. Η συνεργασία μας καθέστησε εφικτή την εκπλήρωση αυτής της διπλωματικής. Επίσης η διαρκής παρότρυνσή του ήταν καθοριστική στο να ασχοληθώ με τον προγραμματισμό καθώς και το ελεύθερο λογισμικό.

Τέλος, θέλω να ευχαριστήσω όλους τους φίλους οι οποίοι με στηρίζουν και στέκονται δίπλα μου σε κάθε περίσταση. Ιδιαίτερα θέλω να ευχαριστήσω τον κύκλο των φίλων: Άννα Βήχου, Βίκυ Βλάχου, Πάνο Μάντζιο και Νίκολας Μουσιώνη. Η παρουσία τους στη ζωή μου ομορφαίνει την καθημερινότητά μου.

Χρήστος Σταυρακάκης

Αρχικά, θα ήθελα να ευχαριστήσω τους γονείς μου, Φώτη και Έφη Τσιούρη, και την αδερφή μου, Ευθαλία, που ήταν πάντα δίπλα μου ανέχοντας τις “ιδιοτροπίες” μου και στηρίζοντας κάθε μου επιλογή. Όλα όσα μου προσέφεραν με έκαναν αυτό που είμαι σήμερα.

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Κωστή Σαγώνα, για την έμπνευση του θέματος της διπλωματικής, τη διαρκή καθοδήγηση και τη συνεργασία μας κατά τη διάρκεια του τελευταίου έτους. Επιπλέον, για την εμπιστοσύνη που μου έδειξε από την πρώτη στιγμή ανάθεσης της διπλωματικής.

Θα ήθελα επίσης να ευχαριστήσω τον Νίκο Παπασπύρου, ο οποίος με ενέπνευσε με τη συμπεριφορά του και το ήθος του και συνέβαλλε στην απόφασή μου να ασχοληθώ με την Πληροφορική από το πρώτο έτος των σπουδών μου.

Τέλος, θέλω να πω ένα μεγάλο ευχαριστώ σε όλους τους φίλους μου που μου στάθηκαν τα τελευταία χρόνια και μου έδειξαν πως σε όλα τα πράγματα χρειάζεται ένα μέτρο. Χρήστο, ένας από αυτούς είσαι κι εσύ.

Γιάννης Τσιούρης

Contents

Περίληψη **5**

Abstract **7**

Ευχαριστίες **9**

Contents **12**

List of Figures **13**

List of Listings **15**

1 Introduction **17**

2 Background **19**
2.1 Compiler Design 19
2.2 The HiPE Compiler 20
2.2.1 Phases in the compiler 21
2.2.2 Interface issues with Erlang/OTP 23
2.3 Low Level Virtual Machine 28
2.3.1 LLVM Assembly Language 29
2.3.2 LLVM Type System 31
2.3.3 LLVM Instruction Set 32
2.3.4 Other projects using LLVM 34

3 The LLVM back end **37**
3.1 Pipeline Design 37
3.2 LLVM Representation 40
3.3 Generation of LLVM assembly 40
3.3.1 Handling RTL Virtual Registers 40
3.3.2 Handling Immediate Values 42
3.3.3 Handling RTL Instructions 43
3.3.4 Calling Convention 44
3.3.5 Calls with Stack Arguments 45
3.3.6 Garbage Collection 45
3.3.7 Exception Handling 47
3.3.8 Frame Management 48
3.4 Rest Phases 49
3.4.1 LLVM Assembler 49
3.4.2 LLVM Optimizer 49

3.4.3 LLVM Compiler . . . 49
3.4.4 Object File Generation . . . 50
3.4.5 Object File Parsing . . . 50

4 Evaluation **51**
4.1 Current State of LLVM back end . . . 51
4.2 Performance of LLVM back end . . . 51
4.2.1 Results . . . 52
4.2.2 Performance Analysis . . . 53
4.3 Complexity of Implementation . . . 58

5 Conclusion **61**
5.1 Concluding remarks . . . 61
5.2 Future work . . . 62

Bibliography **63**

List of Figures

2.1 The Three Components of a Three-Phase Compiler 19
2.2 Structure of a HiPE-enabled Erlang/OTP system 21
2.3 Call stack for function `bar/L` in the call chain `foo/K` $\rightarrow$ `bar/L` $\rightarrow$ `baz/M` . . 23
2.4 Call stack "snapshots" when `f` calls `g` (`f` $\rightarrow$ `g`) and `g` tail-calls `h` (`g` $\stackrel{tail}{\rightarrow}$ `h`). . . 24
2.5 Icode CFGs for functions `foo` and `bar` from Listing 2.1. 25
2.6 The stack frame layout when function `bar/7` calls `zap/0` and the corresponding stack descriptor. 26

3.1 The new LLVM back end inside the Erlang/OTP system 37
3.2 The LLVM component 39
3.3 RTL CFG of a function calling `bar/1`, protected with an exception handler. 48
3.4 LLVM assembly of a function calling `bar/1`, protected with an exception handler. 48

List of Listings

2.1 An exception thrown by `bar` is caught by `foo` 25
2.2 C example: factorial 30
2.3 LLVM assembly for factorial (Listing 2.2) 30
3.1 Erlang implementation of function `length` 43
3.2 RTL example: Length of a list. The ”strange” numbers in the example are tags and tagged values. Addition is performed either by the ’add’ instruction or by the ’+’ BIF, depending on the type of the value. 44
3.3 LLVM assembly for handling a GC root 46

Chapter 1

Introduction

Programming languages are notations for describing computations to people and machines. For these computations to be able to run on machines, the programs written in some high-level programming language should be first transformed to executable machine code. This is exactly what *compilers* do.

Compilers are complex software systems that translate source code written in a high-level programming language (the *source* language) into a lower level language (the *target* language), e.g. another programming language, assembly language or machine code. A compiler operates as a sequence of phases, each of which transforms the source program from one intermediate representation to another preserving the meaning of the program that is compiled. These phases are conceptually grouped in three parts: the *front end*, the *optimizer* and the *back end*. In the front end, the program is checked as far as the programming language syntax and semantics is concerned, errors are spotted and informative messages are provided to the programmer and, finally, an intermediate representation is built. The optimizer transforms the intermediate representation into functionally equivalent, yet faster, forms. The back end, constructs the desired target program from the intermediate representation and the information collected during the front end phases.

In this thesis we will be looking at the generation of executable code for the Erlang programming language. We have worked on the Open Telecom Platform (OTP), an industrial strength system that is the primary implementation of Erlang and provides a framework to structure Erlang systems offering robustness and fault-tolerance together with a set of tools and libraries. We will be evaluating an implementation of a new back end target for High-Performance Erlang (HiPE), the native code compiler of Erlang/OTP.

For our work we have used Low Level Virtual Machine (LLVM), a state-of-the-art compiler infrastructure supporting both static and dynamic compilation of arbitrary programming languages. It is an "umbrella" project consisting of a number of tools and libraries for creating high quality compilers. In detail, we have targeted its high performance code generator in order to generate Application Binary Interface (ABI) compliant machine code for the Erlang Run-Time System (ERTS).

The work of this thesis aims at providing multiple back ends for HiPE with the use of the LLVM compiler infrastructure. The ultimate goal is to improve both the performance and code maintenance of HiPE back ends.

The rest of the thesis is organised as follows. In Chapter 2 we give an overview of the HiPE

system and the LLVM framework. Chapter 3 is the main chapter of the thesis, where we present the most critical design decisions and the implementation of the new Low Level Virtual Machine back end for HiPE. In Chapter 4 we evaluate the new back end in terms of complexity and performance. Finally, in Chapter 5 we present our concluding remarks and future work.

Chapter 2

Background

2.1 Compiler Design

Over the last twenty years, the advent of microprocessor technology along with the evolution of high-level programming languages has resulted in complicating the design and implementation of compilers. Situated between the modern programming language and the architecture, the compiler is responsible for making the application perform as well as possible on a target machine.

The most popular design pattern for a traditional static compiler is the *three-phase* design whose major components are the front end, the optimizer and the back end (Figure 2.1). As already stated, the front end asserts that all language-specific requirements (i.e. syntax and semantics) are met and parses the input code to an *intermediate representation* (IR), usually an Abstract Syntax Tree (AST). The AST is optionally converted to a new representation for optimization and the optimizer and the back end are run on the code .

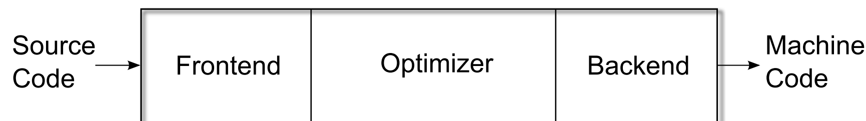

Figure 2.1: The Three Components of a Three-Phase Compiler

The optimizer is responsible for doing a broad variety of transformations to try to improve the code's running time, for example by eliminating redundant computations, and is usually more or less independent of language and target. This usually involves more than one IR, with each transformation into a new IR simplifying the code and representing it at a lower abstraction level, closer to the machine architecture on which the compiled program will run on. Then, the 3rd component is invoked. The back end (also known as the *code generator*) maps the code onto the target instruction set. Not only is it responsible for making *correct* code, but also *good* code that often takes advantage of features of the underlying architecture. Common parts of a compiler back end include instruction selection, register allocation and instruction scheduling.

The most important feature of this design is that it consists of modular components and it is rather easy to support more than one source language or target architecture. The

only requirement would be to have a common representation in the optimizer, and a front end can be written for any language that can compile to it, and a back end can be written for any target. This model applies equally well to interpreters and Just-In-Time (JIT) compilers. The Java and .NET virtual machines are implementations of this model.

Despite the benefits of the three-phase design, there has been little success in fully utilising the advantages of the model in the past years (see Brown and Wilson's book [4, Chapter 11]). In practice, various implementations of high-level programming languages, such as Python, Ruby, Java and Haskell, share no code. The basic problem lies on specifying a common intermediate representation that can efficiently support the wide variety of programming languages which exist. This IR needs to be of fairly low level in order to be eligible to express the significantly different semantics of these programming languages and support aggressive optimizations as higher level representations do. Furthermore, it should be designed so as to support high-level services, such as garbage collection and exception handling, in a universal and portable manner in the IR.

A notable example that has achieved great progress over the past few years on this field is Low Level Virtual Machine (LLVM). This increasing interest on LLVM is further supported by the number of projects [28] and publications [29] that use or build on it lately. LLVM is basically a compiler infrastructure; it is a very straightforward implementation of the three-phase design pattern that utilises a fairly low-level RISC-like instruction set with a strict type system designed with many important optimizations, such as lightweight runtime optimizations, cross-function/inter-procedural optimizations, whole program analysis, and aggressive restructuring transformations, etc., in mind. Unlike the front end and back end of the compiler, the optimizer is not constrained neither by a specific source language nor a specific target machine. LLVM IR is both well specified and the *only* interface to the optimizer. LLVM will be discussed further in Section 2.3.

2.2 The HiPE Compiler

HiPE (High Performance Erlang) is a native code compiler for Erlang. Erlang is a concurrent functional programming language designed for developing large-scale, distributed, fault-tolerant systems. The primary implementation of Erlang, the Erlang/OTP, is by default based on a virtual machine interpreter (BEAM); but with HiPE it is also allowed for the user to natively compile those parts of the code where the speedups are worth the larger code size and longer compilation times, and also keep the non-time-critical part of the application in interpreted code (i.e. bytecode).

HiPE was an ASTEC[1] project at the Department of Information Technology[2] (division of Computing Science) of Uppsala Univerity, aimed at efficiently implementing concurrent programming systems using message-passing in general and the concurrent functional language Erlang in particular [23]. Since October 2001 the HiPE system is fully integrated in Ericsson's Open Source Erlang/OTP system. The HiPE compiler currently has backends for ARM, SPARC V8+, x86, AMD64, PowerPC and PowerPC64.

In this section we will mostly present an overview of the design of the HiPE compiler and how it is integrated in the Erlang Run-Time System (ERTS) focusing on areas which are relevant to the implementation of a Low Level Virtual Machine (LLVM) back end.

