TIREX : une représentation textuelle intermédiaire pour un environnement d'exécution virtuel, échanger des informations du compilateur et d'analyse du programme

Artur Pietrek

▶ **To cite this version:**

Artur Pietrek. TIREX : une représentation textuelle intermédiaire pour un environnement d'exécution virtuel, échanger des informations du compilateur et d'analyse du programme. Autre [cs.OH]. Université de Grenoble, 2012. Français. NNT : 2012GRENM046 . tel-00780232

HAL Id: tel-00780232
https://theses.hal.science/tel-00780232v1

Submitted on 23 Jan 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 août 2006

Présentée par

Artur Pietrek

Thèse dirigée par **Jean-Claude Fernandez**
et codirigée par **Benoît Dupont de Dinechin**

préparée au sein **VERIMAG**
et de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

TIREX: a textual target-level intermediate representation

for virtual execution environment, compiler information exchange and program analysis

Thèse soutenue publiquement le **2 octobre 2012**,
devant le jury composé de :

Mr. Philippe Clauss
Professeur, Université de Strasbourg, Rapporteur
Mr. Albert Cohen
Directeur de Recherche, INRIA, Rapporteur
Mr. Benoît Dupont de Dinechin
Directeur du Développement Logiciel, Kalray, Co-Directeur de thèse
Mr. Jean-Claude Fernandez
Professeur, Université Joseph Fourier, Directeur de thèse
Mr. Jean-Francois Méhaut
Professeur, Université Joseph Fourier, Président
Mr. Fabrice Rastello
Chargé de recherche, INRIA, Examinateur

Acknowledgements

Foremost, I would like to express my gratitude to my both advisors: Jean-Claude Fernandez from Verimag, and Benoît Dupont de Dinechin from Kalray; first for the opportunity to work on the subject, and for their patience and knowledge, as well as motivation and encouragement in the tough moments.

I would also like to thank Philippe Clauss, Albert Cohen, Jean-François Méhaut and Fabrice Rastello, the members of my thesis committee, for their interest in my work and insightful comments. Special thanks go to Albert Cohen for his attempt to change my focus on engineering problems into a deeper, research view; and to Fabrice Rastello for interesting discussions on loops, data dependencies and traces.

Also, I thank to all my colleagues from Kalray and my fellow labmates from Verimag, for support during that three and a half years, and helpful comments. Especially to Florent Bouchez and Marc Poulhiès for their help.

I also have to mention my friends who helped me in many different things and spent their time with me on many activities: Vasso, Paris, Jean, Eduardo, Kasia, Piciu, Asia, Gosia, Karo, Justyna, Maciek, Radek, Michał, and others. Thank you! Especially I would like to thank to my parents for encouragement and to Oyuna for supporting me.

Abstract

Some environments require several compilers, for instance one for the operating system, supporting the full C/C++ norm, and one for the applications, potentially supporting less but able to derive more performance. Maintaining different compilers for a target requires considerable effort, thus it is easier to implement and maintain target-dependent optimizations in a single, external tool. This requires a way of connecting these compilers with the target-dependent optimizer, preferably passing along some internal compiler data structures that would be time-consuming, difficult or even impossible to reconstruct from assembly language for instance.

In this thesis we introduce Tirex, a Textual Intermediate Representation for EXchanging target-level information between compilers, optimizers an different tools in the compilation toolchain. Our intermediate representation contains an instruction stream of the target processor, but still keeps the explicit program structure and supports the SSA form(*Static Single Assignment*). It is easily extensible and highly flexible, which allows any data to be passed for the purpose of the optimizer.

We build Tirex by extending the existing Minimalist Intermediate Representation (MinIR), itself expressed as a YAML textual encoding of compiler structures. Our extensions in Tirex include: lowering the representation to a target level, conserving the program data stream, adding loop scoped information and data dependencies. Tirex is currently produced by the Open64 and the LLVM compilers, with a GCC producer under work. It is consumed by the Linear Assembly Optimizer (LAO), a specialized, target-specific, code optimizer.

We show that Tirex is versatile and can be used in a variety of different applications, such as a virtual execution environment (VEE), and provides strong basis for a program analysis framework. As part of the VEE, we present an interpreter for a Static Single Assignment (SSA) form and a *just-in-time* (JIT) compiler. We show how interpreting a target-level representation eliminates most of the complexities of mixed-mode execution. We also explore the issues related to efficiently interpreting a SSA form program representation.

Résumé
Certains environnements ont besoin de plusieurs compilateurs, par exemple un pour le système d'exploitation, supportant la norme C/C++ complète, et l'autre pour les applications, qui supporte éventuellement un sous-ensemble de la norme, mais capable de fournir plus de performance. Le maintien de plusieurs compilateurs pour une plateforme cible représente un effort considérable. Il est donc plus facile d'implémenter et de maintenir un seul outil responsable des optimisations particulières au processeur ciblé. Il nous faut alors un moyen de relier ces compilateurs à l'optimiseur, de préférence, en gardant au passage certaines structures de données internes aux compilateurs qui, soit prendraient du temps, soit seraient impossible à reconstruire à partir du code assembleur par exemple.

Dans cette thèse, nous introduisons Tirex, une représentation textuelle intermédiaire pour échanger des informations de bas niveau, déjà dépendantes de la cible, entre les compilateurs, les optimiseurs et les autres outils de la chaîne de compilation. Notre représentation contient un flot d'instructions du processeur cible, mais garde également la structure explicite du programme et supporte la forme SSA (*Static Single Assignment*). Elle est facilement extensible et très flexible, ce qui permet de transmettre toute donnée jugée importante à l'optimiseur.

Nous construisons Tirex par extension de MinIR, une représentation intermédiaire elle-même basée sur un encodage YAML des structures du compilateur. Nos extensions de Tirex comprennent : l'abaissement de la représentation au niveau du processeur cible, la conservation du flot de données du programme, ainsi que l'ajout d'informations sur les structures de boucles et les dépendances de données.

Nous montrons que Tirex est polyvalent et peut être utilisé dans une variété d'applications différentes, comme par exemple un environnement d'exécution virtuel (VEE), et fournit une base forte pour un environnement d'analyse du programme. Dans le cadre d'un VEE, nous présentons un interprèteur de la forme SSA et un compilateur *just-in-time* (JIT). Nous montrons comment l'interprétation d'une représentation au niveau du processeur cible élimine la plupart des problèmes liés à l'exécution en mode mixte. Nous explorons également les questions liées à l'interprétation efficace d'une représentation de programme sous la forme SSA.

Contents

Contents 7

1 Introduction **3**
- 1.1 Program compilation . . . 3
  - 1.1.1 Just-In-Time . . . 5
  - 1.1.2 Dynamic optimization . . . 8
- 1.2 A third approach . . . 9
- 1.3 Contributions . . . 10
- 1.4 Outline of this thesis . . . 11

2 Background **13**
- 2.1 Program and control-flow graph . . . 13
  - 2.1.1 Program structure . . . 14
  - 2.1.2 Control-flow graph and its dominator tree . . . 15
- 2.2 Intermediate representation . . . 16
- 2.3 Static Single Assignment form . . . 16
- 2.4 On data, data dependence and loop nesting forest . . . 18
  - 2.4.1 Data dependence definition and types . . . 18
    - 2.4.1.1 Data dependence graph . . . 19
  - 2.4.2 On loops and loop nesting forests . . . 19
- 2.5 Summary . . . 21

3 Prior Work **23**
- 3.1 Interpretation of the SSA form . . . 23
- 3.2 Compilation Just-In-Time . . . 24
  - 3.2.1 The beginning . . . 24
  - 3.2.2 Java . . . 25
  - 3.2.3 Common Language Infrastructure . . . 26
  - 3.2.4 Staged dynamic compilation . . . 27
- 3.3 Static Binary optimizers . . . 27
- 3.4 Dynamic Optimizers and Binary Translators . . . 28
- 3.5 Binary instrumentation . . . 29

3.5.1 Static binary instrumentation 29
3.5.2 Dynamic binary instrumentation 29
3.5.3 Hybrid approach 30
3.6 Code cache management 30
3.7 Summary 31

4 C to CIL compilation 33
4.1 Common Intermediate Language 34
4.2 The Community 36
4.2.1 LCC.NET 36
4.2.2 The DotGNU Portable.NET and ILDJIT Projects 37
4.2.3 The GCC4NET 37
4.2.4 The Mono Project 38
4.2.5 The LLVM-MSIL code generator 38
4.3 C to CIL compilation: study of issues 38
4.3.1 Example 39
4.4 The GCC-CIL Compiler 42
4.4.1 LLVM versus GCC4NET 42
4.4.2 The Compiler 42
4.4.3 Correctness 44
4.4.4 Improvements 45
4.5 Results 46
4.6 Summary and Conclusions 48

5 The Framework 49
5.1 Machine Description System 50
5.1.1 SSA Form on Target-Level Code 51
5.1.1.1 MDS Support for SSA Form 51
5.1.1.2 Operand Constraints in SSA Form 53
5.1.1.3 Predicated Instructions in SSA Form 53
5.2 Low Level Virtual Machine 54
5.2.1 LLVM in our toolchain 54
5.2.2 The Tirex Code Generator 55
5.2.3 Current state and limitations 56
5.3 Linear Assembly Optimizer 56
5.3.1 Program processing 57
5.4 Summary 58

6 Tirex 59
6.1 The MinIR project 61
6.2 Consistent architectural description 62
6.3 Extensions to MinIR 64

6.3.1 Code Stream Representation 67
6.3.2 Sections 68
6.3.3 Data Stream Representation 70
6.4 Loop Scoped Information 71
6.5 The use cases 74
6.6 Summary 75

7 The Tirex Runtime 77
7.1 Overview 79
7.2 Interpreter 81
7.2.1 Interpreting Instructions 82
7.2.2 Calling Native Code 82
7.2.3 Interpreting the SSA Form 84
7.3 The JIT Compiler 86
7.3.1 Code generation 87
7.3.2 Code cache 88
7.4 Summary 90

8 Conclusion 91
8.1 Future work 91
8.1.1 Intermediate representation 91
8.1.2 The runtime 92
8.2 Perspectives 93
8.2.1 Execution traces and program analysis 93
8.2.2 WCET 95
8.2.3 Native software simulation 96
8.3 Summary 97

List of Figures 101

Listings 103

Bibliography 105

Chapter 1

Introduction

1.1 Program compilation

Software development evolved drastically since first computers. Starting from programs written directly in the machine binary code, through assembly languages that made the process easier, up to many different kinds of high-level languages with different levels of abstraction. Depending on the needs, programmer can choose the language that makes it easier and less time-consuming to express the algorithms, provides portability among several different architectures, and allows to reuse some parts of the code, as well as provides easier ways of debugging.

Programs written in high-level languages depend on an other special program, a compiler, to transform them into binary code that can be executed on a given machine. The most important task of the compiler is to preserve the semantics of a source program in the resulting code. However, the higher abstraction of the source language, the more complicated it is for the compiler to generate optimal code, that is, code that would have performance similar to code written directly in the assembly language by an experienced programmer.

Of course, modern compilers deal with optimization problems quite well, assuming they do not have constraint resources in terms of time and memory. This is usually the case when a whole program is compiled into a given processor binary code prior to execution. Figure 1.1 shows a simplified compilation model, which includes in the

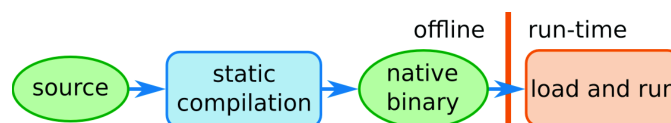


Figure 1.1: Static compilation

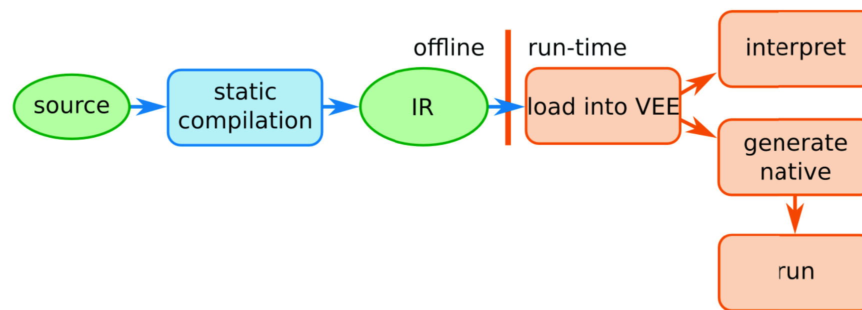


Figure 1.2: Dynamic compilation

compilation box phases such as assembling[1] and linking[2]. In such a case it does not matter how long the process will take, with, obviously, some reasonable limits, as it would be difficult to develop an application when a compiler processes one compilation unit for several hours or days. This compilation model is known as *static compilation*, where a complete, directly executable, native code is generated statically before execution of the program.

A static compiler, can produce fairly well optimized code, however, it does not have the knowledge about program execution like given parameters, size of processed data, or even the fact that not the whole program is run but rather some specific parts of it, for instance, not all conditional branches might be taken. The compiler can only speculate on the program execution behavior and misses some optimization opportunities.

Opposite to static compilation, which is terminated prior to program execution, the dynamic compilation model as shown on Figure 1.2 employs a *virtual execution environment* (VEE) or *virtual machine*[3] (VM) where the program is in some kind of intermediate form (generated statically from a high-level language) and is loaded and executed either by interpretation or after being compiled to native code on demand, prior to execution (hence the name *dynamic compilation*). Such a model adds to program execution time, thus the VEE should spend as little time as possible managing the interpretation and the compilation processes. Also, the dynamic compiler should perform only transformations that would not exceed the time gained by applying these transformations in the first place.

The dynamic compilation techniques have been known for several decades. Although at the beginning perceived too expensive in terms of time and resources to be used as an alternative to static compilation, they evolved in the last two decades be-

[1]The process of binary encoding the assembler language, resulting with so called *object files*.

[2]Combining several object files into single executable.

[3]A Virtual Execution Environment or Virtual Machine is a software that implements a machine capable of executing a program or a whole operating system. It hides the physical machine or/and the host operating system from the executed program.

coming more and more popular with the increasing performance and memory space of modern machines. They were also popularized due to the widespread of Java and .Net environments. Nowadays they could be found even in embedded, resource-constraint devices such as mobile phones and other portable devices.

These techniques, even though aiming lots of different goals, have one thing in common, that is the ability to inspect and alter a program during its execution and make use of obtained information. This could be useful not only for gaining additional performance, but also for detecting and preventing faulty or malicious program behaviors.

Among many different dynamic compilation systems, two main branches can be distinguished: Just-In-Time (JIT) and dynamic optimization.

1.1.1 Just-In-Time

Just-In-Time divides the compilation process into two stages. First the program written in a high-level language is translated into a target-independent intermediate representation with the instruction set and the ABI[4] of a given Virtual Machine. Then, the intermediate representation can be partially interpreted, as well as compiled to target-processor code during execution, or in other words compiled *just-in-time* for execution.

The main goal of this approach is to provide a way to abstract any program from the underlying operating system and processor. A program can be written and statically compiled only once into a redistributable package, without any knowledge or assumptions about the host machine, targeting a Virtual Machine instead.

JIT also aims to improve performance of programs versus their statically compiled versions. For instance, a static compiler has to assume that the smallest subset of instructions of a given architecture can be used, otherwise a program could be executed only on the specific versions of a processor that provide the particular functionality. A JIT compiler, in contrary, can generate code that use all the capabilities of the version of the processor that the Virtual Machine runs on.

A simplified JIT structure is depicted on 1.3. Usually the execution starts in interpretation mode. Opposite to compilation, and although less efficient than running native code, this process does not introduce overhead. During interpretation, program execution is observed and frequently executed fragments of the given program are identified. Then, instead of translating the whole program to the native instruction stream, the compilation and optimization processes are limited only to these "hot" fragments.

[4]Application Binary Interface – describes the interface between functions and a program with the operating system. It defines details such as data types with theirs sizes and alignments, the calling conventions (how the arguments are passed and results returned), stack-frame organization and register usage.

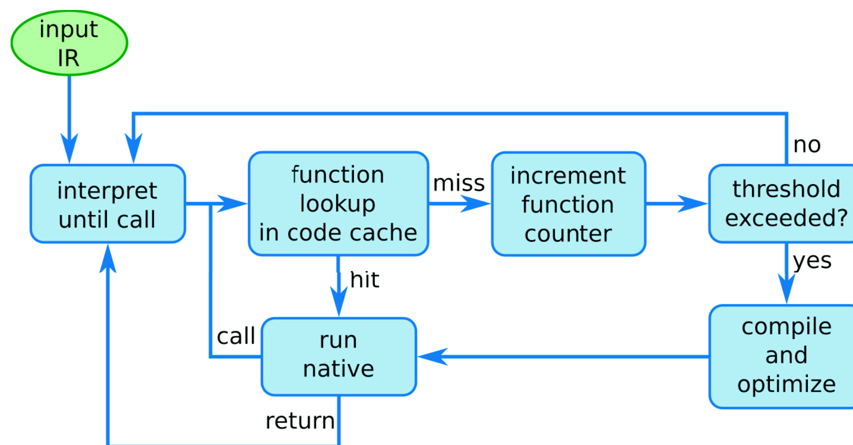


Figure 1.3: Simplified JIT structure

As an alternative, some JIT environments do not implement an interpreter at all, but rather employ two or three different compilation schemes, such as:

- fast non-optimizing compiler that produces instrumented code,
- non-optimizing compiler,
- optimizing compiler.

JIT can also benefit from the knowledge about the program execution gathered during interpretation, e.g., specializing the code to favor the paths of a program that are taken more often, or by observing that some variables in fact have constant or predictable values it can benefit from value-based optimizations. All these improvements are limited just to most frequently executed fragments, while the rest of the program, depending on the JIT constructions, is interpreted or compiled on demand with a fast, non-optimizing compiler. The hot fragments used by JIT environment as compilation and optimization units are either *hot functions* or *hot traces*.

Hot Functions In classical JIT a unit of compilation and optimization is a function. Each time a call to a function occurs a counter associated to it is incremented. When a threshold is exceeded, a function is considered to be *hot* and its (optimized) compilation is triggered. From that moment every call to this function results in calling the native code from the code cache instead of interpreting.

Operating on whole functions makes the management of compiled code relatively easy (its recompilation with more optimizations or eviction). However, obviously some paths within a function could be never or not frequently taken, thus the native code would take more space in the code cache than it is really required.

Hot Traces Some recent JIT compilers operate not on functions, but rather on traces of instructions. These *hot traces*, or *hot paths*, are usually gathered with a resolution of basic blocks during the interpretation of a program and correspond to the path taken in the control-flow graph. The scope of such a trace is not limited to a function boundary, but can capture several function calls or even recursive calls to the same function.

The difficulty of such approach is to handle the non-frequently taken, conditional paths which could be not captured into the trace. If the result of some conditional branch is different than expected, there is a need for special stub that will exit from the trace and execute the correct code. Also, some parts of different traces can overlap when capturing a recursive call, leading to code duplication in the code cache. However, the code locality in the code cache is much better in comparison to the function compilation approach.

Mixed-mode execution in JIT Although there exists JIT environments that do not rely on interpretation but rather on a few different JIT compilers instead (e.g., fast non-optimizing and slower optimizing), the process is still employed as a cheap and easy way for gathering profile, identification of hot functions or hot traces, as well as some other information. It is much easier to add profiling mechanisms along with interpretation routines than to dynamically insert special code inside a compiled program.