[1] `http://www.astec.uu.se/`
[2] `http://www.it.uu.se/`

2.2.1 Phases in the compiler

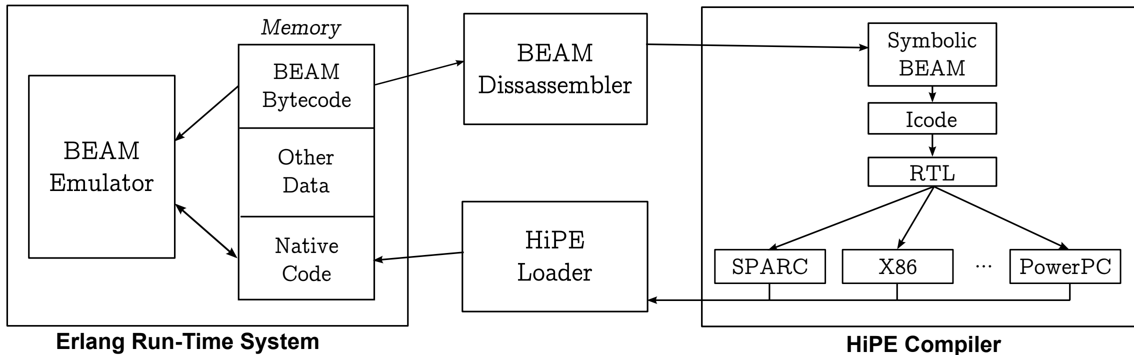

Figure 2.2: Structure of a HiPE-enabled Erlang/OTP system

The compilation process in HiPE starts by disassembling the bytecode generated by the BEAM compiler, and representing it in a symbolic version. The overall structure of the HiPE system is shown in Figure 2.2. HiPE's pipeline uses various intermediate representations ending in assembly code [15].

- **BEAM**: The BEAM code is just a transformation of Erlang source code to virtual machine code generated by triggering the Erlang compiler of the underlying Erlang/OTP system. BEAM operates on a largely implicit heap and call-stack, a set of global virtual registers and a set of slots in the current stack frame.

- **Icode**: Icode is an idealised Erlang assembly language with a minimal instruction set of only 16 instructions. The Icode IR assumes an infinite number of registers and an implicit stack. Registers are preserved around function calls and all bookkeeping operations, such as memory management and process scheduling, are implicit.

  BEAM is translated to Icode mostly one instruction at a time. However, some obviously poor sequences of virtual machine code are peephole-optimised into more efficient Icode sequences. Common operations, such as fetching an element from a tuple or pattern-matching, are in-line expanded into fetches and tests.

  Temporaries are also renamed through conversion to *static single assignment* (SSA) form [34] to avoid false dependencies between different live ranges. This form enables many optimizations, such as constant and copy propagation, constant folding and dead-code removal.

- **RTL**: RTL is a generic three-address register transfer language. RTL itself is target-independent, but the code is target-specific due to references to target-specific registers and primitive procedures. The instruction set is similar to MIPS consisting of 27 instructions. RTL has tagged registers for proper Erlang values and untagged for arbitrary machine values, such as an address or a raw integer. To simplify the garbage collector interface, function calls only preserve live *tagged* values.

  In the translation to RTL a large number of operations (e.g. arithmetic, data construction, tests) are in-lined. Data tagging and untagging are made explicit, data accesses and initialisations are turned into loads and stores. Icode-level `switch` instructions for switching on basic values are translated into code that implements

the switches. In this form, optimizations like common subexpression elimination and constant propagation and folding are performed. Moreover, stack and exception handling code is expanded into explicit code.

- **Symbolic target-specific assembly**: This intermediate representation is just a simple abstraction of the assembly of the targeted architecture (SPARC, X86, PowerPC, etc.). It differs from the RTL in a way that it better represents architecture-specific instructions, e.g. in x86 IR memory operands exist which are described by simple addressing modes (base register plus offset).

The next step in the compilation pipeline is that of **register allocation**. In this step, temporaries (virtual registers) are mapped to actual machine registers. Every temporary that remains unallocated is mapped to a specific stack slot during the subsequent phase of frame management. Coloring graph-based register allocation is typically performed in a loop. First an attempt is made to allocate registers for the code. If this fails because some temporaries were spilled (could not be assigned to registers), the code is rewritten under the assumption that those temporaries are in memory, and the process continues with a new allocation attempt. Eventually, however, the allocation will succeed. The HiPE system has several register allocators implemented: an *iterated register coalescing* allocator [13], a *Briggs-style graph colouring* register allocator [3], a *linear scan* register allocator [27, 33] and, lastly, a *naive* register allocator.

After that, stack frames are introduced to the code. The **frame management** pass is responsible for:

- mapping spilled temporaries to stack slots and rewriting uses of these temporaries as memory operands in the stack frame,
- adding code to the function prologue in order to check for stack overflow and setting up the call frame (the frame size and maximal stack usage are computed and taken into consideration),
- creating stack descriptors for each call site, describing which stack slots correspond to live temporaries (using the result of the liveness analysis) and whether the call is in the context of a local exception handler, and
- generating code, at each tail call, to shuffle the actual parameters to the initial portion of the stack frame.

During most phases of the compiler, the code is represented in the form of a *Control Flow Graph* (CFG). Before translating to native code, the CFG must be linearised by ordering the basic blocks and redirecting jump instructions accordingly. The **linearization** step is responsible for performing this ordering while taking into account the likelihood of a conditional jump being taken or not, and the static branch prediction algorithm used in hardware. The translation from CFG to linear code generates the most likely path first and then appends the code for the less likely paths. This is a crucial phase of the compiler as it can lead to generating mature code with considerably better performance.

Finally, the custom **assembler** converts the final symbolic representation to binary machine code and produces a loadable object file, with the machine code, constant data, a symbol table and the patches needed to relocate external references, ready to be loaded in the runtime system.

2.2.2 Interface issues with Erlang/OTP

It was an early design decision for HiPE to be based on the Erlang/OTP that was already an industrial strength system, widely used in real world applications. To achieve that, the HiPE compiler had to implement all features of Erlang. Furthermore, it had to extend the Erlang/OTP runtime system to permit Erlang processes to execute both interpreted and native machine code while maintaining the semantics of code replacement of the language (i.e. the ability to upgrade code at runtime, without affecting processes currently executing the old version of that code). In this part, we describe some features of Erlang and the Erlang/OTP system and how they were implemented in the compiler.

Stack frame layout

The stack frame of a function is composed of two parts: a *fixed-size* part at the top for the caller-save registers and spilled temporaries and a *variable-size* part at the bottom for pushing the outgoing parameters in calls (see Figure 2.3). On entry, the function first checks that enough stack space is available for the largest possible frame, calling a runtime system primitive if this is not the case, and the fixed-size part is set up. The main benefit of fixed-size frames is their low maintenance cost. On the other hand, they may contain dead or uninitialised stack slots, which complicate garbage collection and exception handling.

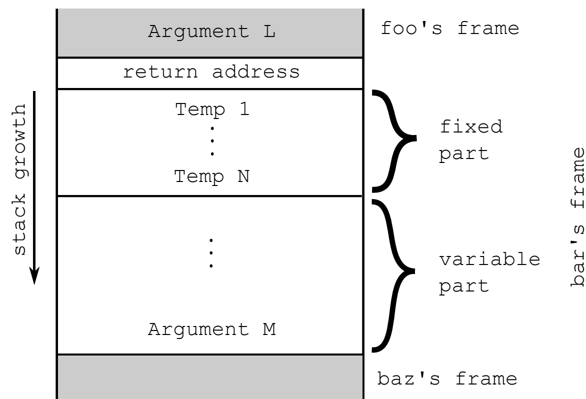

Figure 2.3: Call stack for function `bar/L` in the call chain `foo/K` → `bar/L` → `baz/M`

Tail calls

Erlang, like most functional programming languages, relies on tail-recursive function calls for expressing iteration. In order to implement this in constant stack space, which is a requirement for the language, the HiPE compiler generates special code that shuffles the stack contents in tail calls.

To illustrate how calls and tail calls are implemented by HiPE, assume that `f` calls `g`, `g` tail-calls `h` and `h` finally returns to `f`. Figure 2.4 shows the stack layout changes in this process. At first, state (a), `f`'s frame is loaded on the stack. Then, `f` prepares the stack for the `g`-call by pushing the arguments and executes the `call`. The `call` instruction is responsible for storing `f`'s return address and pushing `g`'s frame on the stack. On x86, the

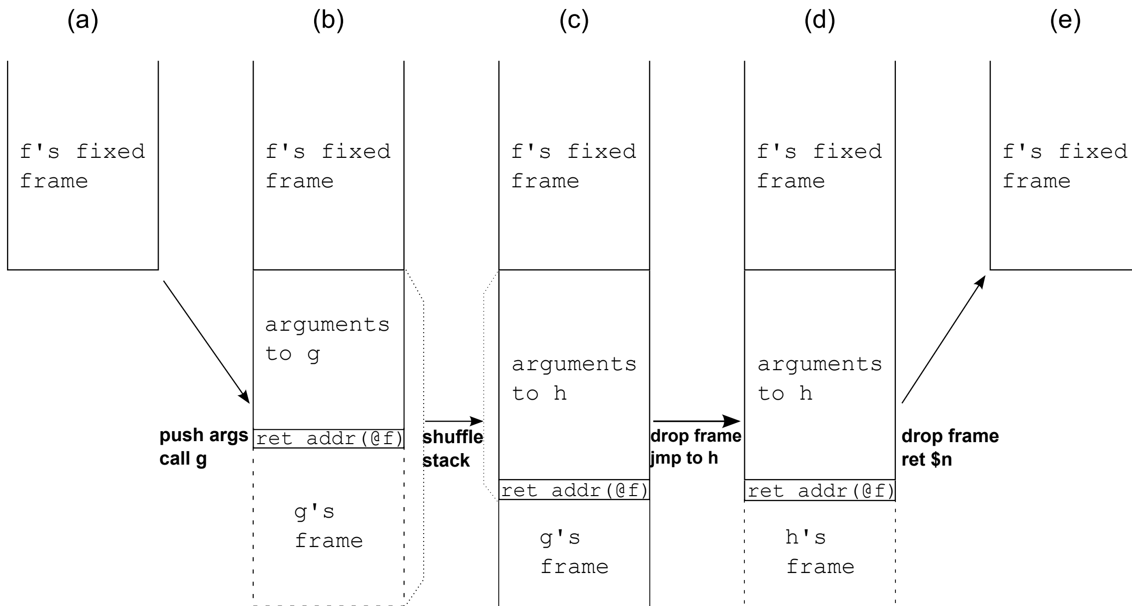

Figure 2.4: Call stack "snapshots" when `f` calls `g` (`f` $\rightarrow$ `g`) and `g` tail-calls `h` (`g` $\stackrel{tail}{\rightarrow}$ `h`):
$(a) \Rightarrow (b)$: `f` pushes arguments to `g` and executes the call.
$(b) \Rightarrow (c)$: `g` prepares the stack for tail-calling `h`.
$(c) \Rightarrow (d)$: `g` tail-calls `h` by dropping its frame and jumping to `h`.
$(d) \Rightarrow (e)$: `h` returns to `f` by dropping its stack arguments and frame.

`call` instruction pushes the return address on the stack, while on SPARC, it stores it on a register and the loader is responsible for pushing the return address on the stack. After that, `g`'s code starts executing and the parameters for `h` are evaluated. `g` shuffles the stack to overwrite the argument area and possibly parts of its frame with the new parameters, leading to state (c). Then, `g` tail-calls `h` by dropping its frame and jumping to `h`. Notice that in state (d) the stack is exactly as if `f` called `h` directly. Eventually `h` returns to `f` by removing its stacked arguments and dropping its frame, leading state (e).

In this scheme, it is the callee's responsibility to pop the stacked arguments before the execution returns to the caller's code. In the presence of tail calls, the caller (`f` in this example) does not know which function finally returns to it, and thus does not know how many parameters there are on the stack upon return. Therefore, the caller cannot deallocate the stacked parameters, but the returning function *can* since it knows how many parameters *it* takes. Note that this calling convention is the opposite of the common C calling convention.

The disadvantage on HiPE's handling of tail-calls is that the stack shuffle step introduces some complexity on the runtime. Fortunately, the shuffling is only needed when both the caller and the callee have arguments on the stack. On the SPARC which passes the first 16 arguments in registers this situation is very rare, while on x86, where at most 5 arguments can be passed in registers, this shuffling occurs more often.