This, however, results in *mixed-mode execution*, a situation where both interpreted and native code are executed together. The difficulty lies in the differences between the Application Binary Interface (ABI) of the Virtual Machine and the target platform. These differences, especially *endianess* and *calling conventions* complicates the process of calling a native function from within the interpreted code and getting back the result. The same situation occurs when the native code calls a function that is not yet compiled, which results in falling back to the interpreter. These calls have to be handled through a special interface that ensures the arguments, results and the global data are compatible on both sides. This becomes more complicated when using non basic data types but complex structures and when accessing global data.

A few words on JIT compilation Before starting optimizing the code, a JIT compiler has to perform a lot of tasks, including among others:

- interpretation,
- ABI lowering,
- instruction selection,
- data layout and stack frames formation,

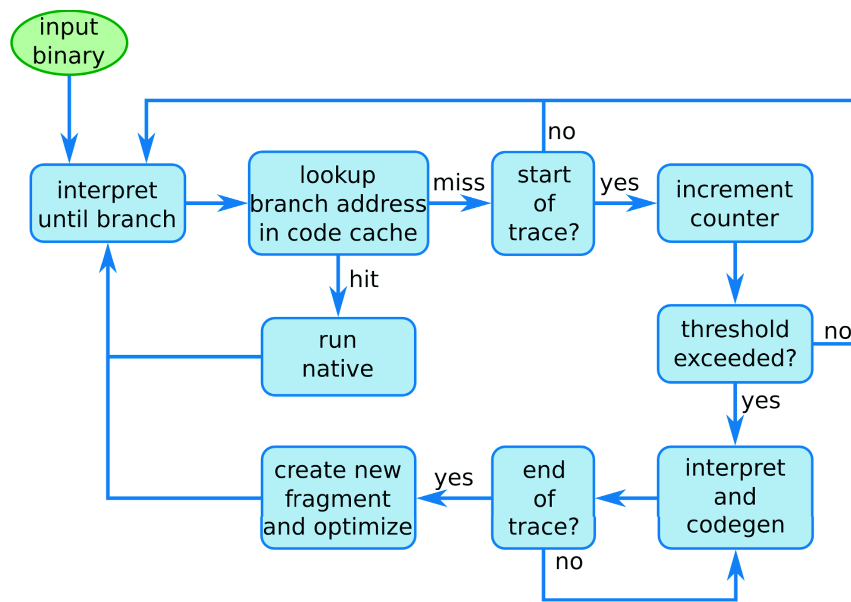


Figure 1.4: Simplified dynamic optimizer structure

- register allocation,
- instruction scheduling.

Most of these tasks are the result of target-independent intermediate representation and do not profit much from information gained at run-time. Thus even before starting optimizations, a JIT compiler introduces overhead, limiting the time that can be spent on optimizations to gain more performance.

1.1.2 Dynamic optimization

Dynamic optimization focuses mainly on improving performance of existing target-dependent binaries by exploiting run-time information that was not previously available for the static compiler prior to program execution. Figure 1.4 shows the structure of a simplified dynamic optimizer. First, the input native instruction stream is disassembled, then the program execution is observed by interpreting that disassembled instruction stream. When a branch occurs, the branch address is looked up in the trace cache. If found, the trace is executed from the cache, otherwise interpretation is continued until a trace of instructions for code generation and optimization is selected.

Hot Traces Similarly to trace JIT, only the hot traces are subject to optimization and specialization. A trace of blocks of instructions is gathered during interpretation, and is not limited to function boundaries, but can cover several function calls (including recursive calls) and can contain the same portions of code as other traces.

Limitations The target-dependent binary does not contain as much information as higher-level intermediate representation. For instance, there is no explicit program structure inside a binary. Because of that, during interpretation dynamic optimizer has to rebuild, often limited, program view and identify functions, basic blocks and loops. Identification of variables can be more complicated and sometimes even impossible, as it requires tracking memory accesses and operations on registers.

Mixed-mode execution in dynamic optimizer In contrast to JIT, dynamic optimizers interpret a native instruction stream of a given machine. Thanks to that there is only one ABI, in particular the *endianess* and *calling conventions* used for interpretation and compiled code are the same. This drastically simplifies the interoperability between interpreted and directly executed native code as there is no need to convert parameters and global data.

A few words on dynamic optimizers Before starting optimizing the code, dynamic optimizers also have to perform some time consuming tasks:

- disassembly,
- interpretation,
- instruction scheduling,
- reassembly.

The task of rebuilding the program structure, including identification of variables needed to optimize the code, is time consuming, difficult and sometimes even impossible. This limits both the time spent for optimizing the code and possible optimizations.

1.2 A third approach

The two approaches presented above (JIT and dynamic optimizers) have one goal in common: gain more performance. In many ways being similar to each other, by using interpretation, the same code cache management concepts and optimization techniques, they approach the problem from opposite directions: high-level and target-level program representation, resulting in different problems they have to struggle with.

A closer look on the two aforementioned dynamic environments leads us to a conclusion that both of them have, among others, one important need in common: information not explicitly provided with the input program representation. This need requires considerable effort to construct or retrieve that information, or even guess it from the execution context, and both of these environments spend the resources for this task at the expense of the time that could be used for their main purposes.

Interestingly, both of them rely on the static compilation phase, and the static compiler already has to construct the same information for its internal use, just to discard it before even emitting the IR or target binary.

The problem of the need for reconstruction of information (i.e., program structure, loop information, data dependencies, Static Single Assignment form and others which we discuss later) that has been already created but discarded after its use (which at some point can lead to an irretrievable loss) is not limited only to the virtual execution environments. As we will show later in this work, it occurs also in complex static compilation toolchains, where the compilation process can be divided into several phases implemented in different tools, as well as for the purpose of static and dynamic program analysis that have to be performed in the context of the underlying machine, thus on a very-low level representation.

Following these observations, we raise and provide answers for the question whether it is useful to pass previously mentioned information through an intermediate representation as an input for a virtual execution environment which combines the advantages of JIT and dynamic optimizers and tries to avoid their problems. Finally, we ask whether additional information would be advantageous in complex toolchains and as a base for low-level program analysis frameworks.

1.3 Contributions

The CIL compilation toolchain Motivated by the opportunities given by the JIT compilation, that is gaining performance while keeping the redistributable form of the program target-independent, we have studied the possibilities of compilation of the C language into the Common Intermediate Language (CIL), followed by its JIT compilation and execution using the existing open-source tools. This led us to the investigation of the CIL backend in the Low Level Virtual Machine (LLVM) compilation framework and identification of the issues related to the execution of the generated CIL code on the open-source Mono platform. We have completed the backend and corrected the issues found in the backend as well as provided a working set of tools allowing to compile programs (with the focus on C language) and execute them correctly on the Mono platform, with the intention to use that code in our own Virtual Machine.

We have noticed, however, that the JIT compiler was spending time and resources on compilation phases that do not benefit much from information gained during execution of a program. Moreover, the target-independent intermediate representation (IR) was making the interoperability between interpretation of the CIL code and execution of native code complicated, especially when operating on a complex, non-basic data structures. Also, the fact that the frontend C language allows for direct memory access and pointer arithmetic, it is difficult and in can be impossible to guarantee the target-independence of the IR.

This execution model does not leave much time for the compiler to perform opti-

mizations and limits the possibilities only to optimizations that will not consume too much time and resources, otherwise they would slow down the execution instead of improving performance.

A universal IR The need for connecting several tools in a single compilation toolchain lead us to development of Tirex, a hybrid, linear SSA intermediate representation with instructions very close to the underlying processors, an explicit program description and additional information, including loop nesting forest and memory dependencies. Its purpose is also to provide a way for writing tests for optimizer passes, as well as a strong foundation of program analysis framework and virtual execution environments.

Virtual Execution Environment The experience with JIT and the study of dynamic optimization pushed us to explore a new path of dynamic code manipulation, placed between the two aforementioned. We have decided to give up the portability in favor of gaining more time for optimizations. We have come to a clear realization that Tirex can be perfectly suitable for this task.

As interpretation is a vital part of any virtual execution environment, which provides easy way of gathering program execution traces, in this thesis we present study on SSA interpretation of Tirex. Although interpreting SSA form is challenging, we are showing that it is feasible. Also, the interoperability between interpreted and native code (mixed-mode execution) proved to be much more simplified with Tirex in comparison with classical JIT approach.

We were able to generate native code from the Tirex IR dynamically, showing that this process is simplified with regard to classical JIT, and again confirming the ease of interoperability between interpreted and native code, leaving more time for optimizations.

1.4 Outline of this thesis

In Chapter 1 we have presented a brief introduction to the world of dynamic compilation and optimization, as well as the problems related to it. Additionally, Chapter 1 summarizes the contributions of this thesis. Next, in Chapter 2 we introduce definitions and descriptions of basic concepts and terms used in this work. After that in Chapter 3 we discuss existing dynamic execution and optimization environments, as well as other concepts that made the foundations of our work. Chapter 4 shows issues related with C to CIL compilation and our contributions in this matter. After that in Chapter 5 we describe the framework used for the purpose of this thesis. In Chapter 6 we introduce and describe in details Tirex intermediate representation following by its virtual execution environment consisting of a interpreter SSA and JIT compiler, both

described in Chapter 7. Finally we conclude our work and discuss perspectives in Chapter 8.

Chapter 2

Background

In this chapter we define the terminology for the next chapters. First, we define the vocabulary related to the program itself and its structure from the compilers point of view and provide overview of intermediate representations. Then, we introduce the Static Single Assignment form. After that, we describe loop forests and their relation with data dependencies.

2.1 Program and control-flow graph

A computer program is a sequence of instructions that executed would perform the task intended by a programmer. Or more formally, a **program** is a set of instructions connected by control-flow edges. An edge from instruction s to instruction d means that instruction d can be executed after s. The **control-flow**[1] refers to the order in which the instructions are executed, and can be represented by a graph, described in Section 2.1.2.

An **instruction** is a smallest, single operation that possibly uses some variables and possibly defines other variables. It can, but does not have to, take some input values, and also can return a result of some computation. Examples of some instructions are shown on the Figure 2.1.

Instruction	Definitions	Uses
`add $r0, $r1, 123`	`$r0`	`$r1, 123`
`goto BB4`	$\emptyset$	`BB4`
`nop`	$\emptyset$	$\emptyset$

Figure 2.1: Instructions with the defined values and used parameters.

We also consider a group of instructions that will be executed in parallel, and call it

[1]Or **flow of control**.

a **bundle**. Bundles are specific for VLIW processors that have the capability of executing multiple instructions in the same clock cycle, depending on the resources available (logical units). For instance, a load from memory and some arithmetic operations on registers can be performed simultaneously. Of course, the exact number of instructions in one bundle is constrained by architecture of the processor, availability of resources and mutual dependencies between the instructions. All the instructions in a bundle are encoded in a binary form together resulting in a *very long instruction word*, hence the name *VLIW*.

2.1.1 Program structure

For the convenience of the programmer, and for the purpose of the compiler as well, a program can be divided into more units than simple instructions (or bundles of instructions). What we call a **structured model of programming** was initiated in 1960s by work of Bohm and Jacopini, and Edsger Dijkstra. As a contrast to an older, previously used model, where some condition test and goto instructions were used, this one introduced the use of procedures[2], blocks of instructions, and loops. Although this model can slightly differ within different programming languages, internal low-level representation (independent from the used programming language) from the compiler point of view is structured in a similar fashion. This structure evolved during the years to have granularity suitable for different kind of optimizations.

In this thesis we will refer to this structure hierarchically: a **program** contains **procedures** build of **basic blocks** made of **instructions** (or **bundles**). Also, some basic blocks within a procedure can form a **loop** (we talk about loops in Section 2.4.2).

A **basic block** is a maximal sequence of instructions in a given program, in which flow of control enters at the beginning and leaves at the end without branch or call in the middle. In other words, a basic block is a portion of code that would be executed sequentially from its beginning (*entry point*) to the end (*exit point*). No entering or leaving path is allowed in the middle. Usually the last instruction in a basic block is a branch, call or return instruction, however it is not mandatory – in that case the control flow falls to the following basic block in the program.

Grouping the instructions that are executed sequentially without any control change in between allows for using blocks of instructions as optimization unit, e.g., in *basic-block reordering* which is used to reduce branch cost and the number of mispredicted branches.

A **procedure** is a way of abstraction and encapsulation of a part of a program that usually performs a specific task. In many environments it serves as a unit for separate compilation. At the same time it is also a unit of some optimizations that could not

[2] Also called functions, or subroutines.

be performed more locally (i.e., cannot be done on a single basic block). In more formal way, a **procedure** is a portion of code build from one or more basic blocks, that possibly takes some *arguments* and possibly returns a *result*.

2.1.2 Control-flow graph and its dominator tree

A procedure can be represented by the **control-flow graph** (CFG), a directed graph $G = (N, E, r, t)$, with set of nodes N, set of edges E, the **entry** node r with no incoming edge, and the **exit** node t with no outgoing edge.

The CFG models the flow of control in that procedure and provides a graphical representation of all possible run-time paths. Usually the nodes represent basic blocks. In static analysis, the CFG is a basic tool used in several phases of compilation, including instruction scheduling and register allocation.

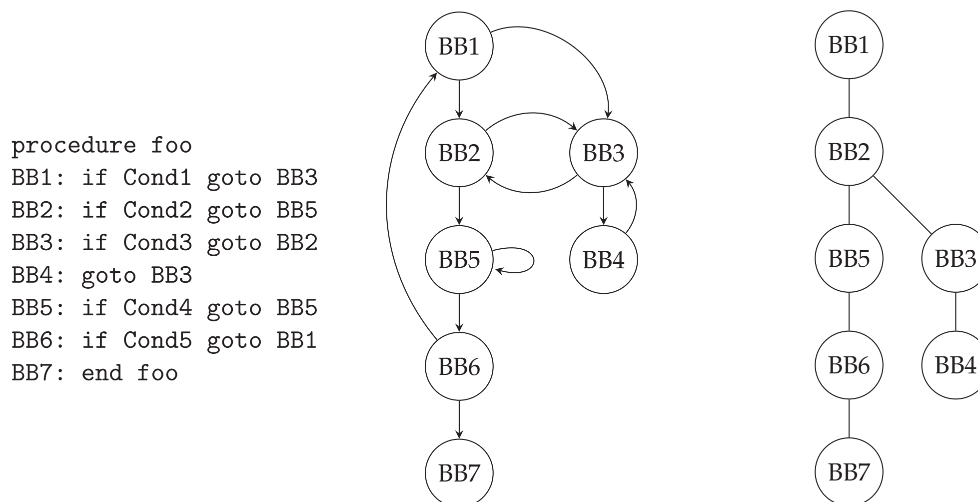


Figure 2.2: Example procedure, its control-flow graph and a dominator tree.

Figure 2.2 shows an example procedure, its CFG and a *dominator tree* (defined below) for that CFG. The basic block BB1 is the entry node, and BB7 – the exit node.

Dominance property and the dominator tree A node a in a CFG **dominates** a node b, denoted a dom b, if every path from the entry node r to b contains a. If a dom b and $a \neq b$ then a **strictly dominates** b, or a sdom b. The node a is an **immediate dominator** of the node b, a idom b, if a sdom b and there is no node c such that a sdom c and c sdom b.

A **dominator tree** rooted at r is a directed graph whose nodes are the nodes of the given CFG and where each node other than r is pointed to by its immediate dominator.

2.2 Intermediate representation

Intermediate representation (IR), is a language of an abstract machine used to represent a program independently from a programming language used to write that program and often independently from the target processor. Modern compilers usually convert the original program into such IR, and perform analysis and optimizations on it. In some compiler implementations there could be more than one IR involved, however in such cases one of the representations is the **main IR**.

From the view of structural organization IRs can fall into three categories: graphical, linear, and hybrid.

Graphical IRs represent the code as **trees** or **graphs** and consist of **nodes** and **edges**. Trees are a natural representation for the grammatical structure of the source code. Graphs, however, are more useful for representing other properties of a program, such as the control flow.

The graphical IRs are usually not the only IR in the compilation process. The data dependence graphs are generally used for a specific task, created just before the use, and destroyed afterwards. The control-flow graph (CFG) also comes with another IR for representing the operations inside basic blocks.

Linear IRs share a resemblance to assembly code in the sense that they both are a simple sequence of instructions, executed in order of their appearance. In that form, in contrast to graphical IRs, there is a need to encode transfer of control, hence generally each basic block ends with a branch, call or return instruction.

The two, most common linear IRs are **bytecode** and **three-address code**. Bytecode is also called stack-machine code or one-address code, as each instruction has at most one parameter (one address) which is placed on the evaluation stack before its execution. The result of the instruction is also placed on the stack, thus it has to be explicitly stored in memory if required. The bytecode is used in applications where size is essential, as it is very compact. Three-address code resembles more the code of modern processors, were most of the instructions get two operands and produce one result (thus making it *three*-address). Of course, some of the instructions will need fewer arguments and in some cases more than three.

Hybrid IRs try to combine properties of both aforementioned. A common practice is keeping linear code within basic blocks, and use graphs to represent the flow of control between these blocks. An example of commonly used hybrid IR is a linear IR in the Static Single Assignment form, which we describe in the following section.

2.3 Static Single Assignment form

The **Static Single Assignment** form is a naming convention for variables, which was the result of work on data-flow analysis, in particular on the identification and elimination of redundant computations published by Alpern et al. [1988]; Rosen et al. [1988].

Later, the foundations of SSA were presented in more details in a journal paper by Cytron et al. [1991].

A program is in SSA form if each variable is defined exactly once in the program text. In other words, a variable can occur on the left hand side of an assignment expression only once in a given program, hence there is one unique **static** definition. It is possible, however, to get multiple **dynamic** definitions – for example, as a result of the definition inside a loop. Usually, in SSA all definitions should occur before its use, i.e. it is considered with **dominance property**.

We have already defined the dominance property while talking about the CFG in Section 2.1.2. SSA is said to be **with dominance property** if, for every variable a, $def(a)\ dom\ use(a)$, where $def(a)$ is the instruction that defines a and $use(a)$ is a set of the instructions that use a.

Since some variables have multiple definitions, to convert a program to SSA unique names have to be given to each definition of the same variable. However, simple renaming is not enough. At points where different control-flow paths merge (e.g., after an if else condition), the correct value has to be chosen depending on the path. For that purpose so-called virtual ϕ-functions were introduced.

A **ϕ-function** is a virtual operation placed at the beginning of a basic block, which takes as many arguments as the number of this block's incoming flow edges, and returns the nth argument, when the execution path comes from the nth incoming flow edge.

```
if(b)
  a = 2;
else
  a = 3;
c = a + 1;
```

```
if(b)
  a_1 = 2;
else
  a_2 = 3;
a_3 = φ(a_1, a_2);
c = a_3 + 1;
```

Figure 2.3: Example of C code and its representation in SSA.

The ϕ-functions need a particular treatment for multiple reasons. First, they act as multiplexers that choose a value depending on which control-flow path the program comes from, hence their behavior depends on past execution.

For instance, as shown on Figure 2.3 suppose a conditional branch in the original program with assignment `a=2` in the *true* branch and `a=3` in the *false* one. Since SSA allows only one static assignment, this would be transformed into a_1=2 and a_2=3 in the branches, and the ϕ-function $a_3=\phi(a_1, a_2)$ at the join, meaning that a_3 takes the value of a_1 if the path comes from the *true* branch, and a_2 if it comes from the *false* branch.

Moreover, multiple ϕ-procedures at the beginning of a block need to be executed concurrently as their semantics is that the selection of values is done in parallel. Failure to do so may produce wrong results, the classical example (see Figure 2.4) being the

```
a = φ(b, ...);
b = φ(a, ...);
```

Figure 2.4: Swapping of two variables in SSA form using ϕ-procedures.

"swap problem" was identified by Briggs et al. [1998] where ϕ-procedures are used to swap values in variables. Executing them sequentially would result in the overwriting of variable a before its use in the second ϕ.

2.4 On data, data dependence and loop nesting forest

A complete program, besides the instruction stream, has to contain **data** used for the computations. While talking about the **data stream** in our intermediate representation, we will use the term **object**. Data objects correspond to definition of **variables** and **constants** referenced in a program. They describe how much storage has to be assigned for the variables and constants, and how should they be placed in memory in order to correctly perform operations on them. Some of the variables and all constants have to be set to some initial values before program execution. We will refer to that value as **initializer**.

In more general view data is an **input** (**operand**) of some program statements[3], and can be its **product** (result of computation). The compiler, in order to perform some optimizations has to keep track of data dependencies between the program statements. Otherwise it could for example move a definition of a variable after it use, resulting in a faulty computation. To perform so-called *aggressive optimizations*, the compiler needs memory dependencies annotated by the type of relation and discriminated by loop level.

2.4.1 Data dependence definition and types

A statement S_w **depends** on statement S_v if and only if both of these statements access the same memory location and at least one of them stores into that memory location, and there exists a path of execution from S_v to S_w.

There are three types of dependencies (see Figure 2.5 for examples):

Flow dependence Statement S_w is data flow-dependent on S_v (denoted as $S_v \delta S_w$), if a value computed in statement S_v is used in statement S_w.

Anti dependence Statement S_w is data anti-dependent on S_v if a variable is used in statement S_v before it is reassigned in statement S_w ($S_v \bar{\delta} S_w$).

[3]Or instructions in case of low-level program representation

Output dependence Statement S_w is data output-dependent on S_v when a variable is assigned in statement S_v and later reassigned in statement S_w ($S_v \delta^\circ S_w$).

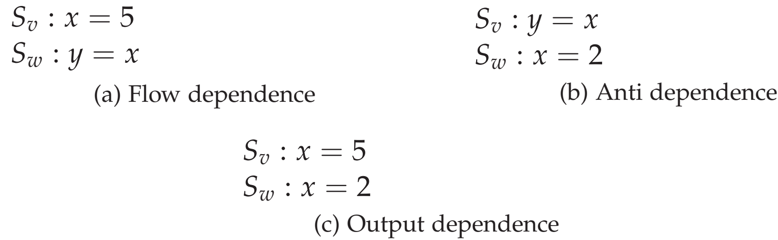


Figure 2.5: Example of data dependencies between statements.

2.4.1.1 Data dependence graph

The data dependence graph is a directed graph $G = (N, E)$, where each node $n \in N$ corresponds to one of s program statements $S_x (1 \leq x \leq s)$, and each edge $e = (S_v, S_w) \in E$, is a dependence relation between statements S_v and S_w.

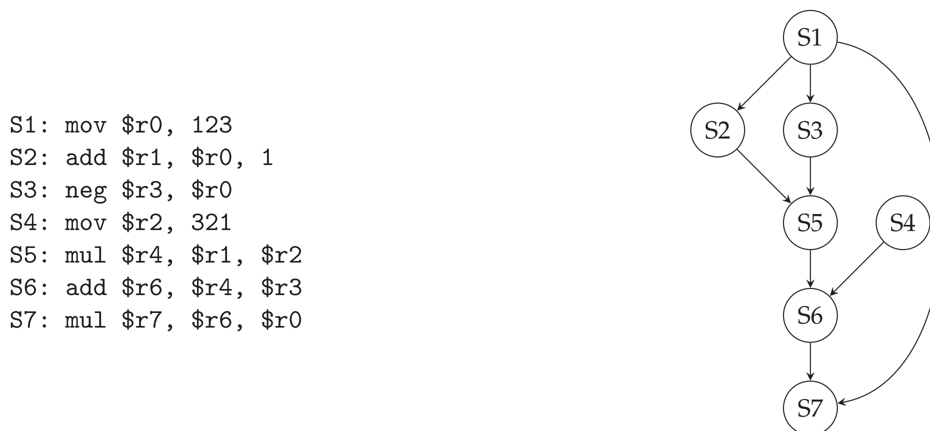


Figure 2.6: Example of data dependence graph.

Figure 2.6 shows a simple program in a three-address form and its data dependence graph. Statements $S1$ and $S4$ have no incoming dependency, while the other statements are flow-dependent on other statements in the program.

2.4.2 On loops and loop nesting forests

When discriminating dependencies by loop nesting level, all the producers and consumers of the intermediate representation have to agree on what loop they are talking about. A loop nesting forest is a data structure representing loops and relations

between them in a control-flow graph. There are several definitions (with different properties) of what a loop nesting forest is, e.g., Sreedhar et al. [1996]; Steensgaard [1993]; Havlak [1997].

Ramalingam [2002] proposed the axiomatic definition of loop nesting forests. In his definition, the header of a loop can be any node not dominated by other in the SCC (strongly connected collection) that defines the loop body. So in case of irreducible loops, there is an ambiguity.

In his PhD thesis, Boissinot [2010] defines the notion of a minimal connected loop nesting forest, which has applications to find live sets and perform liveness checks under the SSA form. A minimal loop nesting forest is a loop nesting forest such that no loop body from the forest is covered by another loop, and it is connected if the loop headers are selected from the set of entry nodes of the loop. In practice, loop forests proposed by Havlak [1997] are the best choice of minimal connected loop nesting forests.

Havlak's loops A *loop* is a sequence of statements that can be executed repeatedly. A loop consists of a *header* basic block and loop body that contains one or more basic blocks. The body of a loop can be executed a specified number of times, until some condition is satisfied, or indefinitely.

For a compiler, a loop is one of the most important scopes for some optimizations, as this is usually where the most time is spent during execution. These optimizations aim to improve cache performance and speed of loop execution, making use of parallel execution of some statements in the loop.

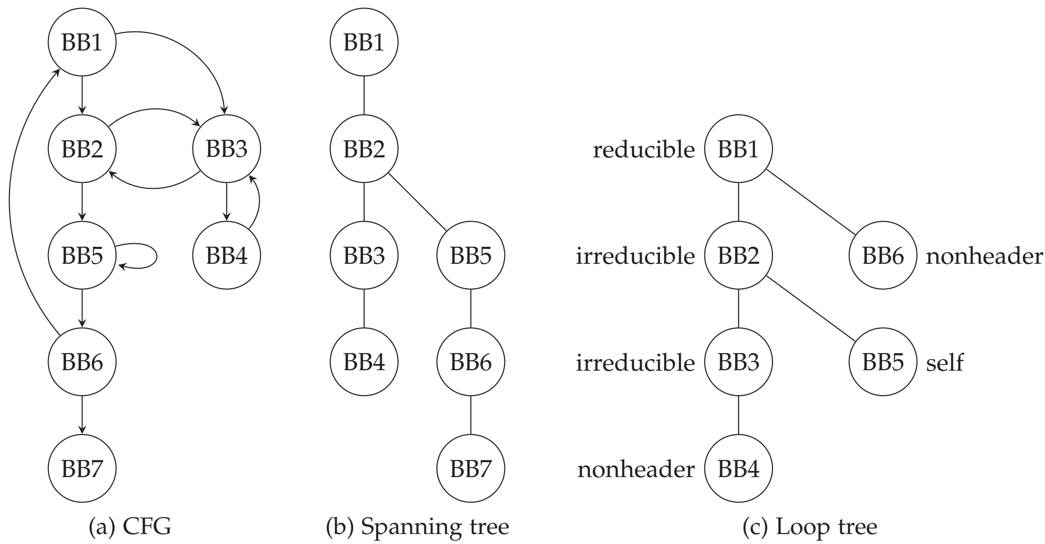
In this work we use definitions as provided by Havlak [1997], because only one basic block can be the header, hence there is no ambiguity as in Ramalingam's axiomatic definition and others. A more formal definition of loops comes from the notion of a *strongly connected component*.

In a given graph $G(N, E)$, a **strongly connected component** (SCC) is a nonempty set of nodes $S \subset N$ such that for any nodes $a, b \in S$, there exists a path from a to b and a path from b to a. A *maximal strongly connected component* is a SCC to which no more node can be added.

A **loop** in a graph $G(N, E)$ is a pair (B, H) of non-empty sets of nodes $B \subset N$ and $H \subset N$, such that $H \subseteq B$, where B is a strongly connected component, the body of the loop, and H is the set of headers of the loop. An **outermost loop** is a maximal SCC with at least one internal edge.

When traversing a graph with the classical depth-first search algorithm, a spanning tree is created as a by-product, which is then used for numbering the visited nodes. A header node of the loop is the first visited node of the SCC. A loop **nested** inside a loop L with header h is an outermost loop in the subgraph with node set $(L - h)$.

This definition allows us to represent the loop nesting with a tree, where for each node its parent is a header of the smallest loop containing that node. A loop with



(a) CFG (b) Spanning tree (c) Loop tree

Figure 2.7: An example of a loop tree for given CFG and its spanning tree.

a single entry is a **reducible loop**, while loop with multiple entries – an **irreducible loop**.

We give an example of a loop tree on Figure 2.7 with a corresponding CFG and a spanning tree. Havlak gives additional labels to the nodes in a loop tree. Nodes with descendants are headers and are marked **reducible** or **irreducible**. Nodes that do not have any descendant are marked either as *nonheader* or **self**, where **self** is a special case of a reducible loop with a single node, hence the header in itself is the loop.

2.5 Summary

In this chapter, we defined the main notions used in this thesis. We provided the notions about program structure and data stream, defined the control-flow graphs and discussed the intermediate representations. We introduced the SSA form, which is supported by our intermediate representation and used by the framework, both of which are presented in the following chapters. Finally, we provided background information on data dependencies and loop nesting forests, also supported by the proposed IR, to be used in the future by our framework.

Chapter 3

Prior Work

The main motivation of this thesis is to fill the gap between the execution environments that employ the classical *Just-In-Time* execution model with the *Dynamic optimizers*, and combine the advantages of both of them. These virtual execution environments seem to be the two major approaches in the wide family of dynamic code modification systems. The work done in this thesis rely however also on other technologies, that we describe in this chapter.

3.1 Interpretation of the SSA form

Interpretation, although providing much less performance than execution of native code, is nonetheless very important tool in a virtual execution environment, as it allows for fast start-up of execution without the overhead of a compiler. It is also a cheap and easy implementation base for gathering traces and other useful information about the program execution. The interpretation techniques themselves being not new are well known, however not many interpreters of SSA form exist.

One of the interpreter operating on the SSA form could be found as a part of the Low Level Virtual Machine (described in more details in Chapter 5). It is written in C++ and designed for executing LLVM's bitcode, a target-independent intermediate representation which is in the SSA form by design. However, no work on measuring its performance or describing its implementation could be found. The authors state themselves in the source code, that it was designed "to be very simple, portable and inefficient."

Another work on interpretation of the SSA form was done by von Ronne et al. [2004]. Von Ronne proposes an "Interpretable SSA intermediate representation", which introduces some extensions to a classical SSA flavor in order to enable direct imperative interpretation of SSA. These extensions include a dedicated virtual register containing the edge number used by ϕ-functions and a special instruction marking explicitly the end of ϕ-functions in a basic block.

Although Von Ronne mentions that recursive calls would require storing the variables, probably on the stack, the discussion ends on that point.

3.2 Compilation Just-In-Time

3.2.1 The beginning

The idea of the classical JIT compilation can be traced to Rau [1978] who was considering a *universal host machine* (UHM). Rau classified program representations into three groups (see Section 2.2) according to their level of abstraction and usability for the execution on a virtual machine. The UHM was targeting multiple source languages, called by Rau *high-level representations* (or HRLs). He postulated the introduction of an intermediate representation (DIR) that the different HLRs would be statically compiled to. This DIR was suitable for interpretation and dynamic translation to the host machine code (DER).

Rau in his work also discussed the different design criteria of DIRs, such us code compaction versus semantic level of instructions and their impact on the UHM. He proposed also the use of dynamic translation instead of interpretation, as well as the organization of *dynamic translation buffer* (what would be called now a *code cache*) and its similarities to processors' cache memory.

The work of Rau inspired Deutsch and Schiffman [1984] for improving the Smalltalk [Goldberg and Robson, 1983] system. Smalltalk is a purely object-oriented language used in an interactive, exploratory programming model. Its source code is compiled statically to a bytecode representation. This bytecode was originally executed in a virtual machine by a stack-oriented interpreter. Deutsh and Schiffman in their implementation of Smalltalk as one of the optimizations used dynamic translation of the bytecode to native code. The procedures were compiled in a lazy scheme, i.e. just before the call, and the native version was cached for later use.

Smalltalk influenced another purely object-oriented system that uses dynamic translation techniques, Self Ungar and Smith [1987]. Rather than interpreter, Self used a fast, non-optimizing compiler for lazy compilation of procedures. Hölzle and Ungar [1994] credited this idea to work of Deutsch and Schiffman in Smalltalk. Three generations of Self's implementation provided lots of important observations used later in other systems.

Hölzle proposed the use of second, optimizing compiler and its application to *hot spots*. When a method was executed often, it was recompiled with that optimizing compiler. It was also Hölzle who noticed, that it is more important for the performance to carefully choose *what* to compile rather than *when* to compile, i.e., that instead of compiling a method whose counter triggered the compilation it could be better to compile its caller and inline the frequently called method.

3.2.2 Java

One of the best known environments using dynamic translation is Java [Gosling et al., 1996] released by Sun Microsystems in 1995. It was designed as an alternative to C++, that would provide facilities for security, threading, distributed programming, as well as automatic memory management, and above all – target architecture independence that would allow programmers to write and compile their software only once but deploy on any environment where a Java Virtual Machine is available. As a result of lot of invested time and money, nowadays it could be found everywhere from desktop computers to embedded devices, even in mobile phones; in fact it is Java that brought the term *just-in-time* (or simply JIT) into use in computing.

Java, as other JIT environments, separates the compilation process into two phases. First the source code is compiled statically into a bytecode representation, next the bytecode is executed on the Java Virtual Machine (JVM). This bytecode was designed for a stack-based (virtual) machine and in the first releases was simply interpreted which resulted in very poor performance.

This problem was addressed by Cramer et al. [1997] with, once more, use of the techniques proposed by Rau and pioneered by Deutsch and Shiffman: dynamic translation of bytecode to native code. The compiler used for runtime code generation has to be much faster than static compiler. Cramer and his colleagues stated, that even though the bytecode, as being designed for stack-based interpretation, is not perfect for JITing, there is no time and resources for constructing different intermediate representation as in static compilers. They advocated the use of bytecode as the intermediate representation of the JIT compiler. Although the compiler was very simple, the results showed increase of performance, depending on benchmark, from 2 to 9 times versus the interpreted code.

Cramer also noticed, that although there obviously is lot of place for improvement in the JIT compiler, the performance of whole virtual execution environment depends not only on the native code generation and optimizations. According to him, the interpretation process took in average only 68% of execution time, while the rest was spent by the virtual machine on synchronization and garbage collection, and surprisingly only 1% on calling native code.

The following years, in particular with the work of Cierniak and Li [1997]; Adl-Tabatabai et al. [1998]; Burke et al. [1999]; Suganuma et al. [2000] and others, resulted in new ideas in Sun's JIT compiler as well as in different research JIT compilers for Java that improved the performance significantly. Important conclusion of this work is that simply generating native code instead of interpreting is not enough, bringing focus on optimization techniques suitable for JIT compilation.

Interesting implementation of Java Virtual Machine is Java HotSpot [Oracle, 2010]. It addressed also other issues related to execution of Java bytecode, i.e., synchronization and garbage collection, however we will focus only on the JIT compilation.

From the very beginning [Microsystems, 2001], instead of the lazy compilation model, i.e., compiling one method on demand, the execution started in the interpretation mode. The interpretation process was used not only to avoid the compilation overhead, but also to detect the frequently executed parts of the program (i.e., *hot spots*). Java HotSpot provides two JIT compilers: client and server. The client compiler is tuned to be used with a typical client applications that require fast response and does not perform any optimization except minimal inlining. The server compiler is tuned to optimize long-running server application. It perform all classical optimizations, such as dead code elimination, loop invariant hoisting, common subexpression elimination and constant propagation.

The stack-based IR considered previously by Cramer as suitable for the JIT process have been replaced for compiler's internal purpose by a register-based IRs, and the server compiler started to exploit the properties of the SSA form. Later versions of the client compiler use two intermediate representations, one of which is SSA-based, and the server compiler has been improved versus the older one to gain more performance and has been equipped with Java-specific optimizations, such us null-check and range-check elimination and exception throwing optimizations.

3.2.3 Common Language Infrastructure

In early 2000 Microsoft introduced his .Net Framework, which was later partially published under the name of Common Language Infrastructure (CLI) as ECMA International [2006] standard. Its goal was to unify other Microsoft technologies under one API, thus allowing the programmers to develop and debug GUI, console, web and other applications in the same fashion. CLI in many ways is similar to Java, however it was designed later, thus its creators could make use of lessons learned by the creators of Java. In contrast to Java, which was designed to support only the Java language, CLI's philosophy is to provide support for multiple frontend languages and the interoperability between the code written in these languages, hence giving the possibility to choose the best suitable language to perform a given task.

Programs written in high-level frontend languages are compiled to stack-based bytecodes called Common Intermediate Language (CIL) previously known as Microsoft Intermediate Language, which is part of the Ecma standard and was described in details by Lidin [2006]. CIL is independent of the frontend language and from the underlying hardware, and although is suitable for interpretation, the .NET framework does not use an interpreter, but directly calls a JIT compiler [Gough, 2001] whenever a function is called for the very first time.

The .NET framework relies on two JIT compilers [Microsoft, 2006]: Standard-JIT and Econo-JIT. The latter is a fast, non-optimizing compiler, while the former produces better code.

A shared source, research implementation of CLI, named Shared Source CLI (SSCL

or Rotor) was also released by Microsoft. It was equipped with a one, very simple JIT compiler that did not perform any optimization, nor register allocation. It was designed to generate native code very fast in one pass and simulate the stack in a dedicated register. Vaswani and Srikant [2003] proposed a profile guided JIT compiler for Rotor with two level optimizations. Again, as it happened in Java, this JIT compiler internally was operating on a register-based intermediate representation.

The Mono project[1] provides an open-source implementation of the Common Language Infrastructure. Mono has its own JIT compiler. For its internal purpose starting from version 2.0 it converts the CIL bytecodes into a register-based linear intermediate representation [Mono, 2012]. It performs all the usual optimizations, such as copy and constant propagation, branch optimizations, dead code elimination. It performs the SSA optimizations as well, however the internal linear IR is not in SSA, thus requires its construction and destruction.