Exception handling

In Erlang, an exception thrown in one function `bar`, can be caught by an exception handler in a function `foo`, calling `bar` (see Listing 2.1). This return from `bar` with an invoked exception is handled in HiPE by adding the basic blocks implementing the handler and one "exception" edge in the CFG of the caller for calls that might fail to it (see Figure 2.5).

```
foo(X) ->
    %% The catch instruction will set up an exception handler which in
    %% this simplest form will just turn an exception into an Erlang term.
    catch bar(X).

bar(X) ->
    %% This operation will throw an exception if X is not a number.
    X + 42.

%% Example execution:
%% > foo(3).
%% 45
%% > foo(hello).
%% {'EXIT', {badarith, [...]}}
```

Listing 2.1: An exception thrown by `bar` is caught by `foo`.

The compiler inserts some code in edges between calls and exception handlers in order to move exception values to the right local temporary. The loader recognises calls within exception handlers and registers their address together with the address of the exception handler in a stack descriptor (see next section). This way there is no runtime cost for setting up an exception handler. When an exception is thrown, the stack map is used while traversing the call stack and if a return address has an exception handler the control is transfered to the handler.

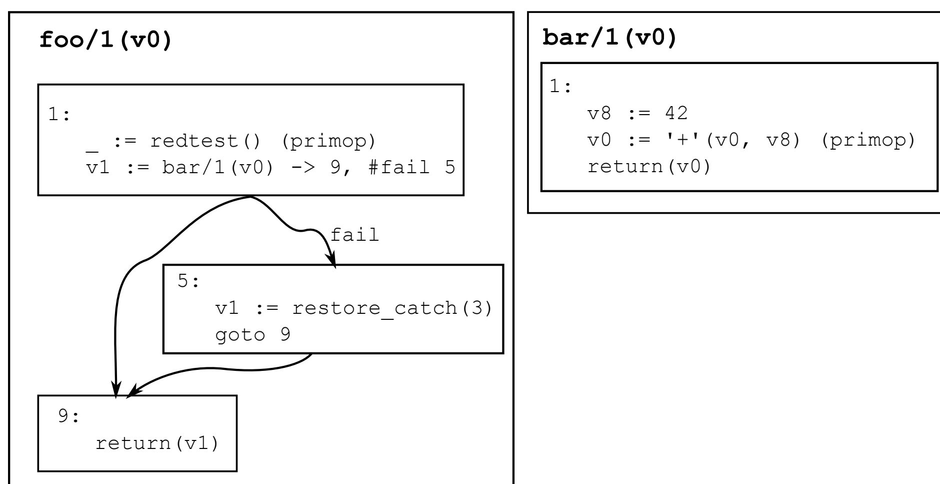

Figure 2.5: Icode CFGs for functions `foo` and `bar` from Listing 2.1.

Stack Descriptors

In order to support precise garbage collection and exception handling, HiPE must provide the necessary information to the runtime system, in order to be able to traverse the stack frames. This is achieved with the use of stack descriptors. HiPE constructs a stack descriptor for each call site. The stack descriptor contains all necessary information about the call:

- the caller's exception handler
- the caller's fixed frame size, excluding incoming arguments
- the caller's stack arity (i.e. the number of arguments that are passed to the stack)
- the indices (from SP) of the live words in the caller's frame
- the return address of the call site

In Figure 2.6 we can see an example of a stack descriptor. Function `bar/7` calls `zap/0`. Function `bar/7` has no exception handler, the frame size is 5 (3 stacked arguments plus 2 function-local variables) and there are 2 live words while calling `zap/0` in indices 0 and 2, relative to the stack pointer. Finally the stack arity is 3, because function `bar` has arity 7, but the first 4 arguments are passed in registers.

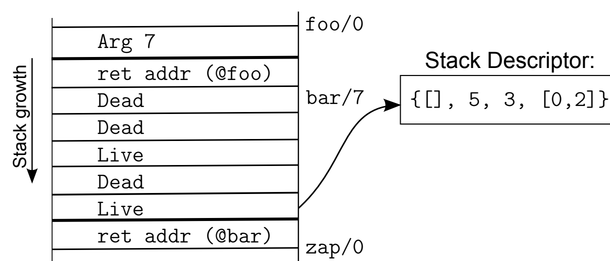

Figure 2.6: The stack frame layout when function `bar/7` calls `zap/0` and the corresponding stack descriptor. Notice that in the AMD64 back end 4 arguments are passed in registers, so in this case the rest 3 will be passed on the stack. Also there are 3 dead variables and 2 live during the call to `zap/0`.

Garbage Collection

HiPE implements a *precise two-phase generational garbage collector*. It uses safe point strategy, with post-call safe points. The compiler emits code which checks if the heap pointer goes beyond the heap limit, and calls the garbage collector in that case [26]. With the information from the stack descriptors, the garbage collection is able to traverse the call stack and identify the live roots.

Mode switching

Each Erlang process has two stacks, one for interpreted code (`estack`) and one for native code (`nstack`). As explained in *Concurrent Programming in Erlang* [1], this simplifies

garbage collection and exception handling since each stack contains only frames of a single type; control flow between these two modes is handled by a *mode-switch* interface.

Since HiPE compiles individual functions to native code, a mode-switch occurs whenever there is a transfer of control from a function to another and the two functions are in different modes. Thus, mode-switches occur at call and return sites, when an exception is thrown and the most recent handler is in a different mode and when a process is suspended and the next process to run executes in another mode. The implementation uses linker-generated *proxy code stubs* and *software trap return addresses* to trigger the appropriate mode-switches when invoked.

Two important design decisions and properties of this interface are that it preserves tail-recursion (i.e. no sequence of consecutive mode-switching tail calls grow either stack by more than a constant) and that imposes no runtime overhead on same-mode calls and returns (i.e. from native to native or from BEAM to BEAM). For more details of how this is achieved please refer to Erik Stenman's PhD thesis [35, Chapter 3] and [].

Built-in functions

The Erlang/OTP system is shipped with a number of built-in functions (*BIF*s) and a big standard library. Some of these functions are implemented in Erlang, but there are functions implemented in C (mostly in the runtime system) and, even, in assembly (mostly calls to special primary operators), e.g. calls invoking the GC, increasing the native stack, forcing a mode-switches or suspending a process.

In order for these calls to be able to be executed from native code, they should be compiled using C's calling convention and the loader should be able to find the address of the C code at load time. Some of these BIF-calls are recognised by the translation to RTL and inlined directly in RTL code.

Process switching

The Erlang/OTP system takes advantage of a multi-core or multi-CPU computer by running one or more scheduler threads (typically, the same as the number of cores). Each scheduler thread schedules Erlang processes by giving a time slice to each process. This is implemented in the system with a *reduction counter*: The process starts with a number of reductions to execute and at each call this number is decremented. When the number of reductions reaches zero the process is suspended. Then the scheduler chooses which process to execute next.

Code loading

As already described, Erlang requires the ability to upgrade code on-the-fly without affecting processes currently executing the old version of that code (*hot-code loading*).

The underlying Erlang runtime system maintains a global table of all loaded modules. Each module descriptor contains a name, a list of exported functions and the locations of its *current* and *previous* code segments. At a remote function call

(`module:function(parameters...)`) a lookup on that table is performed and if no entry is found an error handler is invoked.

In native code, each function call is implemented as a machine-level call to an *absolute* address. When the caller's code is being linked, the linker tries to initialize the code to directly invoke the callee. However, if the callee has not been loaded yet, the linker will direct the call to a stub which performs the appropriate error handling. If the callee exists, but *only* in emulated code (bytecode), the linker directs the call to a stub which in turn will invoke the virtual-machine emulator.

To handle hot-code loading and dynamic compilation at runtime, the linker also maintains information about all call sites in native code. This information is used along with appropriate proxy stubs (`trap-to-emulated`, `trap-to-native`) for *dynamic code patching* when a native mode function calls another one in emulated mode, a new version of a native mode function is loaded or a module is unloaded. For more details about the implementation, refer to Erik Stenman's PhD thesis [35, Chapter 3].

Pattern matching implementation

In Erlang, as in many other functional programming languages, it is very common to do pattern matching on some values. This is usually translated by the front end of the compiler to some intermediate representation `switch` instruction that compares the contents of a temporary to a set of constants and jumps to the label corresponding to the matching constant. A default label that is used when no constant matches is also given.

In the HiPE compiler, the `switch` instructions are translated into sequences of lower level instructions during the translation from Icode to RTL. If the set of constants is too sparse the switch is split into several smaller switches [2]. There are then several ways these switches may be translated to: 1) as an in-lined binary search (used when the number of constants is low), 2) as a direct jump table (for large and dense sets of small integers), or 3) as a binary search in a table. Atoms are problematic since their runtime values differ between invocations of the runtime system, so switches on atoms are translated into semi-symbolic code which is finalised by the code loader.

2.3 Low Level Virtual Machine

The Low Level Virtual Machine (LLVM) is an open source, mature optimizing compiler framework that began as a research project at the University of Illinois by *Chris Arthur Lattner* in 2000 as part of his Master thesis [17]. It provides a modern source- and target-independent optimizer along with a very efficient code generator that can be used for static or dynamic (just-in-time) compilation targeting more than 16 CPU and microprocessor architectures, e.g. ALPHA, ARM, Blackfin, MIPS, MIPSEL, PowerPC32, PowerPC64, PTX 32-bit, PTX 64-bit, SPARC, SPARC V9, x86, x86-64, XCore and more. We have used LLVM in this thesis to create a new back end for the HiPE compiler.

The main reason why we decided to use LLVM is that it is a state-of-the-art, very well designed platform for back end code generation and optimization work. It has a very active community of developers and the code base is well written and easy to change. Furthermore, LLVM is available under the *University of Illinois/NCSA Open Source License*

[20] and thus not only is it a great choice for an academic research project but also allows commercial products to be derived from it. Today, numerous projects are using LLVM either as a static or as a just-in-time compiler, as well as just for static code analysis. Some of these projects are described in Section 2.3.4.

Due to the aforementioned reasons, we decided to implement the new back end based on LLVM; let us now look through the assembly language provided by the LLVM framework in order to obtain some insight on what the requirements are in order to compile a language to LLVM Assembly.

2.3.1 LLVM Assembly Language

LLVM defines a high-level portable assembly language, providing abstraction between assembly and source language. It is basically a common, low-level code representation in Static Single Assignment (SSA) form [34], with several novel features: a simple, language-independent type-system that exposes the primitives commonly used to implement high-level language characteristics; an instruction for typed address arithmetic; and a simple mechanism that can be used to implement the garbage collection and exception handling features of high-level languages uniformly and efficiently. LLVM IR aims to be a "universal IR" able to express many different characteristics and aggressively optimise arbitrary high-level languages.

The LLVM code representation is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler), and as a human readable assembly language representation. This allows LLVM to provide a powerful intermediate representation for efficient compiler transformations and analysis, while providing a natural means to debug and visualise the transformations. In this section we will describe the human readable representation and notation.

At first, let us elaborate a bit on what Static Single Assignment form means. In compiler design, SSA form is a property of an intermediate representation which says that once a virtual (pseudo-) register is assigned, it becomes immutable. LLVM assembly provides an infinite number of virtual registers, abstracting away actual hardware registers, and a special phi (Φ) instruction to handle control flow in SSA form. A key design point of an SSA-based representation is how it represents memory. In LLVM, *no* memory locations are in SSA form, which makes things rather simpler. The reason why we want the IR in SSA form is that it either enables or strongly enhances various data flow optimizations, such as constant propagation, sparse conditional constant propagation, dead code elimination, global value numbering, partial redundancy elimination, strength reduction and register allocation. It is better to see LLVM assembly in detail through an example. Listing 2.2 implements a simple factorial program in C code, while listing 2.3 depicts the equivalent LLVM program.

LLVM programs are composed of *Module*s, each of which is a translation unit of the input programs. Each module consists of meta information, global variables, external symbol definitions and function definitions. Meta information include definitions of the endianess, the pointer size and the alignment of the data layout. Global variables define regions of memory allocated at compilation time instead of runtime. They are pointers to data with global scope and are prefixed with the @ symbol, as functions. This should be compared

```
int factorial(int X) {
  if (X == 0) return 1;
  return X*factorial(X-1);
}
```

Listing 2.2: C example: factorial

to the local temporaries (virtual registers) spotted inside a function and denoted by the % prefix. External declarations define symbols that may be used in the current module but are meant to be linked later when compiled to a native object file.

```
define i32 @factorial(i32 %X) {
  %1 = alloca i32
  %2 = alloca i32
  store i32 %X, i32* %2
  %3 = load i32* %2
  %4 = icmp eq i32 %3, i32 0
  br i1 %4, label %5, label %6

; <label>:5                          ; preds = %0
  store i32 1, i32* %1
  br label %12

; <label>:6                          ; preds = %0
  %7 = load i32* %2
  %8 = load i32* %2
  %9 = sub i32 %8, i32 1
  %10 = call i32 @factorial(i32 %9)
  %11 = mul i32 %7, i32 %10
  store i32 %11, i32* %1
  br label %12

; <label>:12                         ; preds = %6, %5
  %13 = load i32* %1
  ret i32 %13
}
```

Listing 2.3: LLVM assembly for factorial (Listing 2.2)

A function definition contains a list of *basic blocks*, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and explicitly ends with a terminator instruction, such as a branch (e.g. `br`) or function return (`ret`). The first basic block in a function is not allowed to have predecessor basic blocks (i.e. there cannot be any branches to the entry block of a function).