3.2.4 Staged dynamic compilation

Other methods are used in staged dynamic compilation [Auslander et al., 1996; Consel and Noël, 1996; Grant et al., 1999]. The compilation process of a program is also divided into two parts: static and dynamic. The static compiler chooses only portions of code, typically previously annotated by user and generates templates, which are used at runtime by a dynamic compiler to specialize these parts of code without introduction of any intermediate representations.

Leone and Dybvig [1997] proposed even to statically compile the regions of code into three forms for different kind of optimizations: high-level IR for heavyweight optimizations, mid- or low-level IR for simple optimizations, and native code for regions that cannot benefit from dynamic optimizations.

3.3 Static Binary optimizers

Srivastava and Wall [1992] developed OM for link-time optimizations. OM takes object files and converts them into RTL language. This representation is optimized profiting from inter-modular optimizations not possible during compilation of each unit, and converted back to object form.

Spike [Cohn et al., 1997] is a profile-driven Alpha binaries optimizer for Windows NT. It automatically instruments binaries and collect profile data during execution. The collected data is used for static optimizations of the entire images of programs, including dynamic libraries.

[1] http://www.mono-project.com/

3.4 Dynamic Optimizers and Binary Translators

We classify the dynamic optimizers and binary translators together, as they are similar in the way they operate on existing, target-dependent binaries rather than dedicated intermediate representations. Usually they observe the execution by interpreting the instruction stream, gather hot traces of instructions, and generate (optimized) native code.

One of such systems is Dynamo introduced by Bala et al. [2000]. It is designed for to transparently improve performance of statically generated binaries for PA-RISC processor. While interpreting the program, it gathers the hot-traces and generates optimized code for them. In case of performing worse than original binary it has a bailout mechanism to fallback to the original program. Dynamo converts the traces into a low-level IR, very close to underlying architecture and performs simple branch optimizations, redundancy removal, constant and copy propagation, strength reduction, loop invariant code motion and loop unrolling. Bala reported performance gain up to 22% in some cases, and an averages speedup of 9%.

Bruening et al. [2003] proposed a direct descendant of Dynamo called DynamoRIO, which provides a dynamic binary inspection and modification framework with an API exposed to higher-level applications, allowing to construct tools that can perform several different tasks. Its goal is not only to dynamically optimize existing programs, but rather to provide a framework for implementing such dynamic optimizations and analyses. It also uses interpretation for gathering traces, however to avoid overhead it caches translations of frequently executed code for direct execution.

Chen et al. [2000] worked on Mojo, a dynamic optimization system targeting Windows running on IA-32 architecture. Its goal is to run not only research benchmarks but also real-world, large applications with multithread support. As it is designed for a CISC processor, the authors, instead of using the interpreter as in Dynamo or DynamoRIO, proposed direct execution of native basic-blocks identified through disassembling and placed in a *basic-block cache*. Control instructions in such basic-blocks are patched to go back to a dispatcher. After identifying a hot-path, corresponding basic-blocks are optimized and placed in the *path cache*. The optimizations focus on code layout and branches, as well as loop unrolling. Mojo was also equipped with bailout mechanism in case of performing worse than native version of a given program.

DELI is a binary translation framework developed by Desoli et al. [2002], meant for near-native performance emulation with an API exposed to clients. It allows to build code manipulation, observation and emulation tools by providing some basic functionalities, such as code caches and fragment linking mechanisms. However, interpreter or JIT compiler of one ISA to native has to be provided by the client tools.

Strata [Scott and Davidson, 2001] is yet another framework for developing dynamic translation clients. Its goal is to provide a retargetable platform that has some common

services, such as memory management, code cache and dynamic linking. The authors show a use case for inspecting applications during execution and protect the system from malicious behaviors, as well as an dynamic instruction scheduler.

3.5 Binary instrumentation

3.5.1 Static binary instrumentation

A framework for writing program analysis tools for the Alpha platform, through a high-level API was implemented in ATOM [Srivastava and Eustace, 1994]. It is based on the OM project and contains instrumenting services and requires the user to specify only the tool detail. It inserts function calls to analysis framework at the user-specified points. User's tool compiled with the ATOM services results in the profiling tool which is further combined with the application resulting in a self-profiling and analyzing executable. Atom, however does not allow for modifying existing instructions, but only allows to call instrumentation functions.

EEL is a very similar toolkit to ATOM. Although released for SPARC, was designed with a platform-independent interface for "exutable editing" that would require only minimal knowledge about the target platform.

Another instrumentation toolkit, PEBIL Laurenzano et al. [2010], was influenced by a dynamic toolkit DynInst (described later), but instruments program statically by replacing some instructions with calls to instrumentation routines at specified points. To acquire enough space for this reason in case of variable-length instructions, it performs function relocation. Also, it supports insertion of *instrumentation snippets* – hand written assembly code, to avoid instrumentation functions.

3.5.2 Dynamic binary instrumentation

DynInst [Buck and Hollingsworth, 2000] is a C++ class library that provides an API for dynamic program instrumentation and modification. The API also allows for changing function calls and removing them from the application. It uses machine independent representation to describe the instrumentations that allows to use the same instrumentation code across different platforms. The inserted code is translated to machine language and emitted into array in the application address space. Than, in order to call this code, one or more instructions in the point of instrumentation is replaced with a branch to a *trampoline*. Next another, *mini-trampoline* is called that saves the machine state (registers and condition codes), executes a *snippet* of the instrumentation code, restores the machine state and branches back to the main trampoline. Here another *snippets* can be called. When the process is finished, the replaced original instructions are executed, and the trampoline branches back to the point of insertion.

Another library that provides API for building dynamic instrumentation tools is implemented in Pin [Luk et al., 2005]. Similar to DynInst it provides the possibility to

write target-independent instrumentations. However, instead of trampolines, it uses dynamic compilation techniques to instrument executables while they are running. It has capabilities to optimize the instrumentations and uses code caching and trace linking.

Different framework for dynamic binary instrumentation is provided by the Valgrind [Nethercote and Seward, 2007] project. It comes as a core tool with plug-ins. The core provides services to make common tool tasks easier. Valgrind relies on dynamic compilation techniques – after loading the plug-in, the client's code is disassembled into an intermediate representation, instrumented and compiled just-in-time. As opposed to other systems, Valgrind does not execute the original code and it does not target the performance, but the program analyses itself.

3.5.3 Hybrid approach

The previously mentioned static approaches instrument existing binaries. VMAD [Jimborean et al., 2012] in the other hand, rely on a specially prepared x86_64 binary files. The profiling is initiated by the programmer who insert in the source code dedicated *pragmas* to mark the regions of interest. Next, a custom compilation pass generates multiple versions of that regions: instrumented and original. It differs from the static framework also by the fact, that this *multiversion* binary is executed on a virtual machine. Although the x86_64 binary does not require dynamic compilation techniques, VMAD tries to optimize the code according to the gathered profile. The multiversion approach minimizes the overhead of the instrumentations, as the instrumented version can be executed only when needed.

3.6 Code cache management

Important of any dynamic code manipulation environment is the code cache and its efficient management, especially in environments with constraint resources.

Several management schemes were evaluated by Hazelwood and Smith [2002]. Hazelwood concluded, that treating the code cache as a circular buffer reduces the miss rate by half compared to what is achieved by flushing the cache when full. Furthermore, such a model does not require complicated bookkeeping and hence does not add significantly to the overhead and memory requirements of the whole virtual execution environment.

Hazelwood and Smith [2003] brings her work even further with a proposition of a generational cache management. She proposes to group together code traces with similar expected lifetime and introduces two separate circular buffers to keep short-lived and long-lived traces and a third one called nursery used as a temporary step before either promotion of shored-lived trace to long-lived one or its eviction. As

Hazelwood noted, this scheme results in an average miss rate reduction of 18% over a unified code cache.

3.7 Summary

In this chapter, we have described several different virtual execution environments, from classical JIT to dynamic optimizers and binary translators, showing many ideas and evolution of dynamic code manipulation systems. We have also discussed existing interpreters of the SSA form, as well as the techniques related to code cache management – equally important element of a virtual execution system as the dynamic compilers itself. The VEE proposed in this thesis is based on some of the ideas presented here and tries to avoid problems that exist in these solutions, to provide another, not yet explored path of virtual execution.

Chapter 4

C to CIL compilation

The C language is still very popular, especially among the embedded software developers, even though several other high-level languages exist. This comes partially from the fact that most of such developers are used to write software which is very close to the underlying machine but also due to the performance and resources necessary to execute a program written in C versus higher-level languages such as Java, C# or even C++. Furthermore, lot of legacy code that is still being used was written in C, making rewriting it from scratch in higher-level language time consuming, thus expensive.

Still, programs written in C, including the legacy code, could profit from the dynamic compilation and optimization techniques to gain more performance versus their statically compiled versions. Some open-source tools to achieve this goal exist, however none of them is complete in terms of fully functional compilation and execution chain.

In comparison to JIT compilation of C# or Java, requirements of a virtual machine for executing C-written programs are much lower as C is not a type-safe language; it also does not require garbage collection and complicated exceptions handling. Hence, assuming a robust C to intermediate representation compiler exists, a lightweight virtual execution environment for dynamic code specialization and optimization could be provided.

From the two widely known JIT environments, that is Java Virtual Machine (JVM) and the Common Language Infrastructure (CLI, also known as .NET), the latter brought our attention due to its intermediate representation which has been designed to support multiple frontend languages, including non-type-safe languages, as opposite to JVM's IR designed solely for the compilation of the Java language. Moreover, some parts of the CLI, including its IR were standardized. This fact, along with some open-source tools available, including a virtual machine, makes it interesting subject of experiments.

In this chapter we present existing open-source tools, as well as our contributions

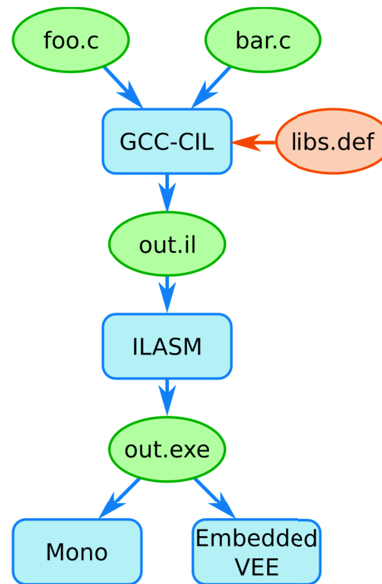


Figure 4.1: GCC-CIL compilation and execution flow

that include the study of issues related to the compilation of C to the Common Intermediate Language (CIL) and its execution on the open source virtual machine, the Mono platform. We also provide an open source compilation chain as shown on Figure 4.1, called GCC-CIL that allows to generate robust CIL from C code and execute it on the Mono platform with the support for native libraries, and a possible future execution in a custom, lightweight, embedded virtual machine. Furthermore, the introduced compilation chain in contrast to some other existing solutions does not rely on DotGNU Portable.NET and a custom linker to produce CIL bytecode from multiple compilation units, hence simplifies the compilation process.

In the whole chapter, we will refer to the CIL code as *managed code*, as opposed to the *native code*.

4.1 Common Intermediate Language

The Common Intermediate Language was introduced as an intermediate language for the Microsoft .NET framework under the name Microsoft Intermediate Language (MSIL). Later however, the foundations of the Microsoft .NET framework, have been published as the ECMA standard 334 for the C# language and the ECMA standard 335 for the Common Language Infrastructure (CLI). In particular, the latter specifies the CIL for use by a virtual execution environment.

Similarly to Java IR, the CIL is also a bytecode program representation designed to be executed on a stack-based machine. It has, however, a major advantage in the

context of C language compilation compared to the Java bytecode: it was designed to be used as a target for the compilation of multiple, different frontend languages. It can support low-level imperative languages like C, object-oriented languages like Java and C#, and even functional languages of the OCaml family like F#. Furthermore, while Java itself provides mechanisms to interoperate with native code, it is done via *Java Native Interface*, a special API designed for that function, the calls to native code cannot be expressed directly in the bytecode, making it greatly limited. CIL in contrary is capable to express these calls, hence does not suffer from the same problems.

The standardization of a processor-independent binary format suitable to represent compiled C programs, combined with the availability of dependable tools and the virtual execution environment maintained by the Mono project, offer significant opportunities to the high-performance and embedded systems community:

- Multiprocessors and grid computing systems could abstract away the processor ISA when compiling application code, provided a CIL to native code generator and a virtual execution environment exist for each processor. In case of non-managed languages like C/C++, the virtual execution environment can be reduced to a bare minimum by omitting garbage collection, exception handling, and the .NET framework libraries. C++ is non-managed from the viewpoint of a CLI virtual execution environment, as its exception handling and type introspection are not compatible with the CLI.

- Aggressive program specialization techniques, that leverage JIT compilation of high-level bytecode, could become generally applicable to C/C++ programs. Combining program specialization with dynamic instrumentation and JIT compilation is state-of-the-art in Java virtual machines like the Jikes RVM. Conversely, the application of static program specialization techniques (without JIT compilation) results in significant code expansion. This problem precludes the use of static program specialization in embedded computing, where program code size is a main constraint.

- Assuming robust C to CIL compiler exist, when defining a new processor, most of the compiler development effort reduces to retargeting a CIL to native code generator. As a side note, Microsoft discontinued the C to CIL compilation after the release of Visual Studio 2003. Meanwhile, the C++ compilation on any version of Visual Studio does not produces 'pure' CIL: object files are not CLI-compliant, and they embed native code. So compilation of C/C++ programs to pure CIL is another area where open-source delivers more value than proprietary software.

- Since the CIL bytecode is a standard program representation (as opposed to the serialization of compiler internal representations, such as the Open64 *WHIRL* or

the LLVM *bitcode*), it behaves like any other machine languages as far as GPL licensing is concerned. However, the CIL program representation is processor-independent, so it does not leak details of the target processor architecture and implementation to potential competitors.

A project evaluating the potential of CLI-related technologies for embedded computing applications was started by a team at STMicroelectronics Manno laboratory in 2005, in particular with the work of Costa and Rohou [2005]; Cornero et al. [2008]. The by-products of this project, closed in 2007, are contributions of Costa et al. [2007] and Svelto et al. [2009], known as GCC4NET in the `st/cli` GCC branch. Work of Dinechin [2008] and Boissinot et al. [2008, 2009] provided a research virtual execution environment for CLI called PVM, and a JIT compiler for the ST200 VLIW and ARM processor families. Of these, only the GCC4NET contributions are open-sourced.

Motivated by the opportunities mentioned above, in early 2009 we have resumed the work on C to CIL compilation, based this time on an LLVM-MSIL code generator that was orphaned by its creator, Roman Samoilov. LLVM was selected at the time for its post-link optimization capabilities.

4.2 The Community

The projects that provide tools useful for C to CIL open-source compilation community include:

- LCC.NET,
- DotGNU Portable.NET,
- ILDJIT,
- GCC4CIL,
- Mono,
- LLVM-MSIL.

4.2.1 LCC.NET

LCC.NET is an extension to the Princeton LCC compiler by Fraser and Hanson. Although developed for Windows .NET, is available as source code that is easily portable to GNU/Linux. More important, this extension was described by Hanson [2004] in a companion paper, which provides the blueprint for correct and complete C to CIL compilation. Indeed, Hanson is also one of the fathers of the LCC compiler at Princeton University (before moving to Microsoft Research). As acknowledged in this

LCC.NET paper, he benefited from direct access to the Microsoft .NET team members when developing LCC.NET.

LCC.NET however, comes not without limitations. The main one is that the LCC compiler is C89-compliant only. It means that `long long` types implemented as 64-bit integers are not supported. The other limitation is that LCC itself and LCC.NET are not open-source, even though source code is available. This prevents the redistribution of modifications to LCC.NET, including our port to GNU/Linux. LCC.NET also introduces useless complications, as far as Mono execution on GNU/Linux is concerned. In particular, the provision of a thunk mechanism whose only purpose is to correct the mismatch of calling conventions between the Win32 native code and the Microsoft .NET JITed code.

4.2.2 The DotGNU Portable.NET and ILDJIT Projects

The DotGNU Portable.NET[1] project sponsored by Southern Storm Software aims to provide everything to compile C and C# programs and execute them on a CLI-compliant virtual execution environment. In order to support separate C compilation and access to other assemblies, the C compiler implements extensions to the C language, and the provided `ilasm` tool is able to encode incomplete CIL text files into a binary format that generalizes the CLI assembly specification.

The DotGNU Portable.NET project seems to be stalled since early 2007, and at that time neither the C compiler nor the companion C library were stable enough. Fortunately, the just-int-time compiler called libjit [2] appears to be still actively developed, in relation with the ILDJIT[3] project. The ILDJIT project at Politecnico di Milano focuses on parallel and distributed dynamic compilation.

4.2.3 The GCC4NET

The GCC4NET[4] contributions originate from STMicroelectronics work in two related areas. The first contribution called gcccil is a frontend for GCC that parses .NET assemblies using Mono libraries, and compiles to native code. The .NET assemblies accepted as input are currently constrained to be result from C to CIL compilation. The second contribution is the GCC branch st/cli-be that compiles C to CIL text files.

In order to support separate compilation, a number of choices have been made, most notably the use of GCC attributes to tag references to native functions in the C source code, and the use of the DotGNU Portable.NET `ilasm` tool that allows the compilation of CIL text files into incomplete (non-CLI compliant) assembly binary

[1] http://www.gnu.org/software/dotgnu/
[2] http://en.wikipedia.org/wiki/LibJIT
[3] http://ildjit.sourceforge.net/
[4] http://gcc.gnu.org/projects/cli.html

files. These incomplete assembly files are then turned into CLI-compliant assembly files using the DotGNU Portable.NET `ilalink` tool.

The GCC4NET contribution eschews some of the issues of native code linking by providing a standard C library that can be compiled to managed code. Bindings of this C library to the guest operating system relies on a system call like interface that only deals with unstructured data and basic types.

4.2.4 The Mono Project

The Mono[5] project supported by Novell is the most actively developed and maintained open-source CLI-compliant virtual execution environment. Thanks to on-going collaborations between STMicroelectronics and the Mono team, Mono provides a stable execution platform and dependable tools that enable to execute CIL programs on GNU/Linux, including those that result from C compilation.

The Mono project initially had its own plans to develop a C to CIL compiler based on Open64 then GCC, which were discontinued after the release of GCC4NET [6]. Nevertheless, the Mono project still envisions a different way of addressing the native code linking issues by using their `cecil` link tool library.

4.2.5 The LLVM-MSIL code generator

Development of the MSIL code generator in LLVM (see Chapter 5) was started by Roman Samoilov with the purpose of assembling and executing the resulting managed code with Microsoft .NET.

A particular feature of the LLVM-MSIL code generator is that it is patterned after the C backend of the LLVM, unlike the 'real' code generators of LLVM. Unlike the C backend however, the LLVM-MSIL was unable to emit correct code after SSA optimizations for the classic 'swap problem' of SSA destruction described by Sreedhar et al. [1999].

4.3 C to CIL compilation: study of issues

There are several issues with the C to CIL compilation which could be divided into two groups: generic, independent from any particular code generator; and these related to implementation of code generator in a specific compilation framework. The latter we discuss in Section 4.4, while here we focus on the three generic problems, related to each other, that have, in our opinion, the most significant engineering impact:

- the support of separate compilation;

[5]http://www.mono-project.com/
[6]http://www.mono-project.com/Gcc4cil

- native code linking;
- calls to variadic native functions.

Separate compilation units In the CLI world there is no equivalent for the C-style separate compilation of source files and static linking. Rather, compilation of source code directly produces load modules (similar to System V dynamic libraries) called assemblies, which are the units of deployment and versioning. Within each assembly, external references are either to code and data in other assemblies, or to native code functions in designated dynamic libraries. In order to avoid target processor dependencies in the CIL programs, there is no access to data exported by native libraries. While this feature can be a problem for accessing global variables like `stdin` or `errno`, these are special cases for which workaround exist [7].

Native code linking When working in a traditional separate compilation flow, a C to CIL compiler needs to know whether a particular function referenced but not defined will be provided as managed code, or is a reference to external native code. In the former case, as standard C does not provide support for importing existing .NET assemblies, the function must be part of the current assembly. In the latter case, a so-called p/invoke (platform invoke) stub must be created by the compiler that specifies the dynamic library where the native function will be found.

Calls to variadic native functions If the called native code function is variadic the situation is even more complicated. All the call sites in the managed code must be patched to remove the 'vararg' tag from the call signature. Following that, one stub for each call signature must be provided. This fix is currently required by the Mono runtime on GNU/Linux.

4.3.1 Example

We illustrates these issues using a patched version of LCC.NET. In this example we compile separately two files from C to CIL. The first file defines function `func()`:

```
cat -n func.c
     1 int func(int i) {
     2   return i + 1;
     3 }
```

The second file defines function `main()`, which calls function `func()` and also the variadic function `printf()` twice, with two different signatures:

[7] In orther to use global variables from native libraries, one has to provide native wrapper functions that simply return pointers to these variables.

```
cat -n main.c
     1 int printf(const char *format, ...);
     2 int func(int i);
     3
     4 int main(void) {
     5   printf("Hello!\t");
     6   printf("func(%d)=%d\n", 1, func(1));
     7   return 0;
     8 }
```

The result of compiling `func.c` to managed code is straightforward. Function argument 0, then constant 1, are loaded on the evaluation stack. Managed code adds them, leaving the result on the evaluation stack, then returns:

```
cat -n func.il
     1  // file=func.c
     2  .method public hidebysig static int32 'func'(int32) cil managed {
     3    .maxstack 2
     4    ldarg 0
     5    ldc.i4 1
     6    add
     7  $L1:
     8    ret
     9  }
```

The result of compiling main.c to managed code is more involved. First, the C-style string must be represented by specific data-types, which are defined in lines 2 and 3. These strings are then initialized using the CLI static field mapping feature in lines 24 – 27. Then there are the two calls to the `printf()` function. These two calls should normally appear like the commented in lines 8 and 17. However, the `printf()` function will eventually be discovered to be implemented as native code. In that case, correct execution on Mono requires that native calls are non-variadic (calls to variadic functions implemented by managed code work as expected). So we eliminate the variadic tags from the call sites, and this results in lines 9 and 18.

```
cat -n main.il
     1  // file=main.c
     2  .class private value explicit ansi sealed 'int8[8]' { .pack 1 .size 8 }
     3  .class private value explicit ansi sealed 'int8[13]' { .pack 1 .size 13 }
     4  .method public hidebysig static int32 'main'() cil managed {
     5    .locals ([0] int32 '1')
     6    .maxstack 3
     7    ldsflda valuetype 'int8[8]' $4a2ce9df_65e5__2
     8    //call vararg int32 'printf'(void*)
     9    call int32 'printf'(void*)
    10    pop
    11    ldc.i4 1
    12    call int32 'func'(int32)
    13    stloc 0
    14    ldsflda valuetype 'int8[13]' $4a2ce9df_65e5__4
    15    ldc.i4 1
    16    ldloc 0
    17    //call vararg int32 'printf'(void*,...,int32,int32)
```

```
    18    call int32 'printf'(void*,int32,int32)
    19    pop
    20    ldc.i4 0
    21  $L1:
    22    ret
    23  }
    24  .field public static valuetype 'int8[13]' $4a2ce9df_65e5__4 at $4a2ce9df_65e5_7
    25  .data $4a2ce9df_65e5_7 = { bytearray ( 66 75 6e 63 28 25 64 29 3d 25 64 a 0 ) }
    26  .field public static valuetype 'int8[8]' $4a2ce9df_65e5__2 at $4a2ce9df_65e5_8
    27  .data $4a2ce9df_65e5_8 = { bytearray ( 48 65 6c 6c 6f 21 9 0 ) }
```

Those two files cannot be assembled yet. They need to get completed by a file that defines all the remaining undefined references, whether managed or native code. In addition, because function `main()` is the entry point in C, we need to startup the execution by initializing static data (not necessary in our example), and to terminate execution by a call to `exit()`. The `illink` tool of the LCC.NET distribution, again with our patches, produces the following file:

```
cat -n x.il
     1  .assembly 'x.exe'
     2  {
     3    .hash algorithm 0x00000000
     4    .ver  0:0:0:0
     5  }
     6
     7  //.method public static  hidebysig  pinvokeimpl ("/lib/libc.so.6" as "printf" ansi cdecl )
     8  //vararg int32 printf (void* )  cil managed preservesig { }
     9
    10  .method public static  hidebysig  pinvokeimpl ("/lib/libc.so.6" as "printf" ansi cdecl )
    11  int32 printf (void* )  cil managed preservesig { }
    12
    13  .method public static  hidebysig  pinvokeimpl ("/lib/libc.so.6" as "printf" ansi cdecl )
    14  int32 printf (void*, int32, int32 )  cil managed preservesig { }
    15
    16  .method public static  hidebysig  pinvokeimpl ("/lib/libc.so.6" as "exit" ansi cdecl )
    17  default void exit (int32 )  cil managed preservesig { }
    18
    19  .method public static  hidebysig
    20  default void $Main (string[] argv)  cil managed
    21  {
    22    .entrypoint
    23    .maxstack 1
    24    call int32 main()
    25    call void exit(int32)
    26    ret
    27  }
```

The interesting point is that the `illink` tool infers that the unresolved function references (`printf()` and `exit()`) must be native code, so it automatically manufactures the p/invoke stubs for these functions. Again, to work around the limitation of native calls to variadic functions with Mono, we replace the commented in lines 7 – 8 by the lines 10 – 14. Finally, those three files can be assembled with the Mono `ilasm` to produce a single assembly file, executable with the Mono runtime on GNU/Linux, Microsoft .NET, or any other CLI-compliant virtual execution environment.

4.4 The GCC-CIL Compiler

4.4.1 LLVM versus GCC4NET

LLVM, mentioned before in Section 5.2 in 2008 already had a proof-of-concept CIL generator, and offered mature post-link optimization capabilities. LLVM allows to compile multiple files into its own IR and link them together. This plus its post-link optimization before code generation eliminates the separate compilation steps presented in previous section as far as CIL is involved, and also simplify the native code linking problems. Indeed, at the post-linking step all the compilation units that will be combined into a CLI assembly are available to the code generator. Any other function referenced that is not defined in those compilation units must be implemented as native code in a dynamic library.

Given a C to CIL compiler that can delay the managed code generation for a CLI assembly until all the compilation units are known, there is no need for the DotGNU Portable.NET `ilasm` and `ilalink`. These tools are only required to support the non CLI-compliant assemblies that result from C-style separate compilation, and replacing them avoids problems. For instance, the standard CLI static field mapping feature for data initializations Lidin [2006] is now supported by the Mono `ilasm`, but not by the DotGNU Portable.NET `ilasm`.

Because GCC4NET requires the DotGNU Portable.NET `ilasm` and `ilalink` tools, it cannot benefit from the standard CLI static field mapping. So GCC4NET generates potentially huge static class constructors for data initializations, thus negating the code size advantages of the bytecode program representation, and slowing down program startup. In our experience, these problems are a significant inconvenience when using GCC4NET for real-life embedded software.

4.4.2 The Compiler

We call GCC-CIL the compiler that results from combining GCC with LLVM-MSIL through the DragonEgg library. This library takes advantage of the new plugin architecture of GCC 4.5 to disable GCC code generators and create LLVM ’bitcode’ as output. The purpose of DragonEgg is to replace the `llvm-gcc` compiler that was basically a `gcc-4.2.1` augmented with LLVM optimizers.

Multiple compilation units The high-level view of the GCC-CIL compiler is that it operates like the LCC.NET compiler, except that the results of separate compilation are kept as LLVM ’bitcode’ files instead of CIL text files. In our example, the commands to compile to managed code and execute on Mono are:

```
# separate compilation to LLVM IR files:
gcc -fplugin=dragonegg.so -fplugin-arg-dragonegg-emit-ir -S main.c -o main.ll
gcc -fplugin=dragonegg.so -fplugin-arg-dragonegg-emit-ir -S func.c -o func.ll
```

```
# produce ’bitcode’ files from LLVM IR:
llvm-as main.ll -o main.bc
llvm-as func.ll -o func.bc
# LLVM linking of separate ’bitcode’ files:
llvm-ld main.bc func.bc -o out.bc
# producing CIL assembly from ’bitcode’ file:
llc -march=msil out.bc -o out.il
# generating CIL assembly with Mono ilasm:
ilasm2 out.il
# running the assembly with Mono runtime:
mono out.exe
```

The key advantage of using GCC-CIL is that a multi-file C project can be first compiled to LLVM ’bitcode’ independently for each file. Later the ’bitcode’ files are linked together into a single program, which can be further optimized. In particular, trivial function inlining can be applied at this point, which always improves the resulting code. In C there is also a single name-space for the top-level names, so all the static variables and functions must have unique names. The LLVM infrastructure takes care of this step automatically.

Variadic function calls Previously mentioned in Section 4.3 interfacing native function calls through p/invoke has special requirements when the code will be executed on the Mono platform. With regards to GCC-CIL we use exactly the same technique as we proposed for LLC.NET: in case of variadic functions, such as for instance `printf`, we remove the ’vararg’ keyword and generate function declaration for each different call signature of that function found in the code.

For instance, with the following C source code:

```
printf("Hello world!\n");
printf("i=%d\nj=%d\n",1,23);
printf("String=%s\n","[some string]");
```

The resulting CIL text file contains these `printf()` stubs and calls:

```
//Function declarations
.method static hidebysig pinvokeimpl("MSVCRT.DLL")
int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* ) preservesig {}
.method static hidebysig pinvokeimpl("MSVCRT.DLL")
int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* , int32 , int32 ) preservesig {}
.method static hidebysig pinvokeimpl("MSVCRT.DLL")
int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* , void* ) preservesig {}

//Function calls
...
call int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* )
...
```

```
call int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* , int32 , int32 )
...
call int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* , void* )
```

4.4.3 Correctness

Although a Common Intermediate Language code generator was implemented in LLVM, it was only a proof of concept and in the end due to lack of interest among the LLVM developers, was abandoned. In its current state it was not mature enough to provide correct CIL code for any input C program. Several issues had to be investigated.

We have first identified the problems and corrected a number of errors and missing features that prevented correct C compilation and execution on Mono and Microsoft.NET without LLVM optimizations enabled.

Among lots of minor bugs and incorrectly generated instructions, some of the fixes include:

- lowering of the LLVM intrinsic functions,
- initialization of static data storage containing pointers,
- incorrect printing of float and double constants,
- conversion of integers bit widths from LLVM to those supported by CIL,
- some incorrect function call signatures,
- some wrong values returned by comparison operators,
- incorrect signed operations on integers (extension, truncate, compare, shifts).

Interestingly, the LLVM internal representation does not keep the information about the signedness of integer variables. Rather, it uses operations which interpret those integers as signed or unsigned when needed. On the other hand, CIL distinguishes signed from unsigned for a larger number of operations. As a result, we have to insert signed cast operation to a proper bit width just after loading each argument of such operations on the evaluation stack.

A more serious problem appeared when we enabled the LLVM optimizations, which was traced back to the SSA destruction. The correct way of destructing the SSA form with interferences between Φ-related SSA variables was not known until work by Sreedhar et al. [1999], and the full understanding of those issues was provided by Boissinot et al. [2009] quite recently. For the LLVM-MSIL code generator, we

implemented the Sreedhar Method I, which is not efficient but always safe. The Sreedhar methods rely on insertion of COPY operations to break Φ-related SSA variable interferences. For our simple CIL code generation, we implemented COPY operations by storing to / reading from local variables.

4.4.4 Improvements

Native libraries Beyond correctness, we improved the LLVM-MSIL code generator for better identification of the native libraries where native code functions are defined. By default, the LLVM-MSIL code generator emits p/invoke stubs like:

```
//Function declarations
.method static hidebysig pinvokeimpl("MSVCRT.DLL")
int32 modopt([mscorlib]System.Runtime.CompilerServices.CallConvCdecl)
'printf'(void* ) preservesig {}
```

While the library name mapping feature of Mono reroutes such calls to the C library, this is not sufficient when native code functions are implemented in non-standard libraries. The LCC.NET `illink` tool escapes this problem by probing with `dlsym()` what library provides a particular symbol, given a list of libraries on the command line.

We implemented a different solution, based on a mapping file. For instance, given the following sample file `libs.def` (the ';' line is a comment):

```
;<symbol name>,<real symbol name>,<library name>,<managed|unmanaged>,<module|assembly>
printf,myprintf,mylibrary.dll,unmanaged,module
myfunc,myfunc,myotherlibrary.dll,managed,module
```

This file can be referenced during code generation with the `-msildef` option:

```
llc -march=msil -msildef=libs.def out.bc -o out.il
```

This command instructs the code generator to map the `printf` function to the `myprintf` native code defined in `mylibrary`, and to map `myfunc` to the managed function of the same defined in `myotherlibrary.dll`. So it is possible to specify libraries for symbols without modifying the source code, and this solution applies to both managed and unmanaged code. If no file is specified or the mapping file does not include an external symbol name, code generator will use `MSVCRT.DLL` as default native library.

Reducing local variables Another improvement is reduction of local variables in functions. In its current form, the CIL code generated by GCC-CIL is almost a straight translation of the LLVM SSA-based internal representation. Precisely, each LLVM statement is translated to a series of load, execute, and store operations. This results

with lots of local variables and load/store operations that in fact are redundant, with a negative impact on code size. Our only optimization for now is keeping the result on the stack whenever it is an operand of the following operation, thus reducing amount of local variables and load/store operations.

In the similar fashion, a call to a function that returns a value on the evaluation stack will result in a store to local variable after that call, even if the value is never used. In that case we change the store to pop operation, again reducing amount of local variables.

4.5 Results

	GCC4NET	GCC-CIL	Ratio		GCC4NET	GCC-CIL	Ratio
arrayacc	5144	2948	0.57	jpeg	14440	10500	0.73
autcor	4848	3364	0.69	kmpsearch	5196	3556	0.68
bitaccess	4476	6420	1.43	latanal	5892	7060	1.2
bitonic	5488	3908	0.71	lms	9776	3460	0.35
bitrev	5312	3028	0.57	logop	4308	2244	0.52
bitupck	5588	2852	0.51	lsearch	4568	3444	0.75
bkfircopt	5756	3460	0.6	max	5128	4836	0.94
bkfir	5832	2884	0.49	maxindex	5096	4948	0.97
bsearch	5964	3540	0.59	mergesort	5132	3284	0.64
casetest	4132	2708	0.66	param	4004	1844	0.46
control	5184	3204	0.62	polynome	4328	1844	0.43
copya	4908	3060	0.62	quicksort	4464	3060	0.69
ctrlstruct	4112	2532	0.62	recursive	4088	2308	0.56
cxfir	6400	7780	1.22	sha1	5076	4292	0.85
dct	6700	5428	0.81	shellsort	4516	2724	0.6
dotprod	5120	3124	0.61	squareroot	4120	1924	0.47
euclid	4364	2916	0.67	ssfir	5776	2900	0.5
fft99	11436	4772	0.42	stanford	31668	27492	0.87
fieldacc	3864	1812	0.47	strtrim	4492	2436	0.54
fir8	6256	5204	0.83	strwc	4928	2292	0.47
fircirc	6072	6692	1.1	vadd	5360	4452	0.83
fir_int	10584	2628	0.25	vecmax	4700	2484	0.53
floydall	6080	2724	0.45	vecprod	5004	2468	0.49
heapsort	4616	3076	0.67	vecsum	5132	2484	0.48
iir	5368	3092	0.58	viterbi	7152	4132	0.58

Table 4.1: Comparison of code sizes between GCC-CIL and GCC4NET at `-Os`.

Even though the code GCC-CIL generates could be still improved, it is already significantly better than the code generated by GCC4NET, even though the latter has been specifically improved with a new intermediate form Svelto et al. [2009]. Precisely, we report in Table 4.1 the text segment sizes produced by those two compilers on a series of embedded computing benchmarks optimized for size (`-Os`). In the CIL bytecode

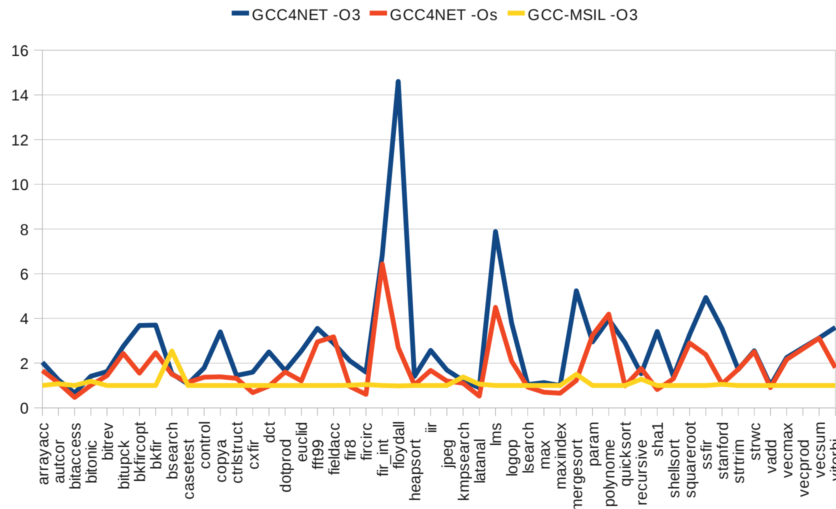


Figure 4.2: CIL Code sizes normalized by CIL code size of GCC-CIL -Os.

representation, the text segment contains both the CIL code and the metadata.

By taking the classic measure of compression ratio (smaller size divided by larger size), it is apparent from this table that the text produced by GCC-CIL is significantly more compact than the text of GCC4NET. Precisely, by summing the text sizes of all compilations from Table 4.1, we obtain 307948 bytes for GCC4NET versus 205624 for GCC-CIL, that is, a 0.67 compression ratio.

In order to investigate the main contributions to the text sizes, we compiled the same benchmarks using maximum optimization level (-O3) in addition to -Os, and we measured both the CIL code sizes and the metadata sizes. In Figure 4.2, we display the GCC4NET -O3, the GCC4NET -Os, and the GCC-CIL -O3 CIL code sizes, normalized by those of the GCC-CIL -Os. In Figure 4.3, we display the GCC4NET -O3, the GCC4NET -Os, and the GCC-CIL -O3 metadata sizes, normalized by those of the GCC-CIL -Os.