In Listing 2.3, we can see one function definition (`@factorial`) that takes one 32-bit integer argument and returns a 32-bit integer. This function consists of 4 basic blocks, `block 0`, `block 5`, `block 6` and `block 12`. The results of all instructions are assigned to *different* virtual registers (i.e. `%1`-`%13` and `%X`) (SSA form). All operations are annotated with type information, e.g. in line 4 a 32-bit integer value (`i32 %X`) is stored to a similar pointer (`i32* %2`) already allocated on the stack in the previous line (`alloca i32`).

2.3.2 LLVM Type System

The LLVM type system is one of the most important features of the intermediate representation. It is thoroughly designed to give enough information to the LLVM optimizer and code generator to produce efficient code without having to do extra analysis on the side before the transformation. See Table 2.1 below for a brief listing of the type system.

LLVM is a strictly typed representation that requires all types to be explicitly stated and does not do any type inference based on the types involved in an instruction. It also requires the programmer to handle all type conversions using explicit casts. This type information enables a broad class of *high-level* transformations on *low-level* code. In addition, type mismatches can be used to detect errors in optimizations by the LLVM consistency checker.

Type	**Syntax**	**Description**
Integer	`i1, i2, ..., i32, ...`	Arbitrary bit width integer.
Floating Point	`float, double, fp128, ...`	Floating point numbers of different standards.
X86MMX	`x86mmx`	A value held in an MMX register on an X86 machine.
Void	`void`	Represents no value and has no size.
Label	`label`	Represents code labels.
Metadata	`metadata`	Represents embedded metadata.
Array	`[4 x i8], [3 x [4 x i32]], ...`	Simple derived type that arranges elements sequentially in memory. Requires a size and an underlying data type.
Pointer	`i32*, [4 x i32]*, ...`	The pointer type represents a reference to an object in memory.
Structure	`{ i32, i32, i32 }, { float, i32 (i32) * }, ...`	Represents a collection of data members together in memory. The elements of a structure may be any type that has a size.
Vector	`<4 x i32>, <8 x float>, ...`	A simple derived type that represents a vector of elements. Vectors are used when multiple primitive data are operated in parallel using single instruction (SIMD).
Function	`i32 (i32), i32 (i8*, ...), ...`	A function type consists of a return type and a list of formal parameter types. Can be thought of as a function signature.
Opaque	`%X = type opaque, ...`	Represent named structure types that do not have a body specified. Corresponds (for example) to the C notion of a forward declared structure.

Table 2.1: LLVM Type System

2.3.3 LLVM Instruction Set

In this Section we give a brief overview of the most common LLVM instructions. Our goal is to make the reader somewhat familiar with the syntax and the semantics of all the LLVM assembly instructions that are required to compile HiPE RTL language to LLVM assembly, explained in detail in Section 3. The complete LLVM Assembly Language Reference Manual is available on the LLVM website [16].

Terminator Instructions

Terminator instructions are always the last instructions executed in a basic block, indicating which block should be executed after the current. They produce control flow, not values (except for the `invoke` instruction).

- `ret`: Return control flow (and optionally a value) from a function back to the caller.
- `br`: Transfer control flow to a different basic block in the current function. It may either be a conditional or unconditional branch.
- `switch`: The `switch` instruction is used to transfer control flow to one of several possible basic blocks. Will usually be translated to a jump table or a series of conditional branches.
- `indirectbr`: Implementation of an indirect branch to a label, stored in a register, within the current function.
- `invoke`: Transfer control flow to a specified function, with the possibility of control flow to transfer back to either the *normal* or the *exception* label, depending on whether the callee returns with the `ret` or the `unwind` instruction respectively.
- `unwind`: Unwind the stack, returning control flow to the first caller which used an `invoke` instruction.

Instructions for Binary Operators

Binary operations are used to do most computation in LLVM. They require two operands that have the same type and produce a single value of the exact same type.

- `add, sub, mul, udiv, sdiv, urem, srem`: Do addition, subtraction, multiplication and division on two integers or vectors of integers. `Udiv`/`sdiv` produce the unsigned/signed integer quotient of the two operands. `Urem`/`srem` compute the remainder of the unsigned/signed integer division of the operands.
- `fadd, fsub, fmul, fdiv, frem`: The floating point versions of the above binary operations.

Instructions for Bitwise Binary Operators

Bitwise operations are used to perform various forms of bit-handling. All operations require two operands of the same type. Valid types are only integers and vectors of integers.

- `shl, lshr, ashr`: Perform left-shift, logical right-shift and sign-extended right-shift respectively.
- `and, or, xor`: Perform bitwise logical and, or and xor on its two operands.

Memory Access and Addressing Operations

As already stated, in LLVM, no memory locations are in SSA form but are instead mutable. However, they are accessed through pointers which are themselves in SSA form.

- `alloca`: Allocate memory on the stack frame of the currently executing function. It is automatically released when the execution of the function is over (i.e. return to the caller or perform a tail call). Creates a pointer to that memory location.
- `store`: Used for writing to memory. It needs a value and a pointer to be written to along with the appropriate types.
- `load`: Used for reading from memory. It needs a pointer and a type for the data to be loaded.
- `getelementptr`: Used for getting the address of a sub-element of an aggregate data structure, such as an array, a struct or a vector. It only performs address calculation and does not access memory.

Other Instructions

The following instructions involve various operations, such as handling of *aggregate* values, explicit type conversions, and other miscellaneous instructions.

- `extractvalue`: Extract the value of a member field from an aggregate value.
- `insertvalue`: Insert a value into a member field in an aggregate value.
- `ptrtoint .. to`: Used to explicitly convert a pointer to an integer type. Takes a pointer along with his type and an integer type to convert to.
- `inttoptr .. to`: Perform exactly the opposite operation of the above instruction. It casts an integer value to a pointer of the specified type.
- `bitcast .. to`: Convert a value to the specified type without changing any bits.
- `icmp, fcmp`: Take one conditional operator and two operands and return a boolean value (`i1`). Valid operands are integer or floating point values or vectors and pointers.

- `phi`: Implement the ϕ node in the SSA graph representing the function. Selects a value from a list according to which predecessor block the control flow came from.
- `call`: Represent a simple function call. It might have the marker `tail` indicating that the call should be tail-call optimized. The optional marker *cc n* indicates which calling convention should be used for the call. Also see Section 3.3.4.
- `select`: Used for choosing a value, out of two having the same type, based on a boolean condition.
- `landingpad`: Used by LLVM's exception handling system for specifying that a basic block is where an exception lands and corresponds to the code found in a *catch* case. It takes as argument a *personality function* that defines the behaviour for handling exceptions (by defining the *common exception frame* for the current compilation unit). For detailed information refer to the *LLVM Exception Handling* page [11].
- `blockaddress`: Used for constant computing the address of a specified basic block in a specified function. Returns an `i8*`. The value has defined behavior only when used as an operand to the `indirectbr` instruction, or for comparison against null.

Intrinsic Functions

Intrinsic functions are functions with well known names and semantics that extend the LLVM language without changing all of the transformations in LLVM. They all start with the `llvm.` prefix and are used as external functions inside an LLVM module. Only a few of them, that are used in the RTL-to-LLVM mapping, are described below.

- `llvm.gcroot`: Declare the existence of a GC root to the code generator and allows some meta-data to be associated with it. The first argument specifies the address of a stack object that contains the root pointer while the second pointer contains the meta-data to be associated with that specific root.
- `llvm.{s,u}{add,sub,mul}.with.overflow.*`: Perform signed/unsigned addition, subtraction, multiplication of two integer arguments and return a struct of two elements: the result of the computation (having the same type with the operands) and a boolean value indicating whether there has been an overflow.

2.3.4 Other projects using LLVM

Some of the projects that also use LLVM at some point of the compilation or analysis process are:

- *Clang*: A very efficient C, C++ and Objective-C native code compiler [6]. Clang's Static Analyser [7] uses LLVM to provide very useful error and warning messages.
- *VMKit*: A framework for building virtual machines that uses LLVM for compiling and optimising high-level languages to machine code, MMTk to manage memory [25]. VMKit [37] has been successfully used to build two Managed Runtime Environments (MREs), a Java Virtual Machine and a Common Language Runtime [12].

- *Rubinius*: A virtual machine for Ruby [32]. It leverages LLVM to compile Ruby code to machine code using LLVM's JIT.
- *Unladen Swallow*: A branch of Python that uses LLVM's optimization passes and JIT compiler for efficient execution of Python code [31].
- *MacRuby*: An implementation of Ruby on top of core Mac OS X technologies, such as the Objective-C common runtime and garbage collector, and the CoreFoundation framework. It uses LLVM for optimization passes, JIT and AOT compilation of Ruby expressions. It also uses zero-cost DWARF exceptions to implement Ruby exception handling [24].
- *Pure compiler*: Pure is an algebraic/functional programming language based on term rewriting. Programs are collections of equations which are used to evaluate expressions in a symbolic fashion. The interpreter uses LLVM as a back end to JIT-compile Pure programs to fast native code [30].
- *LDC*: A compiler for the D programming language [8] that is based on the latest DMD front end and uses LLVM as its back end for high quality code generation [19].
- *llvm-lua*: A JIT and static compiler for the Lua programming language that uses LLVM as the compiler back end [22].
- *GHC*: David Terei wrote a new code generator for the Glasgow Haskell Compiler (GHC) which targets the LLVM compiler infrastructure as part of his thesis [36]. The new LLVM back end appears to be very competitive with the GHC native code generator, a bit slower than the C back end in general, but, should produce big speedups for particular Haskell programs [9]. This work has been included in GHC since the 7.0 release [14].

Chapter 3

The LLVM back end

In this chapter we will present the design and implementation of a new back end for HiPE compiler which produces LLVM assembly and utilises the LLVM compiler infrastructure to generate executable code along with the necessary information for loading it.

3.1 Pipeline Design

The overall goal in the design of the new back end is to fit as easy as possible with the existing pipeline of HiPE. Our new LLVM back end is placed after the RTL representation, which is actually where all the other HiPE back ends are placed. Our decision is based on the fact that RTL aims to represent Erlang in an as low-level form as possible while still being abstracted from the underlying hardware; all high-level characteristics of Erlang, such as exception handling and garbage collection, have been explicitly expanded to code. Similarly, LLVM assembly uses a low-level instruction set and memory model that are only slightly richer than standard assembly languages [18]. Thus, the translation from RTL to LLVM assembly was a neat choice.

The new HiPE pipeline with the LLVM back end can be seen in Figure 3.1.

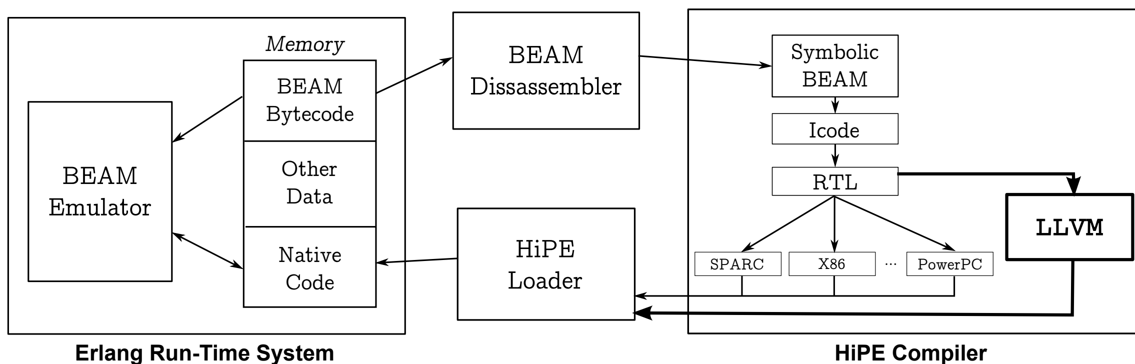

Figure 3.1: The new LLVM back end inside the Erlang/OTP system

The existing pipeline produces symbolic native assembly for the target architecture and after optimizing the code, uses HiPE's custom assembler to produce binary code that will be loaded to the Erlang runtime system by the loader. Through these phases, along with

the binary code the compiler produces all the information needed to make the code a loadable object in the Erlang runtime system:

- the compiled functions and closures,
- Erlang terms, such as constants and atoms,
- function calls, in the form of `Module:Function/Arity` (*MFA*), and BIF calls together with their stack descriptors, and
- jump tables for switch statements.

The new LLVM back end takes the code in the RTL representation and aims to produce native assembly and all the information that the HiPE loader expects. In this way we would not need to modify the HiPE loader, which seemed to be a rather painful task.

Our aim was to rejoin with the existing pipeline as soon as possible. We would actually like to join the pipeline right after the generation of the optimised native assembly, but this was not possible as HiPE uses a custom assembler which operates specifically on the symbolic assembly representation generated by HiPE. Because of this, we used the GCC assembler[1] and created an object file parser in Erlang to extract the binary code and the offsets of all external symbols. So actually the point where we rejoin the pipeline is the HiPE loader.

As seen in Figure 3.2, our new back end involves many phases which we will now present shortly and analyze them through the rest of this chapter. These phases are:

- LLVM back end (`hipe_rtl2llvm` module): In this phase RTL code is translated to LLVM assembly. This is examined in Section 3.3.
- LLVM assembler (`llvm-as`): In this phase the human-readable LLVM assembly language is translated to LLVM bitcode. This is examined in Section 3.4.1.
- LLVM optimizer (`opt`): This is an optional phase where LLVM bitcode is highly optimised. This is examined in Section 3.4.2.
- LLVM back end compiler (`llc`): The LLVM compiler translates LLVM bitcode to native assembly. This is discussed in Section 3.4.3.
- LLVM-GCC assembler (`llvm-gcc`): In this phase, an object file is created from the native assembly. In Section 3.4.4.
- Object File Parser (`elf64_format` module): Finally, we extract the binary code and all the other necessary information from the object file. In Section 3.4.5.

Before analyzing each of this phases, we must present how LLVM assembly is generated inside the Erlang/OTP system.

[1] Actually any other assembler could be used for the generation of the object file from the native assembly file.

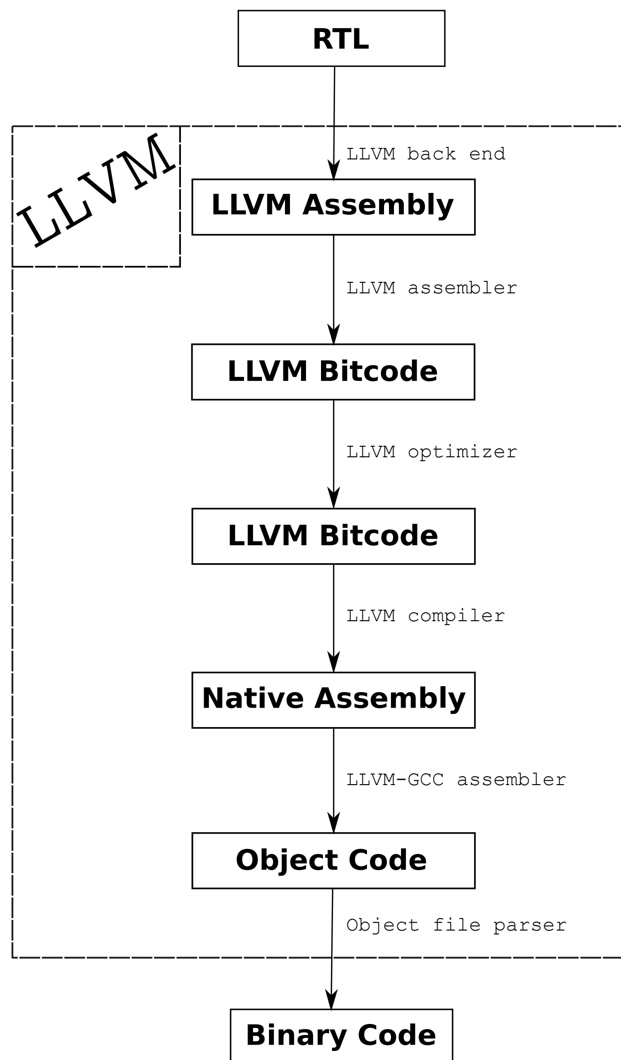

Figure 3.2: The LLVM component

3.2 LLVM Representation

The implementation of our LLVM back end required a way to generate LLVM assembly. The LLVM FAQ[2] suggests three possible approaches:

- Call into the LLVM Libraries using your language's FFI (Foreign Function Interface).
- Emit LLVM assembly from your compiler's native language.
- Emit LLVM bitcode from your compiler's native language.

We selected the second approach as this seemed to be the easiest and most flexible one, allowing us to direct our work to efficiency and correctness of translation to LLVM assembly. The first approach sounded appealing, but while there were no bindings for Erlang to the LLVM API, this option was rejected as we believed that the workload would be overwhelming and should be an independent project. The third approach offers slightly better compilation time but involves a lot of complex and error prone work in order to produce the required binary output and thus it was also rejected.

So we created a library that provides a symbolic representation of LLVM assembly instructions in Erlang, together with functions which pretty-print them. Our library does not, currently, fully cover the LLVM assembly language as only the parts that were necessary for the back end were implemented but the extension to cover everything is pretty straightforward.

3.3 Generation of LLVM assembly

In this section we will present how RTL code, which is the last target-independent representation of code in HiPE, is translated to equivalent LLVM code. Code generation process is done per function, as functions are the compilation units for the HiPE compiler.

RTL is an intermediate representation which has many similarities with LLVM assembly as they both provide low-level operations. However, LLVM assembly is a typed language while RTL is untyped. Translating an untyped language to a typed one is not a problem, as we can bypass the type system by mapping everything to type `word` or pointer to `word`.

We should point out that this part aimed to be totally target-independent, but actually some points depend on the underlying architecture. These points are the size of the `word` type and the use of precoloured registers that is analyzed in Section 3.3.1.

3.3.1 Handling RTL Virtual Registers

RTL provides unlimited virtual registers that are actually separated to three kinds:

- variables containing tagged data that are traced by the GC
- registers that are ignored by the GC

[2] `http://llvm.org/docs/FAQ.html`

- floating point registers

LLVM assembly provides unlimited virtual registers in a Static Single Assignment (SSA) representation. However, memory locations are not required to be in SSA form. This property allows the mapping of no-SSA code to LLVM assembly simply by mapping each mutable object to a stack variable. Stack variables in LLVM are declared using an `alloca` instruction, and reading from or writing to them is done explicitly by using `load` or `store` instructions.

LLVM provides a special pass, called `mem2reg`, which turns explicit stack allocation into the use of virtual registers in a way that is consistent with SSA form by using phi nodes. So, by optimizing stack utilization, `mem2reg` actually implements SSA conversion.

In the HiPE pipeline, RTL exists in both SSA and no-SSA form. The first, and most obvious choice, was to use the RTL in SSA form. However, the LLVM garbage collection infrastructure forces us to retain some virtual registers on the stack, as explained in Section 3.3.6. In order to have a simple and uniform translation, we decided to do the translation on the second one, and rely on the LLVM for the SSA conversion. So, for each RTL virtual register, an LLVM stack variable is created. Using an RTL virtual register as a source operand involves loading the stack variable to a temporary variable and using it instead, while writing to an RTL virtual register involves writing to a temporary variable and then updating the corresponding stack variable.

Precoloured Registers

Although RTL registers are virtual and independent of the target architecture, there is a small subset of them that is not. These registers have special meaning for the Erlang Run-Time System (ERTS). They are the Native Stack Pointer (`NSP`), the Process Control Block Pointer (`P`) and the Heap Pointer (`HP`). We will call these registers *precoloured* (or *pinned*). The back end is responsible for moving and retaining these pseudo-registers to specific hardware registers of the target architecture (in order to be ABI-compliant). For example, in the AMD64 back end, the above registers are pinned to `%rsp`, `%rbp` and `%r15`, respectively.

However, LLVM assembly is designed to be target-independent and offers no way to interact with the architecture. Therefore, in order to achieve placing virtual registers to specific physical ones we used a custom calling convention. This calling convention pins the first `N` arguments to registers, where `N` is the number of precoloured registers. Similarly, it pins the first `N` return values to the same physical registers. Our back end translates each function call with, say, `M` parameters to a new one which take `N+M` parameters, and each function that returns `K` values to a function that returns `N+K` values[3]. The extra arguments and return values are used for updating the pinned registers.

With this transformation, it is guaranteed that precoloured registers will have the correct value on function entry and return. At the middle of the function these registers are handled like any other virtual register. The register allocator may spill them to the stack if there is high register pressure. However, this is not a problem as precoloured registers should, in fact, be pinned to hardware registers *only* on entry and exit of a function, as

[3]For more information about this technique, please read: `http://nondot.org/sabre/LLVMNotes/GlobalRegisterVariables.txt`

these are the points of interaction with the runtime system. Actually, this approach might offer better performance since there are more registers available for the register allocator, and thus more efficient code may be produced.

3.3.2 Handling Immediate Values

In RTL we find the following immediate values:

- simple integers
- constant labels
- atoms
- closure addresses
- MFA and BIF addresses
- code labels

Handling of simple integers is straightforward, however the rest of the immediate values needs extra care as their actual value is dependent on the loader and the runtime system. The back end is unaware of what the actual value of them is and can only treat them as external symbols that will be later patched with the correct value. We will call these values *relocations* and we will present how each of them is translated into LLVM code. The general approach is to declare all relocations as external symbols. However, as we must be consistent with LLVM type system, each case must be treated differently.

MFA and BIF addresses are the easiest to translate as they just need to be declared as external functions of the appropriate type. For example the Erlang function `erlang:length/1` will be declared in the AMD64 back end as:

```
declare hipe_cc  {i64, i64, i64, i64} @erlang.length.1(i64, i64, i64, i64)
```

Notice that: the `hipe_cc` keyword defines that the function follows the HiPE calling convention and, thus, the type of the function is `{i64, i64, i64, i64}` → `{i64, i64, i64, i64}` instead of `i64` → `i64` (because of the use of three precoloured registers). The `declare` symbol is used to define an external function in a module.

Regarding constant labels, atoms and closure addresses we treat them in our back end as global external constant pointers of type `i64*`. For example atoms are declared in the following way:

```
@AtomName external constant i64
```

This instruction declares a global variable, which is an external constant of integer type with 64 bits size, and returns a pointer to this constant. After this, the pointer is converted to integer with the statement:

```
%AtomName_LocalVar = ptrtoint i64* @AtomName to i64
```

In this way the atom can be used in any LLVM expression as an `i64` variable. Constants and closures are treated in the exact same way.

Finally, code labels are mapped directly to LLVM code labels as they have the exact same semantics.

3.3.3 Handling RTL Instructions

Instruction	Description
`alu`	arithmetic/logic operation
`alub`	arithmetic/logic operation and branch after relational operation on the result
`branch`	conditional branch
`call`	call a function
`enter`	tail-call a function
`fconv`	convert a value to float register
`fload`	load float value from memory
`fmove`	move between floating point registers
`fp`	floating point arithmetic operation
`fp_unop`	floating point unary operation
`fstore`	store floating point register to memory
`goto`	unconditional branch
`label`	gives name in point in the code
`load`	load a value from memory
`load_address`	load the address of a constant or closure
`load_atom`	load the address of an atom
`move`	move value between registers
`return`	return list of variables
`store`	store a value to memory
`switch`	jump to a label according to a value of variable

Table 3.1: The RTL instruction subset that is translated in the LLVM back end

RTL and LLVM are both designed to be minimal languages to abstract the underlying architecture. For this reason many of their aspects are similar and they both provide a set of low-level operations. The set of RTL instructions that are *used* in our back end can be seen in Table 3.1.

```
1 length([]) -> 0;
2 length([X|Xs]) -> 1 + length(Xs).
```

Listing 3.1: Erlang implementation of function `length`

In order to get more familiar with RTL we will exhibit a represenative example of RTL code. In Listing 3.1 we can see a simple function which finds the length of a list, while in Listing 3.2 we can see the corresponding RTL code. We can see that the code is in the form of blocks with labels, each of which consists of simple RTL instructions. Variables starting

with `v` represent tagged values, while variables starting with `r` are arbitrary machine values.

```
{demo,length,1}(v19) ->
L13:
    r20 <- v19 'and' 2 if eq then L2 (0.50) else L3
L2:
    v21 <- [v19+7]         % [v19+7] is the address of the tail of the argument
    v22 <- demo:length(v21)
    r23 <- 31 'and' v22    % "31" is the tagged value "1"
    r24 <- r23 'and' 15
    if (r24 eq 15) then L8 (0.99) else L7
L8:
    r27 <- v22 sub 15
    v28 <- 31 add r27 if not_overflow then L15 (0.99) else L7
L15:
    v26 <- v28
    goto L6
L6:
    return(v26)
L7:
    v25 <- '+'(31, v22)
    v26 <- v25
    goto L6
L3:
    if (v19 eq -5) then L10 (0.50) else L11    % "-5" is the tagged nil value
L10:
    v29 <- 15
    return(v29)
L11:
    v30 <- atom_no('function_clause')
     <- erlang:error(v30)
    return(15)
```

Listing 3.2: RTL example: Length of a list. The "strange" numbers in the example are tags and tagged values. Addition is performed either by the 'add' instruction or by the '+' BIF, depending on the type of the value.