We first observe no significant size differences between -O3 and -Os for the GCC-CIL compiler. Indeed, the DragonEgg plugin in its current state bypasses most GIMPLE optimizations. Then, we observe that the metadata sizes show little variation between -O3 and -Os for the GCC4NET compiler, however those sizes are consistently larger than those of the GCC-CIL compiler. We attribute those effects to the post-link optimizations alone. Finally, the CIL code sizes show the most extreme variations, between 0.7 and over 6.4 for GCC4NET and GCC-CIL compared at -Os. The large code size expansions correspond to the cases where GCC4NET initializes nonzero static storage data in the CIL code, instead of relying on the standard CLI static field mapping feature Lidin [2006].

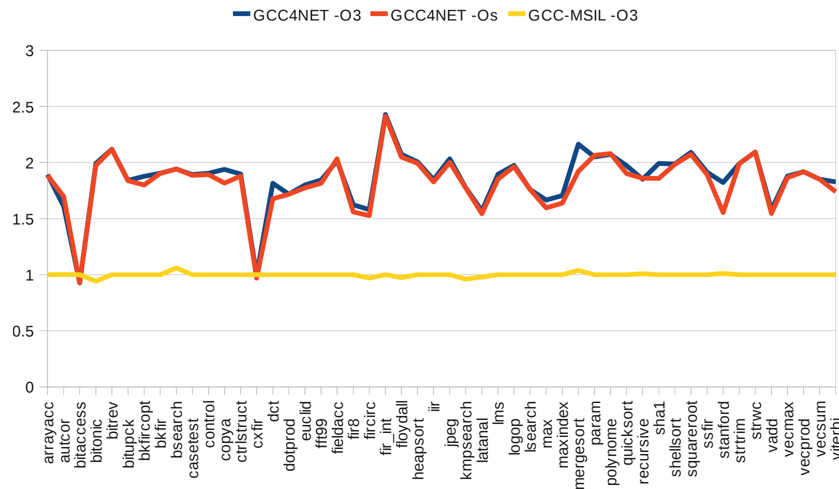


Figure 4.3: Metadata sizes normalized by metadata size of GCC-CIL `-Os`.

4.6 Summary and Conclusions

Our goal was to assemble a production-worthy C to CIL compilation path that leverages the high-level optimization capabilities of GCC, in order to research the areas of dynamic compilation and run-time specialization for high-performance and embedded computing applications.

The significant improvements of the GCC-CIL compiler, although encouraging, did not stopped us from noticing that the LAO-based CLI-JIT compiler (Chapter 5) was spending resources on, among others, instruction selection, function calls lowering and static data layout – steps that seem not benefit much from run-time information.

Also, one can observe the tendency of reducing interest for the .NET platform by its founder, thus reducing effort for its improvements in favor of other technologies; some licence and patent issues reagarding Java after its acquisition by Oracle and the fact that in the popular Android environment its authors although using Java language, do not follow the classical JVM approach.

These problems has led us to a reevaluation of the JIT approach for embedded system and resulted in a concept of a dynamic compilation environment that would target performance rather then processor independence. In Chapter 6 we propose a novel intermediate representation with its interpreter and JIT compiler described in Chapter 7.

Chapter 5

The Framework

In this chapter we describe the framework build and used for the purpose of this thesis. This framework, as depicted on Figure 5.1, allows to compile programs written in C language into the Tirex (see Chapter 6) and its further processing. It could be simple generation of native code, optimization, or even interpretation and Just-In-Time compilation.

The framework consists of three main parts:

- Machine Description System
- Low Level Virtual Machine
- Linear Assembly Optimizer

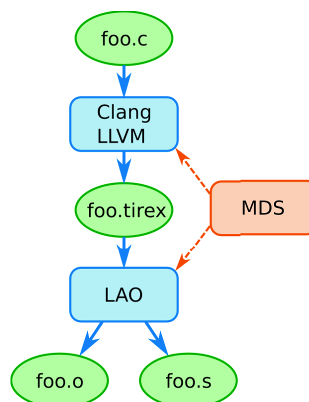


Figure 5.1: The framework

5.1 Machine Description System

The Machine Description System (MDS) is a structured data repository and a collection of programs used to target software development tools to a particular processor architecture family. The software development tools need to implement a number of architecture and processor-dependent tasks:

Encoding and Decoding The binary files that represent a target machine program are encoded by the assembler and the linker, then decoded by the simulator or the debugger.

Assembling and Linking The assembler needs to parse the assembly language syntax in order to produce relocatable binary programs. The linker needs to process the relocations in order to produce the executable binary programs. Both the assembly language syntax and the relocation algorithms are machine-dependent.

Instruction Bundling On a VLIW processor, the instructions are grouped into bundles that execute in parallel. The instruction bundling constraints are usually less regular that the instruction scheduling constraints. In particular, the allowed contents of a bundle may be dependent on the bundle start address.

Instruction Simulation The instruction simulator executes a target machine program on a host machine and provides performance estimates. It needs to model the behavior of the machine at the architecture level and the cycle-accurate level.

Instruction Scheduling The compiler optimizes the target machine program by reordering its operations order. In order to preform this scheduling, the compiler needs an abstract model of the machine resources and of the dependence latencies.

Register Allocation The register allocation of the compiler maps each program variable to either target processor registers or to memory locations. To make these decisions, the compiler needs a complete description of the registers including the cost of moving their contents from / to memory.

Operand Constraints When optimizing the target machine program, the compiler must satisfy the architecture constraints on the instruction operands. In particular: the range of set of values that can be encoded in immediates; the restriction of register specifiers to subsets of register files; the coupling between two register specifiers, such as source and destination in 2-operand instructions, or auto-modified addressing modes.

Instruction Rewriting In this optimization, the compiler matches a pattern of target machine operations and replaces it by a more effective pattern. In its simplest

form this is the so-called peephole optimization (patterns are sequences). In the modern form, patterns are identified on the data-flow graph and matching is enabled by filters on the operands (such as the active bit width).

Instruction Semantics For the purposes of program analysis and optimizations, the compiler needs to abstract behaviors of instructions. One example is constant propagation, the compiler emulates the target processor execution to reduce expressions to constants.

Instruction Attributes Summary information about the instruction properties are needed by the compiler optimizations, such as: control flow instruction, memory access instruction, arithmetic instruction, arithmetic properties, predicated execution, input and output precision, etc.

Instruction Selection The compiler must translate the machine-independent expressions and statements into the target processor instructions. This is usually done by minimum-cost covering of the expressions by tree patterns, where each tree pattern represents a machine instruction.

As illustrated in Figure 5.2, MDS comprises a frontend that processes human-readable machine descriptions to create a Machine Description Database (MDD) as a set of XML tables under the Layman normal form.[1] MDS also comprises backend tools that process the MDD contents and instantiate template files that are then used to build the software development tools. Because several backend processing tools need to share particular views of the MDD contents, these views are created once by a Machine Description Expanded (MDE) contents. Unlike the MDD, the MDE contains redundancy, but is better suited to the needs of the MDS backend processing tools. The common parts between the MDE views of the different processors, in particular the architectural features are then factored by the Machine Description Fusion (MDF).

5.1.1 SSA Form on Target-Level Code

The major trend of code generation is the use of the static single assignment (SSA) form that was previously confined to the target-independent compiler optimizations (before code generation).

5.1.1.1 MDS Support for SSA Form

The SSA form needs to expose all the uses and definitions of the target processor instructions, whether corresponding to encoded operands, or to implicit operands. The MDS exposes the processor instruction set architecture by a three-level hierarchy:

[1] `http://www.ltg.ed.ac.uk/ ht/normalForms.html`

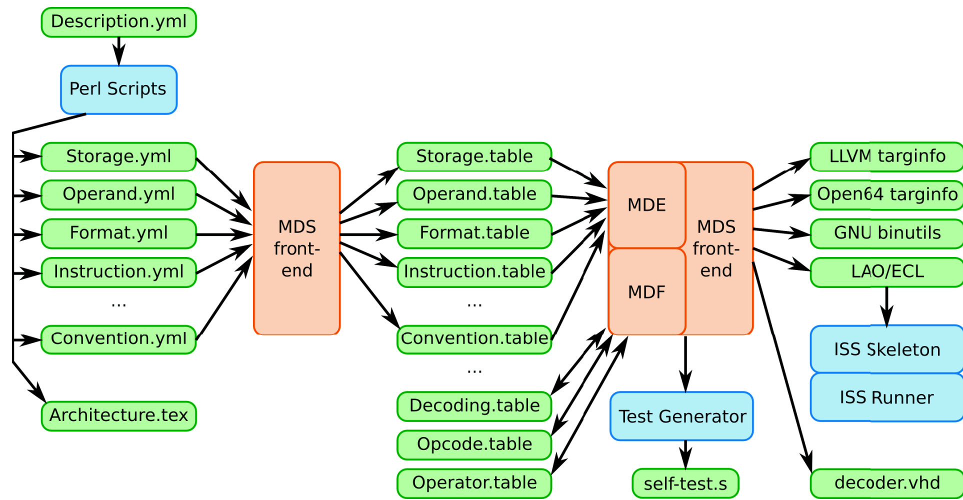


Figure 5.2: MDS (Machine Description System) work-flow.

Instruction Table Element of the processor instruction set, for instance add, load, store, etc.

Opcode Table Results from the database join between Instruction(s), and Format(s). For instance, add two register and add a register with an immediate are two different Opcode(s) of the same Instruction.

Operator Table Results from the expansion of Opcode(s) with regards to Modifier(s), and from the exposition of explicit uses and definitions of the Opcode.

Modifiers are variable fields of instruction encoding that are neither immediates nor register specifiers. They provide a parameter to the instruction behavior, for instance a comparison "`comp`" returning true for equality (`comp.eq`), greater than (`comp.gt`), etc.

Tirex represents code as sequence of *operations*, each operation composed from an operator and the explicit list of uses and definitions. Besides target-dependent operators, we introduce *generic* operators:

- to represent target-level SSA form, including ϕ-functions[2] (`PHI`), and parallel copies with variable number of arguments (`PCOPY`) for operand pinning constraints;
- to represent or recognize in one operation standard computations without having to know the architectural details, e.g., creation of the stack frame at the beginning of a function without knowing if the stack frame size fits in the target architectural immediates (`SPADJUST`);

[2]Note that the order of the arguments matters for ϕ-functions, it is based on the indexes of predecessor basic blocks.

- to write unit tests of code generation algorithms, which need to be target independent, hence we use generic operations to put a value in a register (`COPY`), perform arithmetic operations (`ADD`, `SUB`,...), branch (un-)conditionally (`JUMP`, `GOTRUE`,...), call a function (`CALL`), return from a function (`RETURN`), etc.

As a result, our intermediate representation has the feature of accepting any mix of generic and target-dependent operations.

5.1.1.2 Operand Constraints in SSA Form

Code generation expose the target processor constraints, in particular the instruction set architecture (ISA) restrictions and the calling conventions requirements on register operands. The first general solution to accommodate these register operands constraints in SSA form was proposed by Leung and George [1999]. Before the appearance of the SSA form in code generators, either operands pinned to architectural registers could not be promoted to SSA variables, or register usage constraints were considered as register coalescing problems. However, the SSA form optimizations introduce register interferences that hinders traditional register coalescing, while the register coalescing under SSA form is, according to Rastello et al. [2004], a natural sub-problem of the SSA form destruction problem.

We focus on the register operand constraints on the form of: a *use* and a *def* of an operation must be mapped back to the same pseudo register (before a non-SSA register allocation phase); some or all use and def of an operation are pinned to an architectural register. Boissinot et al. [2009] demonstrated and implemented in the ST200 LAO how to handle these constraints: by introducing parallel copies (`PCOPY` operators) before and after the operation with pinning constraints, then applying a generalized coalescing algorithm.

5.1.1.3 Predicated Instructions in SSA Form

The other issue of using the SSA form in code generators is the support of predicated execution. This problem has not received a general solution until the discovery of the *Psi-SSA* form by Stoutchinin and de Ferrière [2001]; de Ferrière [2007], first implemented in the LAO code generator for the ST120 VLIW-DSP processor. The Psi-SSA form is exploited by the Open64-based ST200 production compilers not only for the classic SSA analyzes, but also for simpler IF-conversion algorithms such as these presented in work of Stoutchinin and Gao [2004]; Bruel [2006].

In the current LAO, we have implemented a simpler alternative to the Psi-SSA form; We let the MDS deduce from the behavior of instructions that the execution of a particular operation is *predicated*, i.e., it has no side effect if some condition of the operand values holds. To emulate this "non-effect," we create an extra use for each definition in the operation and mark each such use/definition pair as constrained to be mapped to the same pseudo register.

5.2 Low Level Virtual Machine

The Low Level Virtual Machine (LLVM)[3] is an open source project written in C++, which aims to be a complete compiler framework supporting lifelong program optimization: compile-time, link-time, run-time, and off-line. It contains modern SSA-based, target and source language independent optimizer, with the code generation support for many architectures. The core of LLVM contains libraries with well defined API and documentation, thus making it good foundation for many different projects, including comercial, open-source and research.

One of LLVM's subprojects is Clang, a C family languages frontend compiler [4] with excelent error and warning messages mechanisms. Clang is also a platform for building source-level tools.

LLVM uses its own intermediate representation designed to be used in three, forms: internal in-memory for compiler purpose; stored on disk as bitcode suitable also for dynamic compilation; and in textual, human readable form. All these three forms are equivalent and could be easily converted from/to each other. LLVM IR is SSA, low-level target independent representation with provided type information.

5.2.1 LLVM in our toolchain

Our target processor is a custom VLIW-DSP processor for which first we have written a LLVM backend. That backend – a part of the compiler that translates the target-independent bitcode into target-dependent code – consists of tables describing the instruction set, register set and calling conventions, and code written in C++ responsible for handling all the CPU related issues. The aim was to emit target assembly as much optimized as possible, that would pass correctly some validation benchmarks. Although the code generator profits from LLVM's target-independent optimizations, it does not contain VLIW instruction scheduler and does not support generation of instruction bundles, hence we emit suboptimal linear code, that would be optimized further in the LAO.

To automate the generation of the backend and to provide synchronization of the target dependent parts (instructions, registers, etc), the tables mentioned above are automatically generated from the MDS. This also facilitates support of modifications in the processor architecture, which is under development, thus constantly changing.

The C to Tirex compilation is done as depicted on Figure 5.3. First the C code is processed by the frontend compiler, Clang, that outputs LLVM's bitcode file. After that, target independent LLVM optimization passes are applied. If all the compilation units are compiled to the bitcode and linked together, the code could be optimized even further due to interprocedural optimizations. After that, instruction selection,

[3] http://www.llvm.org/
[4] Previously LLVM relied on GCC

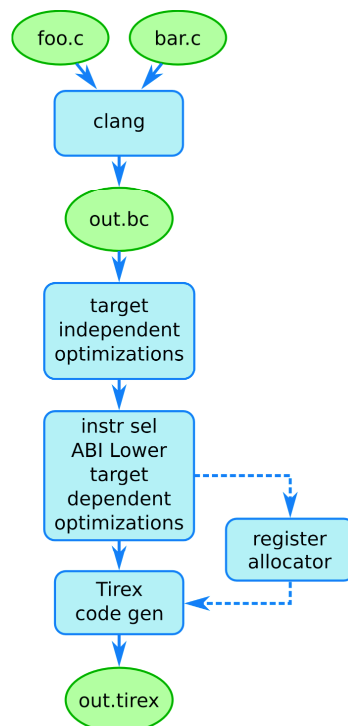


Figure 5.3: C to Tirex compilation using LLVM

ABI lowering and partial prologues and epilogues passes are executed as well as some target dependent optimizations. Optionally registers could be allocated resulting in a fully formed native stream, or the SSA form could be kept. The final step is to generate Tirex code.

5.2.2 The Tirex Code Generator

One of the motivations of the work presented in this thesis was connecting several tools in a compilation chain with the possibility of passing additional information. This resulted in development of the Tirex intermediate representation described in more details in Chapter 6.

Tirex is a descendant of MinIR project, for which a code generator was implemented in LLVM by André Tavares. This code generator, although sufficient for the purpose of MinIR project, was not providing enough information for complete assembly file emission. Moreover, the Tirex specification moved far away from its ancestor. Also, all the features specific to our processor required special care during the Tirex emission. All that issues forced us to reimplement the Tirex code generator from scratch.

The Tirex is very close to assembly language, hence naturally the code generator

is implemented as a subclass of LLVM's machine code streamer and the assembler printer. Additionally some custom passes where written to provide some processor-specific features. The Tirex code can be generated as post- or pre- register allocation pass, thus allowing to output the code in SSA with only partially formed stack frames, or fully formed.

One of the requirements of connecting any compiler to LAO is a definition of basic blocks. In case of LLVM, a call instruction is not a terminator of a given basic block, even though execution of that instruction results in change of the control flow. In contrary, LAO requires that a call instruction should be a terminator of a basic block. For the compatibility reasons, we have written a pass that is executed just before code emission, that splits the basic blocks whenever a call instruction occurs, thus to satisfy the requirements of the LAO.

5.2.3 Current state and limitations

Our previous Tirex code generator was based on the Open64 compiler, in which we have implemented our own pass to construct loop nesting forest with data dependencies and include it in the Tirex output. However, as later we have decided to move our productional compilers to more modern compilation frameworks, the implementation has been abandonded.

In the nearest future we plan to finish the Tirex code generator in GCC framework and complete the implementation in LLVM. For example, the pass for constructing Havlak's loop forest with memory dependencies has to be completed in both LLVM and GCC to provide the same maturity as the implementation in Open64.

5.3 Linear Assembly Optimizer

The *Linear Assembly Optimizer* (LAO) is a tool placed at the end of the compilation chain, which does not contain a compiler for a high-level source language, but rather relies on "external" frontend compiler that provides low-level, target-dependent code and data streams. In the work done for this thesis the function of frontend compiler is performed by LLVM, described in previous section.

LAO was previously used in production compilers for the ST120 VLIW-DSP and the ST200/Lx VLIW processorsas described by Dinechin et al. [2000]; Dinechin [2004]. The LAO was also a platform for an experimental *Just-In-Time* (JIT) compiler for the Common Language Infrastructure (CLI) for the ST200/Lx VLIW processors (CLI-JIT) developed by Dinechin [2008]; Cornero et al. [2008].

LAO is written in C and already have proven itself to be easily portable, especially with the target-dependent parts automatically generated from the MDS (described in section 5.1). Although written in non-object oriented programming language, its construction is object-like providing good encapsulation and a means of self-testing of

each module.

In its early releases, LAO's input was a Linear Assembly Input (LAI) language, which was a superset of the GP32 assembly language with extensions. Later, the connection between LAO and a frontend compiler (Open64 in that case) was done via an API exposed in the LAO. This connection however was fragile and error prone, moreover, replacement of the frontend compiler was difficult due to the fact that the API was shaped to be used by Open64 and factored to its internals. Currently, the Tirex intermediate representation is used (see Chapter 6), which allows easier testing as well as replacement of the frontend compiler or even adding more tools in the compilation chain.

The LAO contains several production-grade instruction schedulers and software pipeliners based on heuristics or integer programming as proposed by Dinechin [2007]. LAO also supports the Static Single Assignment (SSA) form at target level, with innovative high-quality and high-speed SSA form optimizations developed by Boissinot et al. [2008, 2009] and register allocator with split spilling and coalescing phases proposed by Bouchez [2009] in his thesis.

5.3.1 Program processing

As we noted before, the input of LAO is a Tirex program that contains data and program streams along with some additional information (more details in Chapter 6). This file is parsed and kept internally as LAO Intermediate Representation (LIR), and thanks to the explicit structure of Tirex program there is no need for function and basic block boundaries and the data segments identification. **Tirex maps to LIR one-to-one. LIR is just a set of C-structures and data containers internally used by LAO to keep the Tirex instead of YAML or some binary form and provide easy and fast access to all the program fields**.

After that, LAO applies optimizations, where the most important include:

Global Optimizations Constant propagation, dead code elimination and expression simplification.

Loop Restructuring Loop unrolling, mapping to hardware loops.

Pattern Optimizations Recognizing the DSP and other specialized instruction patterns.

IF-conversion Predication of single-entry control-flow regions into superblocks.

Prepass Scheduling Superblock scheduling and software pipelining with modulo scheduling.

Register Allocation Register allocation with split spilling and coalescing phases.

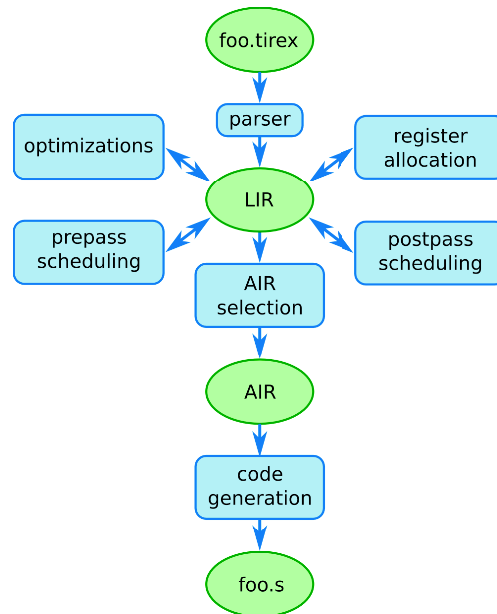


Figure 5.4: Program processing in the LAO

Postpass Scheduling Superblock scheduling.

Finally, after these optimizations, the LIR is mapped to the Assembler Intermediate Representation (AIR) and the assembly code is emitted. LAO can be also used as a Virtual Execution Environment on top of the target architecture. The Tirex program could be interpreted or dynamically compiled (more details in Chapter 7) providing platform for dynamic code optimization and specialization.

5.4 Summary

Working in a multiple tool environment could be complicated when a change in one tool requires changes in all the others, especially when the target architecture is under constant development. In the presented framework synchronization is centralized in one module, thus allows for relatively fast and easy changes whenever needed.

The framework is the base for the work presented in this thesis, including generation of Tirex IR presented in Chapter 6, as well as its virtual execution environment (see 7) consisting of interpreter of the SSA form, a JIT compiler and infrastructure responsible for managing the code generation, code cache and optimizations. It provides the necessary foundation and allows us to avoid implementing all basic services from scratch.

Chapter 6

Tirex

The main objective of this work is to fill the gap between the two major virtual execution environments: the classical JIT and the dynamic optimizers. The idea came from the observation that the Just-In-Time approach spends compilation resources on passes that do not benefit much from the information available in a dynamic compilation environment, in particular: instruction selection, function call lowering, and static data layout. The dynamic optimizers in the other hand leave these tasks for the static compilation phase and focus only on the performance. However, due to lack of higher-level view of the program, their possibility of optimizations is limited.

We believe that the target-level intermediate representation we propose opens a third direction to explore for dynamic compilation. Compared to dynamic optimization, we anticipate significant increases of native code performance, thanks to the availability of global and high-level information. Compared to the JVM, CLI, and LLVM virtual execution environments that embed a bytecode or bitcode interpreter and a JIT compiler, we expect a simpler virtual execution environment thanks to the simplification of mixed mode execution, and a reduction of the JIT compilation resources since the program representation is already lowered at the level of the target processor (see Chapter 7).

Tirex, however, was created because of different reasons than the ones we have mentioned above. We work in a mixed, production and research compilation environment. Such a situation often involves several different compilers and other tools for performing different tasks on the different compilation stages. We maintain three different compilers: GCC, Open64, and LLVM. Although all three of them target the same architecture – a new VLIW-DSP processor that requires advanced optimizations in the code generator – their purpose differs from operating systems and GNU tools compilation, to research on optimization algorithms. These advanced optimizations on the code generator level, in particular, are in the areas of matching complex instruction (e.g., fixed-point arithmetic), register allocation, If-conversion, and global instruction scheduling including software pipelining.

Implementing these target-dependent optimizations three times in three different compilers requires a lot of effort and extensive knowledge of all three of them. Debugging these implementations and making modifications also requires much effort. It seems natural that it would be better to keep all of them in a separate tool, hence, we add an additional, target-specific code generator into our compilation flow.

All these target-specific parts are implemented within LAO tool (Chapter 5) which was previously connected with the Open64 compiler via an API designed specially for the compatibility with Open64's internals. As we moved to LLVM for its modern infrastructure, and to GCC for compiling operating systems with our new VLIW-DSP processor as the target, we were motivated to connect LAO to these three compilers with a more generalized way.

We have decided to create an intermediate representation suitable for passing target-level code with an explicit program structure that would support additional information and be easily extendable when needed. In the end it turned out that Tirex is not only good for connecting different tools, but also for hand-writing tests, program analysis tools, as well as interpretation and dynamic compilation. Moreover it easy very easy extendable it does not need sophisticated parsers.

Tirex is based on MinIR intermediate representation described in more details in Section 6.1. We have selected MinIR because Open64 and LLVM, though with some limitations, were already capable of emitting it when we started this project. Based on YAML, MinIR is human-readable, well structurized and very easily extendable.

Using a target-level intermediate representation for the environment depicted in Figure 6.1 raises a number of challenges, which we discuss and address later, including the following:

- Ensuring that all the tools have a consistent description of the target processor. This is achieved by using a Machine Description System (MDS), which generates all the target-dependent source files for the different tools, see Section 6.2;

- Representing the SSA form on a target dependent code; The challenging areas are the pinning of SSA variables to the architectural registers studied by Rastello et al. [2004], and the representation of predicated instructions presentet in the work of Stoutchinin and de Ferrière [2001], see Section 7.2.3;

- Representing executable programs in Tirex (code stream, data stream), which we present in Section 6.3, with the optional embedding of high-level information that cannot be easily reconstructed by a code generator, for instance loop scoped memory dependences, presented in Section 6.4.

Our contribution presented in this chapter is a extensible, target-level intermediate representation suitable for:

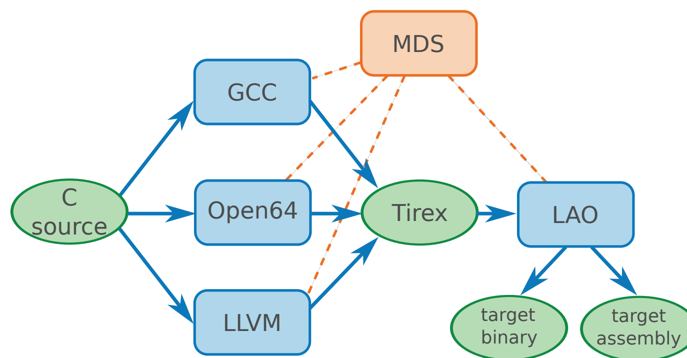


Figure 6.1: Tirex in our toolchain. The MDS supplies target-specific files to build the upstream compilers and LAO code generator. The path from GCC is not yet functional.

- fully functional program representation in SSA,
- data exchange between different tools in the compilation toolchain,
- compilers and tools testing,
- interpretation SSA,
- dynamic compilation,
- program analysis.

6.1 The MinIR project

The MinIR[1] project was started by Christophe Guillon and Fabrice Rastello and work on it was continued by Le Guen et al. [2011]; and stands for *Minimalist Intermediate Representation*. It was intended to be an educational tool that ease the work on compilation research by providing a program representation abstracted from any compiler, language or target. As it is based on YAML serialization specified by Ben-Kiki et al. [2009], it inherits human friendly textual representation of data-structures. Moreover, YAML is easily read and processed by scripting languages such as Perl, Python or Ruby, as well as the C language; they all come with free libraries designed to support reading and editing YAML. Thanks to that it is easy to inspect and modify programs by editing the text files, and read or write it in whichever language one prefers without the need for implementing a parser. MinIR is highly useful for compiler researchers and practitioners interested in trying new algorithms before implementing them in productional compilers, hence without the burden of handling all corner cases typically found in them.

[1] `http://minir.org/`

Thanks to its YAML foundation, MinIR is structured yet extremely versatile. The structure of a MinIR program contains the names of common fields (`functions`, `label`,...), functions are organized into basic blocks (`bbs`) comprising operations (`ops`) that use an operator (`op`) on parameters (`uses`) to define registers (`defs`). Any additional information may be provided in the existing YAML mappings to help the MinIR client or allowing it to perform more advanced optimizations. For instance, a `target` key in a jump operation allow the client to know which basic block is targeted without having to know at which position in the argument list the target is. See the Listing 6.1 for an example of MinIR code.

Since the MinIR format is not fixed, compilers reading this representation also have the possibility of choosing whether to take the information provided. If they cannot handle some of the additional information or do not need it, they simply can ignore it. MinIR can be either sparse and provide only the code stream, or contain arbitrary description, complicated precomputed data for an optimizing compiler. This makes MinIR versatile, easily generated in its minimal version from the internal representation of a compiler, or more complete for a more demanding client, while still being readable by simple compilers. The MinIR format is flexible and easy to extend to fulfill a particular need, while keeping the backward compatibility, hence without breaking the other tools.

Besides the definition of this intermediate representation, the MinIR[1] project provides tools for reading, dumping and verifying a MinIR file, as well as tools for static analysis, experimental register allocation and a SSA form interpreter. MinIR is used as an interchange format between those tools; we show in the next section how we extended it into Tirex to use it not only in a research and educational context, but also as a target-level interchange format in a compiler toolchain.

6.2 Consistent architectural description

As we note in the introduction to this chapter, one of the challenges is ensuring that all the tools have a consistent description of the target architecture. The frontend compiler has to use exactly the same names of registers and the same calling conventions as the backend code generator.

It is often the case that some instructions with the same mnemonic exist in different variants within an instruction set of a given processor. These variants depend on the kinds of operands (i.e., register or immediate) and the size of immediate values; and have impact on how an instruction will be encoded in the binary. As the instruction selection process is done in the frond-end compiler it is better to use a unique identifier (a “shortname”), of instruction’s variant rather than a mnemonic to disambiguate instructions and avoid the need for matching a right variant once more. The mnemonics are defined by the architecture description, however the names of variants are not,

Listing 6.1: Example of a MinIR program from the MinIR project

```
#
# A loop with a test.
#
# x = 0
# y = 0
# while x < 100
# if x < y
# x = x + 2
# else
# x = x + 1
# y = y + 1
#

functions:
  - label: testloop
    entries: [entry]
    exits: [exit]
    bbs:
      - label: entry
        operations:
          - { defs: [X0.R], op: mov.32, uses: ['0'] }
          - { defs: [Y0.R], op: mov.32, uses: ['0'] }
      - label: loophead
        operations:
          - { defs: [X1.R], op: phi, uses: [X0.R<entry>, X4.R<endloop>] }
          - { defs: [Y1.R], op: phi, uses: [Y0.R<entry>, Y4.R<endloop>] }
          - { defs: [B1.B], op: cmplt.32, uses: [X1.R, '100'] }
          - { op: brt, target: test, fallthru: exit, uses: [B1.B] }
      - label: test
        operations:
          - { defs: [B2.B], op: cmplt.32, uses: [X1.R, Y1.R] }
          - { op: brt, target: branch1, fallthru: branch2, uses: [B2.B] }
      - label: branch1
        operations:
          - { defs: [X2.R], op: add.32, uses: [X1.R, '2'] }
          - { op: jump, target: endloop }
      - label: branch2
        operations:
          - { defs: [X3.R], op: add.32, uses: [X1.R, '1'] }
          - { defs: [Y3.R], op: add.32, uses: [Y1.R, '1'] }
      - label: endloop
        operations:
          - { defs: [X4.R], op: phi, uses: [X2.R<branch1>, X3.R<branch2>] }
          - { defs: [Y4.R], op: phi, uses: [Y1.R<branch1>, Y3.R<branch2>] }
          - { op: jump, target: loophead }

      - label: exit
        operations:
          - { op: return }
```

and the naming conventions usually are different for the same architecture in different compilers.

For instance, an add instruction could have different opcodes depending on whether it adds two registers or a register and an immediate. We add a "shortname" field to instructions description; "shortnames" are obtained by appending to the assembly mnemonic the list of types of operands, then any modifier. We then prune the shortname to remove operands common to all variants, see Figure 6.2 for an example of shortnames for an addition and comparison.

Operation	Names concatenation	Shortname
reg = reg1 + reg2	add_r_r_r	add_r
reg = reg1 + signed10	add_r_r_s10	add_s10
reg = reg1 + signed32	add_r_r_s32	add_s32
reg = (reg1 != signed10)	comp_r_r_s10.ne	comp.ne_s10
reg = (reg1 == reg2)	comp_r_r_r.eq	comp.eq_r

Figure 6.2: Shortnames are created by using operand types to disambiguate instructions with the same mnemonic.

Keeping the architecture description centralized makes the maintenance of the tools easier, especially when the architecture is under development and all the details change often making the manual changes in a complex toolchain time consuming. For that purpose, we use the Machine Description System (described in more details in Chapter 5, which allows for synchronization of all the tools in a compilation flow.

6.3 Extensions to MinIR

The extensions to MinIR we have published previously [Pietrek et al., 2011] evolved slightly since, as we have realised that some things could be simplified and others added. A target-level program comprises two mandatory parts: the code stream and the data stream. The root of a YAML file is a "mapping" (a hash table of "key: value" pairs) that contains only the key "functions" in MinIR, listing the functions of the program as a YAML array. We have changed the "root" of the YAML mapping to "program" as now it contains not only functions but also other elements like data objects (object), loop scoped information and dependences (loop),section definitions (section) and architecture description for optimizations (optimize). Instead of having a list for elements of each type, a program simply contains directly each of these elements.

Program elements	
`function`	function description and body
`optimize`	architecture description
`section`	section definition
`object`	objects of the program data stream

Our current extensions to form Tirex are threefold: supporting SSA form on target code; adding data stream to the representation, including unresolved symbols; and allowing more complex program representations to better suit our optimizing needs, in particular to avoid recomputing known data or losing high-level information.

We give an example of Tirex in Listing 6.2 to illustrate the discussions of this section. This example shows a function under SSA form before register allocation with a "for" loop (from 1 to 10) printing at each iteration the induction variable and a float number approximating π. Architectural registers are used because of calling conventions and implicit operands of some instructions, but other computations use temporaries.

Listing 6.2: Dummy example of a Tirex program

```
program:
- optimize:
  processor: xxx # dummy processor architecture
  convention: 3 # convention ID according to MDS
- section: ".text" # section definition
  flags: [ Alloc, Exec ]
  align: 0
- function: callme
  section: .text # function goes in the text section define before
  entries: [ B0 ]
  exits: [ B3 ]
  blocks:
  - block: B0
    flags: [ Entry ]
    successors: [B1: 1]
    frequency: 1 # frequency of execution
    operations: # create a 80-byte stack frame by adjusting the Stack Pointer
    - {op: SPADJUST, defs: [$r12], uses: [$r12, '80']}
    # save callee-save registers in temporaries (incl. return address)
    - {op: PCOPY, defs: [V1-V10], uses: [$ra, $r10, $r13-$r20]}
    - {op: PCOPY, defs: [V11], uses: [$r0]} # get function argument
    - {op: make_s16, defs: [V13], uses: ['1']} # init variable i
  - block: B1
    labels: [.L_001] # additional name for this block
    predecessors: [B0, B2]
    successors: [B3: 0.1, B2: 0.9] # 1/10 chance of branching to B3
    frequency: 10 # block in loop is executed more often
    operations:
    - {op: PHI, defs: [V14], uses: [V13, V15]} # SSA φ-function
    - {op: comp_s10.gt , defs: [V101], uses: [V14, '10']} # is i>10 ?
```

```
    - {op: cb.nez, uses: [V101, .L_002],
       target: B3, # false => branch to .L_002 (i.e., B3)
       fallthru: .next, # true => continue to next block (B2)}
  - block: B2
    predecessors: [B1]
    successors: [B1: 1] # always branches to B1
    frequency: 10 # block in loop is executed more often
    operations:
    - {op: make_s32, defs: [V20], uses: [ [L.float] ]} # make float value
    - {op: lw_r_s10, defs: [V21], uses: [V20, '0']} # from constant pool
    - {op: make_s32, defs: [V110], uses: [ ['L.str'] ]} # address of string
    # prepare call arguments in registers
    - {op: PCOPY, defs: [$r0, $r1, $r2], uses: [V110, V14, V21]}
    # call function using (external) symbol; may clobber some registers
    - {op: call, uses: [ ['printf'] ] implicit_defs:[$ra, $r0-$r9, $r11}
    - {op: add_s10, defs: [V15], uses: [V14, V11]] # increment i by arg
    - {op: goto, uses: [.L_001]} # loop back-edge
  - block: B3
    labels: [.L_002]
    frequency: 1
    operations:
    - {op: make_s10, defs: [V130], uses: ['42']} # prepare return value
    - {op: PCOPY, defs: [$r0], uses: [V130]}     # of function
    # restore callee-save registers
    - {op: PCOPY, defs: [$ra, $r10, $r13-$r20], uses: [V1-V10]}
    # delete stack frame (adjust Stack Pointer)
    - {op: SPADJUST, defs: [$r12], uses: [$r12, '-80']}
    - {op: RETURN, uses: [$ra]} # jump to return address
  loops:
  - loop: L1
    header: B1 # header of the loop
    body: [ B2 ] # body of the loop
  object:
    label: L.float # define π in constant pool
    align: 4        # float are 4-byte aligned
    init:
    - float: 3.14159265e+00
- section: ".rodata"
  flags: [ Alloc ]
  align: 4
- object:
  label: L.str # define new string symbol for printf
  align: 1     # a string may start at any alignment
  size: 24     # string is 24 bytes long (incl. null char)
  section: ".rodata" # belongs to "read-only" section
  init:
  - ascii: "iteration %d, PI is %f\n\0" # data initialization
```

6.3.1 Code Stream Representation

The code stream of the program is partitioned into *functions*, corresponding to functions of the source language. Functions usually belong to the `text` assembly section, but some of them can be put in special sections by the compiler, to be specially managed later by binary utilities. *sections* have to be defined before being referenced in a function (see Section 6.3.2). Functions may also have static storage data, for instance local variables declared with the C `static` keyword, or constants pools.[2] We also added to the code stream loop scoped information, see Section 6.4.

Function level

`section`	*string*	assembly section, defined before
`entries`	*string*	list of entry blocks
`exits`	*string*	list of exit blocks
`blocks`	(see below)	list of basic blocks
`objects`	(see below)	data local to the function
`loops`	(see Sec. 6.4)	loop scoped information

The code stream at this level of representation is not yet linearized. It is still represented as a control-flow graph (CFG), using (labeled) basic blocks that appear in the Tirex `block` field of functions. Instructions are given in the `operations` field, and the program flow is encoded in Tirex in the "jump" instructions through the use of the `target` and `fallthrough` keys. Since upstream compilers may generate several labels for a basic block (e.g., after deleting an empty basic block), we provide a list of additional basic block labels. We also added the possibility to give a list of predecessor and successor basic blocks to easily write unit tests for program-flow analysis algorithms. And finally, we use `frequency` to give profiling information on the execution of a basic block.