Each RTL instruction is mapped to one or more LLVM instructions in an almost straightforward way. The mapping will not be presented here in detail and can be found in the code[4].

3.3.4 Calling Convention

A *calling convention* (CC) [5] is a scheme for how functions receive parameters from their caller and how they return a result (e.g. in which registers they are placed, in what order, etc.). Moreover, it specifies how the task of setting up and cleaning after a function call is divided between the caller and the callee.

HiPE uses a custom calling convention which is different from the C calling convention. The two calling conventions differ in: the registers that are unallocatable and reserved for

[4] `lib/hipe/llvm/hipe_rtl2llvm.erl`

special use, the registers used for arguments and return values, the definitions of callee-/caller-save registers, and which (either the callee or the caller) pops the arguments from the stack.

At the same time, LLVM provides the infrastructure for creating a new calling convention. We defined a new CC that is compliant with HiPE's AMD64 ABI. However, the LLVM calling convention mechanism is not complete. Call-clobbered registers are not determined by the defined calling convention but are hard-coded in the back end and correspond to call-clobbered registers of the common C calling convention. Therefore, we were forced to patch LLVM and change the call-clobbered registers to go with HiPE calling convention.

3.3.5 Calls with Stack Arguments

Depending on the architecture, some arguments are passed to registers and some are passed on the stack. LLVM reserves argument space for call sites in the fixed part of the function's frame. This eliminates, in general, the need for `add/sub SP` brackets around call sites (use simple `mov` instead of `push/pop`). However, this is not the case when the calling convention defines that the *callee* should pop the arguments (like in HiPE) because the `sub` instruction cannot be avoided.

A problem, arises when this technique is combined with garbage collection and creation of stack descriptors. As already mentioned, each stack descriptor must contain the information about the size of the *fixed* part of the function's frame. For ERTS, callee-arguments that are passed on the stack are considered to belong to the *variable* part of the frame; arguments are just `push`-ed. However, with reserved call frame, LLVM stores arguments to the fixed part. As a result of this inconsistency in the handling of stack arguments between HiPE and LLVM, the runtime has a problem to traverse the stack since the information in the stack descriptors is incorrect. With this information from LLVM, the runtime will try to walk the stack by first bypassing the arguments and then the fixed part of the caller's frame, without knowing that the arguments are actually counted in the caller's fixed frame size.

LLVM does not provide a simple way for disabling the reserved call frame feature. Instead of hacking the code generator, we preferred the approach of correcting the stack descriptors, by subtracting the space for passing stack arguments from the fixed part of the caller's frame. Each stack descriptor is associated with the *return address* of a call. So, in order to fix the stack descriptors, we must know the addresses of the calls that have arguments on the stack (the return address is always the call address plus the size of a CPU word). This information is exported to the object file and can be extracted rather easily for named function calls (i.e. MFAs and BIFs) but *not* for no-name calls (closures). For this reason we created a new BIF, the `hipe_bifs:llvm_expose_closure/0` and inserted a call to it before every closure call. A call to this BIF, which is exported in the object file, notify us that the next call in the code is a closure call. In that way, we expose the closure call address and we can associate each call with the corresponding stack descriptor.

3.3.6 Garbage Collection

As already mentioned in Section 2.2.2, HiPE utilises a precise generational garbage collector. Precise garbage collection needs support from the compiler in order to know the set of

live roots in the function's frame at each safe point. To support garbage collection, LLVM provides a special intrinsic function, the `llvm.gcroot`, with which you can mark all the roots. After the compilation phase, the LLVM GC plugin is responsible for emitting the information about the roots in the object file.

The `llvm.gcroot` intrinsic is used to inform LLVM that a stack variable references an object on the heap and is to be tracked. This intrinsic takes as argument a value referring to an `alloca` instruction or a `bitcast` of an `alloca`, and some metadata related to that specific root.

So, for LLVM the root property is not a characteristic of a value but of a stack slot. Instead of marking values as garbage collection roots, LLVM marks stack slots whose contents are GC roots. As stack slots are live through the lifetime of a function by default, it is the responsibility of the front end to mark them when variables that inhabit them are no longer live. We do this by saving to the slot a value that is not traceable by the garbage collector. In our case we store the tagged representation of the empty list (`nil`). It's value is -5.

This approach taken by LLVM is inadequate for garbage collection support and creates inefficient code. Root property should actually be a property of a value and liveness analysis of the roots should be a responsibility of the LLVM back end. The back end should spill and restore garbage collection roots around safe points with the ability of reusing stack slots when roots are not live at the same time.

To understand how marking of garbage collection roots works we will present a short example. In Listing 3.3 we can see the LLVM assembly that is generated in order to mark virtual register `%X` as root.

```
1  Entry:
2     ;; In the entry block for the function, allocate the stack space for
3     ;; virtual register X.
4     %X = alloca i64*
5
6     ;; Tell LLVM that the stack space is a stack root.
7     %tmp = bitcast i64** %X to i8**
8     call void @llvm.gcroot(i8** %X, i8* null)
9     ;; Store a nil value into it, to indicate that the value it not live
10    ;; yet. "-5" is the tagged representation of nil.
11    store %i64 -5, %64** %X
12    ...
13    ;; "CodeBlock" is the block corresponding to the start of the scope
14    ;; of the virtual register X.
15 CodeBlock:
16    store i64 %some_value, i64** %X
17    ...
18    ;; As the pointer goes out of scope, store a nil value into it, to
19    ;; indicate that the value is no longer live.
20    store %i64 -5, %64** %X
```

Listing 3.3: LLVM assembly for handling a GC root

With this approach we are forced to create a separate stack slot for *each* value, which *may* be a live root. As it is understood this has a very bad impact on performance. Our future

goal is to minimise the stack, for example by storing more roots in the same stack slot, when they are not live at the same time.

3.3.7 Exception Handling

Exception handling in HiPE is implemented by adding an extra label to call instructions, if they are in the scope of some exception handler. A local exception handler is represented by the basic block that implements it. So exceptions in HiPE's CFG are just edges between a basic block that ends with a call and a basic block that implements the handler. An example of a CFG that holds a call that is protected with an exception handler can be seen in Figure 3.3. The address of the call together with the address of the exception handler is information that are passed to the runtime system with stack descriptors. In this way there is no runtime cost for setting an exception handler, since when an exception is thrown the stack map is used while traversing the stack and if a return address has an exception handler the control is transfered to the handler.

LLVM implements the `invoke` instruction, which causes control to transfer to a specific function with the possibility of control flow transfer to either a "normal" label or an "exception" label. This instruction operates as a standard call, with the only difference that it creates a connection between the call and the two labels, which is used at runtime. Each basic block that is an unwind target of an invoke instruction must start with a `landingpad` instruction. Because in RTL a basic block that is an unwind target may also belong to a separate execution path, we create a new basic block for each unwind target. This new block contains only the `landingpad` instruction and then the control flow is passed to the "unwind block".

In the code below, we can see a call to a function `foo`. The exception handler is denoted as the unwind label `%L10`.

```
%t0 = invoke hipe_cc {i64, i64, i64, i64} @foo(i64 %nsp, i64 %hp, i64 %p, i64 %arg1)
            to label %L5 unwind label %L10
...
L5:
%nsp1 = extractvalue {i64, i64, i64, i64} %t47, 0
%hp1  = extractvalue {i64, i64, i64, i64} %t47, 1
%p1   = extractvalue {i64, i64, i64, i64} %t47, 2
%t1   = extractvalue {i64, i64, i64, i64} %t47, 3
...
L10:                 ; Fail block
landingpad { i8*, i32 } personality i32 (i32, i64, i8*, i8*)*
       @__gcc_personality_v0 cleanup
;; cleanup code
```

The return address of the call and the address of the fail block are emitted to the object file and, after extracting them, they are used to create the stack descriptors.

This approach seems to work well. However, things go wrong with the usage of the precoloured registers and the calling convention. As already mentioned, the precoloured virtual registers must be updated after each call. There are two major problems that need to be solved.

At first, an `invoke` instruction must be a terminating instruction of a block. So, the update of the precoloured registers must be moved to another block. We cannot move them to

the continuation block ("normal" return) because there might be a separate execution path reaching this block. Thus, we created a new block where we update the precoloured registers.

Secondly, according to the semantics of the `invoke` instruction, the result of an invoke does not dominate its use in the path that corresponds to the unwind label. This means that we cannot access the result (containing the updated values of the precoloured registers) after an exception is thrown. In order to solve this, we introduced a new BIF in the HiPE compiler, the `hipe_bifs:llvm_fix_pinned_regs/0`. The trick with this BIF is that it takes no arguments, not even the precoloured registers. The BIF actually does nothing (returns the atom `ok`) and, thus, its call does not modify any important register. On return, we extract the precoloured registers from the return value, that have the correct values because of the register pinning and the calling convention, and store them back to the corresponding stack slots.

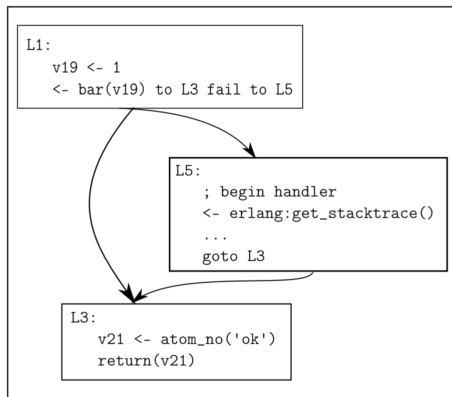

Figure 3.3: RTL CFG of a function calling `bar/1`, protected with an exception handler. Block with label `L5` represents the exception handler.

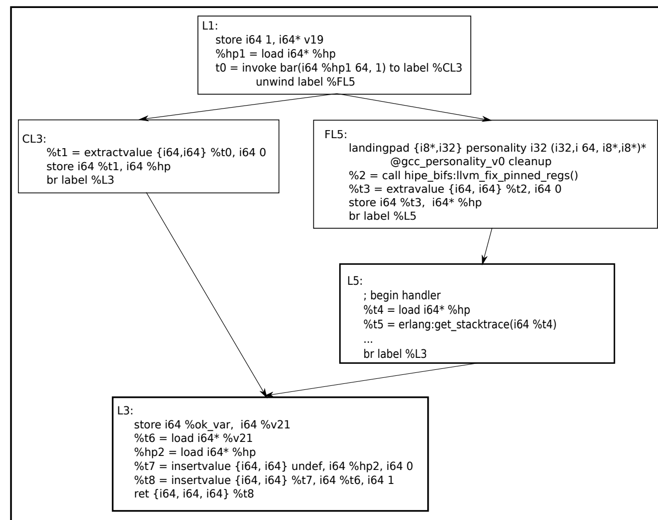

Figure 3.4: LLVM assembly of a function calling `bar/1`, protected with an exception handler. `CL3` and `FL5` are the two new basic blocks.

In Figure 3.4 we can see how the CFG of Figure 3.3 is translated to LLVM code. You can notice that two new blocks are added (`CL3` and `FL5`) which hold the `landingpad` instruction and the pinning of the precoloured registers.

3.3.8 Frame Management

As already mentioned in Section 2.2.1, HiPE generates code for managing the stack explicitly. Actually the code checks that enough space is available for maximum frame that the function may need. If there is not enough space, a call to BIF `inc_stack` is inserted. The check for the maximum possible frame for the function requires the fixed part of the frame to be known. This cannot happen before the generation of assembly, and this piece of code is not part of the RTL code of a function. In fact, it is inserted after the register allocation and the minimization of the frame. In LLVM there is a special pass called *prologue/epilogue insertion* which is responsible for finalizing the function's frame

layout, saving callee-saved registers, and emitting proper prologue and epilogue code for the function.

We modify this pass, and specifically the function `TargetFrameLowering::emitPrologue`, to insert the necessary code to the entry block of each function. The pseudo-code below defines the code that is inserted:

```
Entry:
  push %rbp
CheckStack:
  temp0 = sp - MaxStack
  if( temp0 < SP_LIMIT(P) ) goto IncStack else goto NewEntry
IncStack:
  call inc_stack
  goto CheckStack
NewEntry:
  ...
```

3.4 Rest Phases

3.4.1 LLVM Assembler

The `llvm-as` tool reads a file containing human-readable assembly (`.ll`), translates it to a binary format, the *LLVM bitcode*, and writes it to a new file (`.bc`). This format is better as the LLVM can use the more-efficient bitcode reader when interfacing to the middle end.

3.4.2 LLVM Optimizer

In this phase, a series of optimizations is applied to LLVM bitcode by invoking the LLVM `opt` tool. Optimizations are implemented as *Passes* that traverse some portion of a program to either collect information or transform the program. Every pass is implemented as a plugin library that can be dynamically loaded in the optimizer. `Opt` is used with the standard optimization groups of `-O1`, `-O2`, `-O3`, depending on the level of optimization requested by the user when triggering the HiPE compiler, and the `-mem2reg` pass that promotes memory references to register references. The result of this phase is a new binary file (`.bc`) with the optimized version of LLVM code.

3.4.3 LLVM Compiler

In this phase, the LLVM bitcode file is compiled to native assembly code for the target machine. This is achieved by invoking the `llc` tool with the appropriate options to impose rules about the memory model and the stack alignment on the generated assembly.

3.4.4 Object File Generation

In this phase, we use the llvm-gcc tool[5] in order to invoke the GCC assembler and create an ELF64 formatted object file from the native assembly.

3.4.5 Object File Parsing

This final phase involves extracting the binary code and the relocations offsets from the object file and creating the appropriate data structure for the HiPE loader. The Erlang module elf64_format[6] is responsible for this task.

[5] http://llvm.org/cmds/llvmgcc.html
[6] lib/hipe/llvm/elf64_format.erl

Chapter 4

Evaluation

In this chapter we will evaluate the new LLVM back end in comparison to BEAM, the existing HiPE AMD64 native code generator and Erjang, a virtual machine for Erlang based on Java Virtual Machine (JVM). The evaluation of a back end, usually, focuses on the quality of code and the execution time. However, there are more factors that must be taken into account, like what the compilation times are, how maintainable the code is, how easy it is to extend the back end, etc. So, we will try to evaluate our back end in a more complete way.

4.1 Current State of LLVM back end

Firstly, we will present the state of the LLVM back end. We have focused on the AMD64 architecture and implemented a HiPE back end that targets it. In only a short period of development, the LLVM back end has gone from scratch to being able to handle any Erlang program. Today, the new back end passes HiPE's test-suite and, among others, is able to **build** the complete Standard Library[1] (`stdlib`) and HiPE itself.

4.2 Performance of LLVM back end

The performance will be studied primarily by considering the execution time and, secondly, other metrics, such as the compilation time and the size of the compiled code. The evaluation will be performed against different Erlang implementations: BEAM, HiPE AMD64 back end, and Erjang.

Erjang [10] is a virtual machine for Erlang based on the Java Virtual Machine (JVM) which *just-in-time* compiles BEAM to JVM bytecode. It loads Erlang's binary .beam file format, converts it into Java's .class file format, and loads it into the JVM. Erjang, also, has a BEAM interpreter. It uses a *shared heap* memory model, so messages are not copied between processes. The main benefit of Erjang is that it is running on a virtual machine with mature implementation technology and a lot of engineering effort put into it that does dynamic compilation and selective inlining.

[1] `http://www.erlang.org/doc/man/STDLIB_app.html`

All the benchmarks were run on a sixteen-core Intel Xeon CPU E7340 at 2.40 GHz with 16 GB primary memory, running on a Debian GNU/Linux with 2.6.32 kernel in 64-bit mode. The benchmark suite consists of the following programs:

fib A recursive Fibonacci function. Uses integer arithmetic to calculate `fib(40)` 30 times.

tak Takeuchi function, uses recursion and integer arithmetic intensely. 1,000 repetitions of computing `tak(32,24,17)`.

length A tail-recursive list length function finding the length of a list of 200,000 elements 50,000 times.

qsort Ordinary quicksort. Sorts a short list 500,000 times.

smith The Smith-Waterman DNA sequence matching algorithm. Matches a sequence against 200 others; all of length 64. This is done 30 times.

huff A Huffman encoder which decodes a 32,026 character string 60 times.

decode Part of a telecommunications protocol. 5,000,000 repetitions of decoding an incoming message. A medium-sized benchmark (~ 400 *lines*).

ring This concurrent benchmark creates a ring of 600 processes and sends 100,000 messages. The benchmark is executed 5 times.

life A concurrent benchmark executing 10,000 generations in Conway's game of life on a 30 x 30 board where each square is implemented as a process. This benchmark spends most of its time in the scheduler.

barnes Simulates gravitational force between 1,000 bodies 40 times. Executed 5 times.

yaws_html An HTML parser from Yaws (Yet another Web Server) parsing a small HTML page 1,000,000 times.

prettypr Formats a large source program for pretty-printing, repeated 42 times. Recurses very deeply. A medium-sized benchmark ($\sim 1,100$ *lines*).

nrev Naive reverse of a 1,000 element list 800 times.

stable Solves the stable marriage problem concurrently with 30 men and 30 women. Creates 60 processes which send messages in fairly random patterns.

estone Computes an Erlang system's Estone ranking by running a number of common Erlang tasks and reporting a weighted ranking of its performance on these tasks. This benchmark stresses all parts of an Erlang implementation, including its runtime system and concurrency primitives.

Moreover, we have used the Erlang/OTP's Standard Library and HiPE for measuring the compilation time and the size of the generated code.

4.2.1 Results

The run-time results are summarized in Table 4.1 in terms of speedup. The LLVM back end presents noticeable speedups compared with interpreted code across a range of Erlang programs. On the contrary, the HiPE AMD64 back end still offers the best performance. It is worth to notice that Erjang's superiority on the concurrent benchmarks is attributed to the shared heap model it uses between processes and the huge heap it allocates at start.

	Program	BEAM/LLVM	Erjang/LLVM	HiPE/LLVM
Sequential	fib	2.31	0.74	0.76
	tak	1.89	0.65	0.47
	length	3.00	3.23	1.48
	qsort	1.94	1.89	0.81
	smith	3.19	1.05	0.82
	decode	2.28	1.60	0.88
	nrev	1.02	1.44	0.77
	yaws_html	1.17	2.62	0.76
	huff	1.26	1.26	0.78
	barnes	2.55	1.12	0.72
	prettypr	1.06	5.66	0.42
Concurrent	life	0.97	0.06	1.00
	stable	0.99	0.23	0.94
	ring	0.83	0.33	0.97
	w_estone	1.66	1.54	0.78
	Average	1.74	1.56	0.83

Table 4.1: Runtime Performance of the LLVM back end (in terms of speedup)

The new back end has clearly not managed to outperform HiPE. However, the results are quite promising regarding the early stage of development of the LLVM back end against the years of development and optimization of HiPE. In Section 4.2.2, we indicate some problems which we consider responsible for the slightly disappointing performance of the LLVM back end.

In Table 4.2, we can see the comparison between compilation times of the pre-existing AMD64 and the LLVM AMD64 back end. BEAM was excluded from this comparison since both the other back ends require existence of a `.beam` file in order to generate native code. We can see that the HiPE back end is approximately 50% faster than LLVM. This fact is not surprising since the LLVM back end is still immature, and attention was put to correctness rather than efficiency. The main problem lies on the fact that the new back end uses an inefficient way for representing LLVM assembly and many intermediate files for passing data from one tool to the other. We believe that the compilation time is an area that can be improved considerably in the future.

Finally, in Table 4.3 we see the comparison of the binary code sizes. LLVM back end produces slightly larger binaries than HiPE. This is imposed by the way we *have to* handle GC roots (they have to be stack allocated and cannot be promoted to registers) and the precoloured registers (they, also, live on the stack). Moreover, with our approach, we introduce extra BIF calls to the code in order to handle closures with stack arguments (see Section 3.3.5).

4.2.2 Performance Analysis

In this section we will try to examine in depth the reasons behind the performance of the LLVM back end. Specifically, we are presenting the issues which we consider to have critical impact on the quality of the generated native code.

Program	HiPE/LLVM
array	0.33
ets	0.32
io	0.61
lists	0.66
qlc	0.86
random	0.56
re	0.83
string	0.81
timer	0.64
zip	0.77
(69 more)	...
Average for stdlib	0.52
erl_bif_types	0.71
erl_types	0.82
hipe_amd64_assemble	0.76
hipe_bb	0.49
hipe_beam_to_icode	0.70
hipe_coalescing_regalloc	0.72
hipe_graph_coloring_regalloc	0.78
hipe_ls_regalloc	0.66
hipe_icode_ssa_copy_prop	0.62
hipe_rtl_ssa	0.37
hipe_x86_assemble	0.23
(185 more)	...
Average for hipe	0.49
Total Average	0.50

Table 4.2: HiPE/LLVM ratio of Compilation Times

Program	**HiPE**	**LLVM**	**HiPE/LLVM**
array	37188	61393	0.61
beam_lib	51532	70210	0.73
dict	23628	36266	0.65
erl_compile	7840	13182	0.59
io	15100	21353	0.71
lists	123932	182353	0.68
random	4620	6901	0.67
re	40688	56232	0.72
string	14048	22437	0.62
sys	18168	27002	0.67
(69 more)	...	...	...
Average for stdlib	-	-	0.67
hipe_amd64_assemble	35680	54273	0.65
hipe_amd64_liveness	4276	7108	0.60
hipe_amd64_ra	1260	2549	0.49
hipe_beam_to_icode	91332	147010	0.62
hipe_icode2rtl	21384	38659	0.55
hipe_main	32224	51525	0.63
hipe_rtl	44188	75625	0.58
hipe_rtl_cfg	27700	43223	0.64
hipe_rtl_exceptions	1868	2658	0.70
hipe_rtl_to_amd64	23332	44261	0.53
hipe_rtl_to_arm	22400	40258	0.56
(185 more)	...	...	...
Average for hipe	-	-	0.61
Total Average	-	-	0.63

Table 4.3: Binary Code Sizes (in Bytes)

Spilling/Reloading of Precoloured Registers

While the HiPE back end permanently stores the precoloured registers to specific hardware registers and marks them as unavailable to the register allocator, the LLVM back end moves them around as function-arguments and return-values and uses the calling convention to retain them to those registers. As already mentioned in Section 3.3.1, this may produce better code as during the execution of the function these registers may be spilled and used for something else, provided that there is high register pressure.

Precoloured registers have special use in the code (they are the heap pointer, the native stack pointer, the current process control block pointer, etc.) and are not always clobbered by each function. However, the LLVM back end spills and reloads them, around each call site, as it has no way to know which call can modify them. In addition, function calls in Erlang programs are very frequent as even primary operations are implemented as BIF calls. Thus, performance is reduced not only because of spilling and restoring of those registers but also because the register allocation will be suboptimal.

Code Pattern

LLVM supports all default groups of optimizations (-O0, -O1, -O2, -O3), but their impact is not reflected on the run-time results to the expected degree. In Table 4.4 we can see a comparison of the run-time speedups of our benchmark suite when compiled with three different optimization levels. We can see that the performance is only improved slightly when compiled with the `-O2` optimization group ($\approx 10\%$); `-O3` produces no further improvement. This is quite disappointing given that the LLVM optimizer is one of LLVM's most advertised features. We believe that the main reason for this is the nature of the compiled code.

Program	**O1**	**O2**	**O3**
fib	0.95	1.09	1.10
tak	1.14	1.20	1.19
length	1.12	1.47	1.39
qsort	1.00	1.03	1.03
smith	1.03	1.21	1.21
decode	1.04	1.06	1.06
nrev	1.07	1.06	1.05
yaws_html	1.05	1.05	1.05
huff	1.02	1.06	1.04
barnes	1.04	1.07	1.06
prettypr	1.03	1.03	1.03
life	0.96	1.01	1.02
stable	1.01	1.01	1.01
ring	1.07	1.03	1.04
w_estone	1.05	1.09	1.09
Total Average	1.04	1.10	1.09

Table 4.4: Runtime speedups gained by various optimization levels in LLVM

It is common for Erlang modules to consist of small and simple functions; the native code usually follows a pattern of a few primary operations and low register usage but many func-

tion calls and high memory traffic. Additionally, because of the dynamic typing of Erlang, even primary operations may involve BIF calls, for example Erlang uses arbitrary-sized integers ("bignums") and integer arithmetic is performed with a built-in function (when needed). Finally, hot-code loading restricts the effect of intra-module optimizations; the unit of compilation in HiPE is a single function and, thus, each function is compiled independently in our back end. Therefore, no optimization can occur even between functions in the same module. Because of all these factors, most of the optimization passes have no significant effect currently. We believe that there is work to be done in the future in order to take advantage of all of these optimizations that LLVM offers.

Stack Usage and Garbage Collection

In our opinion, interaction with the garbage collector and the effect it has on the register allocator is the main bottleneck in the LLVM back end. As already mention in Section 3.3.6, each probable garbage collection root occupies a stack slot, and all operations that involve this value are mapped to memory operations, i.e. reads and writes to the corresponding stack slot. Roots should be register allocated, instead, where possible, and be spilled and restored around safe points.

Even worse, as all probable roots are indicated in the stack descriptors, the garbage collection is forced to traverse those roots. Of course we store a value that the garbage collector cannot traverse to stack slots that host dead roots. However, accessing a root slot and checking its value has a considerable cost. Especially in functional languages where garbage collection occurs very often.

New BIFs

Updating precoloured registers in exception handling blocks, and exposing closure call return addresses required the creation of some new built-in functions. Calls to these BIFs is added when needed. Even if the execution time of these functions *is* minimal, the operations of setting up the stack frame, performing the call and, then, destroying the frame are not negligible. Additionally, these calls are affecting the register allocator since all values, that are allocated in non callee-saved registers, must be spilled on the stack and be restored after the call.

Branch Mechanism

HiPE incorporates a branch prediction mechanism in order to take advantage of hardware branch prediction technology and optimize the code. As HiPE is aware of the semantics of each branch, a notation is added to each branch in the form of a probability to be taken or not. After that, the code is linearized with respect to these probabilities. In this way, branches that are actually *taken* are minimized and better spatial locality is achieved.

Currently, there is no way to pass this information to LLVM. So the information that some branches will rarely be *taken* is not taken advantage of in our back end.

4.3 Complexity of Implementation

In this section we evaluate the new LLVM back end as far as the complexity is concerned. The term “complexity” refers to the amount of work that is required to build, maintain and extend the back end, and is of primary concern for the HiPE developers.

The creation of each back end of HiPE required the development of a native code compiler from RTL and an assembler to create object code. On the contrary, the new LLVM back end requires only the translation from the RTL intermediate representation to LLVM assembly, independently of the target machine. It is clear for any compiler developer that mapping from one intermediate representation to another is far more easy than creating a back end for each machine architecture.

Back end	**Size**		
ARM	Total:	5362	
	Code:	3891	
	Comments:	883	(17.6%)
SPARC	Total:	5148	
	Code:	3622	
	Comments:	881	(19.6%)
X86/AMD64	Total:	10474	
	Code:	7463	
	Comments:	1953	(18.6%)
PPC/PPC64	Total:	6695	
	Code:	5009	
	Comments:	892	(15.1%)
LLVM	Total:	5288	
	Code:	3408	
	Comments:	1293	(27.5%)

Table 4.5: Code Sizes for various HiPE’s back ends. We counted X86 and AMD64 back ends as one because they share a lot of important code. The same goes for PPC and PPC64 back ends.

In Table 4.5, we can see the sizes, in lines of code (LOC), for the various HiPE back ends. These numbers do not involve common code that all back ends may share, e.g. code from the `icode`, `rtl`, `flow` or `misc` directories. It is clear that the LLVM back end is one of the smallest. Actually, only 4072 lines correspond to the back end (many of which are comments) while the rest belong to the `elf64_format` module, that we use for extracting information from a Linux ELF64 object file.

Also, the LLVM back end allows HiPE to be extended to run on more target architectures which is a very important fact for the Erlang community. While HiPE currently supports six back ends, LLVM supports more than fifteen. This gives the opportunity to HiPE to support those architectures only by making the relevant extensions to the runtime system. On the other hand, while the HiPE compiler offers a complete pipeline, the new back end requires the installation of LLVM. Even worse, it requires the installation of a custom version of LLVM to support HiPE features, such as calling convention and Application Binary Interface (ABI). This increases the Erlang/OTP distribution and installation size.

To conclude, maintenance and development of the LLVM back end is easier. That is a big

gain for the developers since they can maintain only one back end instead of six, while the code of it is more straightforward than that of an assembler. Furthermore, LLVM has a very active development community that is quickly progressing in all areas and that progress can benefit HiPE, without extra effort for HiPE developers.

Chapter 5

Conclusion

5.1 Concluding remarks

This thesis described the architecture, design decisions, technical issues and implementation details of a new back end for the HiPE compiler which uses the LLVM infrastructure for code generation. The goal of this thesis was to examine if the creation of an LLVM back end for HiPE is *feasible* and *efficient*.

After introducing all the necessary background information for HiPE compiler and LLVM to understand this thesis, we presented the design and implementation of the new back end, and evaluated it in regard to two broad dimensions: complexity and performance. Special attention was given to being fully compliant with the HiPE Application Binary Interface (ABI) and supporting all features of Erlang, such as hot-code loading, garbage collection and exception handling.

As far as the implementation complexity is concerned, the LLVM back end has a smaller code base and is clearly simpler and more straightforward than the other HiPE's back ends. Furthermore, it effectively outsources a sophisticated part of HiPE's compilation pipeline and frees developer resources to concentrate on issues that are more directly relevant to the Erlang community. However, currently the LLVM back end needs to also distribute a custom version of LLVM with it in order to generate code that is on par with the HiPE's code generator. We hope that this will change in the near future by by contacting the LLVM developers and bringing our patch in an upstream-acceptable state.

Benchmark results indicate that code generated from LLVM back end is significantly faster than BEAM while slightly slower than HiPE AMD64 back-end's code. We have meticulously studied the reasons that we consider responsible for this. Based on this examination, we conclude that there are good indications that the LLVM back end will be able to produce at least as efficient code as HiPE in the near future.

High compilation times is, currently, another drawback of the LLVM back end. However, this area can be improved considerably by optimizing the LLVM assembly printing and avoiding the use of intermediate files between the various phases of transformations.

All in all, we consider the whole work as *successful*. We have managed to create a new back end for HiPE that can compile *any* Erlang program and has achieved good performance in only a short period of development. We strongly believe that this work can become

the basis for further work focusing on improving the HiPE compiler considerably in the future.

5.2 Future work

The primary future goal of the LLVM back end is to be extended to provide all six back ends that HiPE currently supports. This involves patching LLVM to implement HiPE's ABI for these back ends and should be rather straightforward.

An important inefficiency that has a major impact on the performance, lies on the infrastructure that LLVM provides for implementing Garbage Collection. LLVM does not allow marking a common virtual register as GC root but forces all GC roots to be stack allocated variables. There is active discussion on the LLVMdev mailing list[1], on how this infrastructure should be changed in the future LLVM releases. Maybe some work should be done in this direction.

To continue, there are some things that could be done to improve the compilation time. Printing of the LLVM assembly is currently done line-by-line. This results in performing many system calls and, thus, delaying the compilation. Other ways of printing, such as collecting data in a write buffer and periodically flushing it to the file or using other caching techniques, should be examined as solutions. Another solution would be to use the Erlang LLVM bindings [21] in order to create an in-memory representation of the LLVM assembly.

Last but not least, it would be good for both the LLVM patch and the HiPE LLVM back end to be accepted upstream in LLVM and Erlang/OTP, respectively. This involves working closely with the developers on modifying the patches to bring them on an acceptable form for the projects. The big win of this is that it significantly reduces distribution complexity and size of the back end. Furthermore, it motivates the Erlang/OTP community to work on implementing more back ends that are currently supported in LLVM by doing the needed extensions to the Erlang Run-Time System (ERTS).

[1] http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-July/041290.html

Bibliography

[1] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[2] R. L. Bernstein. Producing good code for the case statement. *Software - Practice and experience*, 1985.

[3] P. Briggs, K. D. Cooper, and L. Torczon. Improvements to graph colouring register allocation. In *ACM Transactions on Programming Languages and Systems*, pages 428–455. ACM, 1994.

[4] A. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. 2011. http://www.aosabook.org/en/.

[5] Wikipedia: Calling Convention. http://en.wikipedia.org/wiki/Calling_convention.

[6] clang: a C language family front end for LLVM. http://clang.llvm.org/.

[7] Clang Static Analyzer. http://clang-analyzer.llvm.org/.

[8] The D Programming Language. http://www.digitalmars.com/d/2.0/index.html.

[9] Smoking fast Haskell code using GHC¢s new LLVM codegen. http://donsbot.wordpress.com/2010/02/21/smoking-fast-haskell-code-using-ghcs-new-llvm-codegen/.

[10] Erjang: A virtual machine for Erlang which runs on Java. http://github.com/trifork/erjang/wiki.

[11] LLVM Exception Handling. http://llvm.org/docs/ExceptionHandling.html.

[12] N. Geoffray. *Fostering Systems Research with Managed Runtimes*. PhD thesis, Université Pierre et Marie Curie, Paris, France, September 2009.

[13] L. George and A. Appel. Iterated register coalescing. In *ACM Transactions on Programming Languages and Systems*, pages 300–324. ACM, 1996.

[14] GHC: The LLVM backend. http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/Backends/LLVM.

[15] E. Johansson, M. Pettersson, K. Sagonas, and T. Lindgren. The development of the hipe system: design and experience report. *International Journal of Software Tools for Technology Transfer*, 4, 2002.

[16] LLVM Assembly Language Reference Manual. http://llvm.org/docs/LangRef.html.

[17] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] LDC Compiler for the D Programming Language. http://www.dsource.org/projects/ldc/.

[20] The University of Illinois/NCSA Open Source License (NCSA). http://www.opensource.org/licenses/UoI-NCSA.php.

[21] LLEVM: An erlang wrapper to the C API functions of LLVM. http://www.github.com/garazdawi/llevm.

[22] LLVM-Lua. http://code.google.com/p/llvm-lua/.

[23] D. Luna. Efficiently compiling a functional language on amd64: the hipe experience. In *In PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 176–186. ACM Press, 2005.

[24] MacRuby. http://macruby.org/.

[25] MMTk. http://jikesrvm.org/MMTk.

[26] M. Pettersson, K. Sagonas, and E. Johansson. The hipe/x86 erlang compiler: System description and performance evaluation. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming, number 2441 in LNCS*, pages 228–244. Springer, 2002.

[27] M. Polleto and V. Sarkar. Linear scan register allocation. In *ACM Transactions on Programming Languages and Systems*, pages 895–913. ACM, 1999.

[28] Projects built with LLVM. http://llvm.org/ProjectsWithLLVM/.

[29] LLVM Related Publications. http://llvm.org/pubs/.

[30] The Pure Programming Language Compiler. http://pure-lang.googlecode.com/,.

[31] Unladen Swallow. http://code.google.com/p/unladen-swallow/.

[32] Rubinius. http://github.com/evanphx/rubinius.

[33] K. Sagonas and E. Stenman. Experimental evaluation and improvements to linear scan register allocation. In *Software: Practice and Experience*, pages 1003–1034. ACM, 1996.

[34] Static single assignment. http://en.wikipedia.org/wiki/Static_single_assignment_form.

[35] E. Stenman. *Efficient implementation of Concurrent Programming Languages*. PhD thesis, Uppsala University, November 2002.

[36] D. A. Terei. Low Level Virtual Machine for Glasgow Haskell Compiler. Master's thesis, Computer Science and Engineering, The University of New South Wales, October 2009.

[37] VMKit. `http://vmkit.llvm.org/`.