Basic block level

`block`	*prefix+number*	number must be unique for function
`labels`	*string array*	additional labels
`predecessors`	*string array*	labels of predecessor basic blocks
`successors`	*array of mappings*	labels of successor basic blocks
	[0.0 – 1.0]	with probability of taking the branch
`frequency`	*float*	normalized number of executions
`operations`	(see below)	list of operations in this basic block

Instructions contain a operation field `op`. To easily distinguish generic operators and avoid any conflict with a target-specific operator, we write them in capital letters, while the target-specific are using "shortnames" described in Section 6.2. Finally, instructions also have `uses` and `defs`.[3] Branch (or "jump") instructions have also the

[2]Constants values that cannot be expressed as immediates in the target architecture, for instance, most constant values on ARM.

[3]MinIR/Tirex also provide an `implicit_defs` array used for instance to list clobbered registers (e.g., caller-saved registers for a function call).

target label of branches. To those, we added the possibility to use the .next keyword for the fall-through since, at this stage of the compilation, basic blocks are usually ordered and the program flow naturally continues to the next block if a branch is not taken. We also added the possibility to supply profiling information on the probability of taking conditional branches using frequency. Each operation could have also node field with a unique number, used in for describing dependences (see Section 6.4).

Instruction level		
op	*string*	target *dependent* and *unique* operator
defs	*string array*	list of definitions
uses	*string array*	list of uses
fallthru	.next	fall to next block if conditional jump fails
node	*integer*	a unique node identifier

The only supported operands are registers, immediate values, and temporaries—variables not assigned yet to registers. We use the convention that architectural register names start with a '$', while temporary variable names start with 'V', followed by a (unique) number. We also need to denote ranges of architectural registers or variables for side-effects of instructions, so we allow range of architectural register or variables using the dash (-) sign.

We added support for unresolved symbols, i.e., address locations not known at compile time. It can be for instance the address of some static data, or the address of a function. Symbols may have an offset (positive or negative integer) and a relocation (for explicit relocations, e.g., get an offset relative to the global data pointer or the thread-local storage pointer). We express unresolved symbols by using a YAML array, where the first element is the name of the symbol and the two other elements are optional strings starting with either +, -, or @.

Operand level	
$r42	register number 42
V102	unassigned temporary
$r0-$r7	all registers with numbers between 0 and 7
[printf]	use a function symbol as argument for a call
[foo, '+12']	address of 12th byte in object foo
[bar, '@TP']	offset of bar relative to the thread pointer

6.3.2 Sections

In order to emit correct assembly or binary file, the code generator has to know in what sections parts of the program should be emitted. A *section* definition should contain a unique name, a list of *flags* and *alignment*. Elements of Tirex program, such as functions or objects, should reference only one *section*, which was defined before in the Tirex program.

Section definition		
`section`	*string*	a unique name of a section
`flags`	*string array*	list of flags
`align`	*integer*	alignment of this section

Listing 6.3 shows example of section definitions and their references by other elements of a Tirex program.

Listing 6.3: Example of section definitions.

```
#sections definition
- section: ".text"
  flags: [ Alloc, Exec ]
  align: 0
- section: ".data"
  flags: [ Write, Alloc ]
  align: 4
- section: ".rodata"
  flags: [ Alloc ]
  align: 4

- object:
  section: ".data"
  origin: 0x0
  label: "tab.2488.2.5" # 0x0
  init:
  - word: [ 0x1 ]
  - word: [ 0x2 ]
  space: 8
  # end of initialization for "tab.2488"
- object:
  section: ".rodata"
  origin: 0x0
  label: "q.2486.2.3" # 0x0
  init:
  - word: [ [".rodata", +8] ]
  # end of initialization for "q.2486"
- object:
  section: ".rodata"
  origin: 0x4
  label: "r.2487.2.4" # 0x4
  init:
  - word: [ [".rodata", +16] ]
  # end of initialization for "r.2487"
- object:
  section: ".rodata"
  origin: 0x8
  init:
  - string: "abcd"
- object:
  section: ".rodata"
```

```
origin: 0x10
43 init:
- string: "xyzt"
```

6.3.3 Data Stream Representation

The original MinIR does not include the program data stream. We propose to describe the data stream in the YAML format, using a similar structure as data sections at assembly level. We use the key `objects`, appearing either at the Tirex root level (same as `functions`), or inside a Tirex function for local data.

Each "object" consists of a number of bytes stored in memory. Since the actual object address is unknown at compile time, it is represented by a symbolic name, i.e., a unique string in the compilation unit. Object layout in memory is constrained by the type of data contained in the object, so we provide the memory alignment in our intermediate representation. Different data sections can hold objects, for instance, objects in `rodata` are read-only, and those in the `bss` section are zero-initialized at program launch. Finally, we can pass additional attributes (e.g., "global," "static," or "external" flags) with compiler-defined keywords, and objects can have initialization values.

Symbol specification		
`label`	*string*	symbol name
`size`	*integer*	size in bytes
`section`	*string*	assembly section reference
`origin`	*integer*	offset from the beginning of the section
`align`	*integer*	alignment of the symbol
`attr`	*string array*	optional list of attributes
`init`	(see below)	optional initialization of data

If an object is initialized, all bytes must be specified. In this case, the `init` key provides a YAML list of "`type: value`." Although it is possible to specify all static data initializations using the `byte` type, we provide other data types so that the initialization stays human-readable and modifiable, and allows for instance to recover field values in C structures. If some data field is a pointer, its initializer may be a relocatable symbol instead of an absolute value, see the table below and the initialization of the `ptr` field of the C structure in Figure 6.3 for an example.

```
struct s {
  char str[16];
  int i;
  short s;
  float *ptr;
  long long l;
  float f;
  double d;
};

struct s foo = {
  "Hello␣world!\n",
  −2,
  −1,
  &foo.f,
  123456,
  2.1,
  22.1234
};
```

C code

```
− section: ".data"
  flags: [ Write, Alloc]
  align: 4
− object:
  section: ".data"
  origin: 0x0
  label: foo
  align: 8
  size: 52
  init:
  − ascii: "Hello␣world!\n\0\0\0"
  − s32: −2
  − byte: 0xff
  − byte: 0xff
  − space: 2
  − word: [foo, '+40']
  − s64: 123456
  − f32: 2.100000e+00
  − space: 4
  − f64: 2.212340e+01
```

Tirex data

Figure 6.3: A global structure in C on the left and the corresponding object in Tirex on the right. The initialization is not unique, the `short` −1 uses two bytes to form `0xffff`, but could have been "`s16: -1`" or "`u16: 65535`." The "`space`" where added to satisfy alignment constraints of the pointer (4) and the double float (8).

Data initialization		
`byte`	*hex string*	8-bit hexadecimal value (e.g., `"0x9f"`)
`word`	*hex string**	32-bit hexadecimal value
`quad`	*hex string*†	64-bit hexadecimal value
`s8 / u8`	*integer*	8 bits signed/unsigned data
`s16 / u16`	*integer*	16 bits signed/unsigned data
`s32 / u32`	*integer**	32 bits signed/unsigned data
`s64 / u64`	*integer*†	64 bits signed/unsigned data
`f32 / f64`	*float*	32/64-bit float / double float data
`ascii`	*string*	non null-terminated string of bytes
`space`	*integer*	pad a number of bytes with zeros
*can also be	*symbol*	32-bit address unknown at compile time
†can also be	*symbol*	64-bit address unknown at compile time

6.4 Loop Scoped Information

In Tirex, we use the `loops` key to describe the loops within a function. A `loop` contains its identifier, `flags`, an identifier of `header` basic block, description of `body` of the loop

and a `parent` loop identifier in case of nested loop. Memory dependences between instructions are defined using a list of `nodes` and `arcs` connecting them. A unique node must be previously added to an instruction within a basic block. The arcs contain two node, and type of the dependence : `flow`, `anti`, `output`, and `input`.[4]

Listing 6.4 shows an example of a program with nested loops. On Listing 6.5 are fragments of the resulting Tirex code which show nodes naming and loop scope definitions.

Loop scope information		
`loop`	*string*	loop identifier
`flags`	*string array*	extra flags
`header`	*string*	this loop info header block
`parent`	*string*	identifier of parent loop if nested
`body`	*string array*	list of body basic blocks and nested loops
`nodes`	*string array*	list of nodes in this loop
`arcs`	*array*	arcs defined between the nodes

Listing 6.4: Example of a nested loop in C.

```c
void main(int *A, int *B, int N) {
  int i, j;
  for (j =0;j<N;j++) {
    A[j] += B[j];
    for (i=0;i<N-2;i++) {
      B[i+2] = B[i]+B[i+1];
    }
  }
}
```

Listing 6.5: Loop scoped dependences example.

```
1
  - block: B5
    frequency: 68.9708
    successors: [ B28: 1 ]
    predecessors: [ B4 ]
6   labels: [ ".L_BB5_main" ]
    operations:
    - { op: mov_r_r, defs: [ T187 ], uses: [ '$r0' ] }
    - { op: mov_r_r, defs: [ T184 ], uses: [ T124 ] }
    # operation marked as node N1
11  - { op: ldw_i, defs: [ T189 ], uses: [ '4', T124 ], node: N1 }
    # operation marked as node N2
    - { op: ldw_i, defs: [ T190 ], uses: [ '0', T124 ], node: N2 }
    - { op: mov_r_r, defs: [ T160 ], uses: [ T189 ] }

16 - block: B7
```

[4]"Input" is not an actual dependence, but is used to detect and remove unnecessary loads.

```
    frequency: 6897.08
    successors: [ B22: 0.01, B7: 0.99 ]
    predecessors: [ B28, B7 ]
    labels: [ ".L__0_14" ]
    operations:
    - { op: add_r, defs: [ T160 ], uses: [ T193, T160 ] }
    - { op: cmpne_b_r, defs: [ T163 ], uses: [ T187, T142 ] }
    # operation marked as node N3
    - { op: stw_i, uses: [ '8', T184, T160 ], node: N3 }
    - { op: add_i, defs: [ T184 ], uses: [ T184, '4' ] }
    - { op: mov_r_r, defs: [ T193 ], uses: [ T190 ] }
    - { op: mov_r_r, defs: [ T190 ], uses: [ T160 ] }
    - { op: add_i, defs: [ T187 ], uses: [ T187, '1' ] }
    - { op: br, uses: [ T163, '.L__0_14' ] }
  - block: B4
    frequency: 97.07
    successors: [ B26: 0.289474, B5: 0.710526 ]
    predecessors: [ B2, B8 ]
    labels: [ ".L__0_9" ]
    operations:
    # operation marked as node N4
    - { op: ldw_i, defs: [ T147 ], uses: [ '0', T175 ], node: N4 }
    # operation marked as node N5
    - { op: ldw_i, defs: [ T146 ], uses: [ '0', T178 ], node: N5 }
    - { op: add_r, defs: [ T148 ], uses: [ T146, T147 ] }
    # operation marked as node N6
    - { op: stw_i, uses: [ '0', T178, T148 ], node: N6 }
    - { op: br, uses: [ T141, '.L_BB5_main' ] }
# loops description
  loops:
  - loop: L1 # loop identifier
    flags: [ Inner ] # Inner loop in the loop nest
    header: B7 # header block
    parent: L2 # parent loop identifier
    body: [ B7 ] # body basic blocks
    nodes: [ N3 ] # nodes marked inside the loop
    arcs: [ ] # no arcs between nodes in this loop
  - loop: L2
    header: B4
    body: [ B4, B5, B28, [L1], B22, B26, B8 ] # list of basic blocks and nested loops
    nodes: [ N4, N5, N6, N1, N2 ] # nodes marked in this loop
    arcs: # arcs definition
    - [ N4, N6, anti, 1, 0 ] # anti dependence between N4 and N6
    - [ N5, N6, anti, 1, 0 ] # anti dependence between N5 and N6
    - [ N6, N2, flow, 1, 0 ] # flow depencence between N6 and N2
    - [ N6, N1, flow, 1, 0 ] # flow depencence between N6 and N1
```

6.5 The use cases

Tools connection The first goal of the Tirex representation was to connect different frontend compilers with the LAO optimizer. We have succesfully implemented Tirex code generator in Open64 and LLVM after register allocation and before (SSA form). The Tirex generator in GCC currently is implemented after register allocator, but the work on the SSA Tirex form is ongoing. The generated code can be succesfully loaded by the LAO and processed further, i.e., the program can be optimized and target assembly generated. Specially modified compilation drivers allow for using the Tirex step and connection to LAO transparently for the user.

What is important, as opposed to other existing IRs, Tirex allows to pass arbitrary information when needed – it is very easy to add more extensions to the code generator without supporting it in all the consumers except the one that actually uses it. A good example of such extensions is the loop scoop information that we pass from the front compiler to the LAO.

Tirex in compiler testing One of the difficulties when writing and debugging compilers is to find how to test specific parts of a compiler. Compiler phases are usually deeply intertwined with other optimizations and the closer it is to the backend, the farther it is from the frontend, hence the more difficult it gets to construct a working high-level example written in source language. Indeed, every change in the flow of compilation can slightly modify the intermediate representation, and the test might then not work as expected. Worse, some parts of the compiler might become untested without the programmer even realizing it. To complicate matters, our LAO compiler receives its input from another compiler, which makes it even more difficult to generate the test cases we need.

Still, unit tests are mandatory in a production compiler, and the only alternative we had prior to using Tirex was to explicitly construct the intermediate representation of test programs on which to run our algorithms. For instance, to create a simple register assignment “`$r4 = 42`” in a LAO unit test, we had to perform the six following steps: create a new operation with one argument and one result, set the operator to `COPY`, create a new temporary of type “immediate” with value 42, set it as the argument, create a new temporary of type “assigned” to register 4, set it as the result.

Having a Tirex reading capability in LAO allowed us to rewrite most of our tests in Tirex files given as input of the self-tests, making it easier to understand existing tests and keeping them up-to-date with regards to the functionality they are testing. This is also true for control-flow analyzes by using the Tirex `predecessors` and `succs` keys in basic blocks, hence giving the possibility to have empty blocks, which would not be possible in a C source file.

Virtual execution environment While working on Tirex we have noticed that it could be also used as an intermediate representation for a virtual execution environment, in particular it could be interpreted or native code could be generated dynamically. Tirex is target-dependent, i.e. the instruction set and ABI is exactly the same as for the machine that hosts the VEE, but contains arbitrary information that could be used by a dynamic compiler. This simplifies the interpretation and dynamic compilation processes, but also makes the interoperability between native code and interpreter much easier. We continue the discussion in the following chapters.

6.6 Summary

We have extended the specifications of MinIR to make it a target-level intermediate representation for exchanging information between compilers (Tirex). Extensions include adding support for passing the data stream as well as loop scoped information such as memory dependences to enable code generator optimizations. With these extensions, we were able to connect multiple upstream compilers (LLVM, Open64, and soon GCC) to the LAO code generator, thus factoring target specific optimizations; It can also be used as input of JIT systems lighter than those working on generic representations since it is already at a target level; Finally, we also use Tirex to write unit tests for the LAO code generator.

To conclude, we proposed a novel approach to JIT compilation, using a target-level instruction stream in SSA form augmented, in particular, with loop scoped information. This opens the door for exploration of techniques considered too expensive for runtime. Our work on an SSA form interpreter and JIT compiler is described in Chapter 7.

Chapter 7

The Tirex Runtime

In Chapter 1 we have already presented the main branches of dynamic translation techniques, that is Just-In-Time and dynamic optimizers. The former is using a target-independent IR for execution via interpretation and native code generation, and often optimizes the native code by making use of information gained at runtime. The latter also uses dynamic information to gain more performance, but rather focuses on existing, target-level binaries.

It became apparent since the Self system introduced by Hölzle and Ungar [1994]; Hölzle [1995], that achieving a good compromise between compilation speed and code quality requires dynamic instrumentation or sampling techniques, that would allow to decide what and when compile, and how to optimize it. This could be done by interpretation, which facilitates the gathering of dynamic information during program execution, as it is easier to implement profiling mechanisms along with interpretation routines than inserting special code inside generated binary. Thanks to that, interpretation is still a vital part of most of the modern virtual execution environments that host dynamic optimizers [Bala et al., 2000], binary translators [Desoli et al., 2002] and Just-in-Time compilers. Byte-code interpreters and JIT compilers are nowadays in widespread use thanks to the Java programming language [Gosling et al., 1996]. Although it is less efficient than direct execution of native code, it does not incur the time and space overhead of running a compiler, which makes it beneficial on non frequently executed portions of the program. This makes the coexistence of interpreters and JITs reasonable and justified, and leads to introduction of virtual execution environments such as the Java HotSpot engine designed by Oracle [2010].

Mixed-mode execution A problem inherent with interpreting target-independent representations in a virtual execution environment is the *mixed-mode* execution investigated by Agesen and Detlefs [2000]. Processor calling conventions and data layout rules (the processor ABI), and byte endianness, are usually not the same for the IR and the underlying platform. This makes the calls to native functions (JIT compiled

or from libraries) from within the interpreted program problematic. The interpreter has to call these functions via a *trampoline*, which is a special piece of code that makes the function arguments compatible with the ABI requirements of the target processor, and provides the result of a call compliant to the ABI of the interpreted IR. This task could be more difficult depending on the complexity of the data structures passed as the arguments and results.

The SSA form Modern compilers, including JIT compilers, exploit the advantages of the Static Single Assignment (SSA) form. Whenever dynamic optimizers try to improve performance of an existing binary containing native code (obviously not in SSA), they have to transform this code into SSA to be able to benefit from the SSA optimizations.

With regards to the SSA form, the target-independent IRs are in a slightly different situation – they are not directly executable by a processor but rather by a virtual machine. Although such IRs could be SSA by design, in practice they are not, with the exception of the Low Level Virtual Machine bitcode. We think that one of the reasons could be the fact noticed by Gal et al. [2005], that the size of code increases when transformed to the SSA form because of increased number of variables and the introduction of ϕ-functions in the instruction set. This of course is undesirable in embedded systems with a constraint storage space.

Also, the bytecode IRs of the Java JVM or the CLI were designed before the properties of SSA were explored in virtual execution environments, and with an efficient interpretation in mind. Krintz [2002] reported that for effective use of the SSA optimizations it would be better to have that form prepared statically, hence for the compatibility reasons, he proposed to store and distribute two versions of a program together: in SafeTSA (an IR in the SSA form developed by Wolfram et al. [2001]) for compilation and in the Java bytecode for interpretation.

Stack versus Registers Most of the IRs executed on virtual machines are designed for stack-based machines. However, as we reported in the introduction chapter, nowadays most of such execution environments convert that stack-based IR into a three-address, register based IR, usually in SSA for the JIT compilation purpose. Also, Shi et al. [2008] reported that although stack-based IR is smaller, register-based IR requires less virtual machine instructions to be interpreted, hence the execution time is smaller.

Tirex on the other hand, while being target-level is already in the three-address form, moreover it supports SSA flavor, explicitly maintains the program structure, keeps data objects separate, and allows for additional high-level information. Currently, this includes loop nesting forest and loop-scoped memory dependencies. Such information ensures that program specialization and aggressive compiler optimizations can still be applied. Compared to target-independent IRs such as JVM and CLI

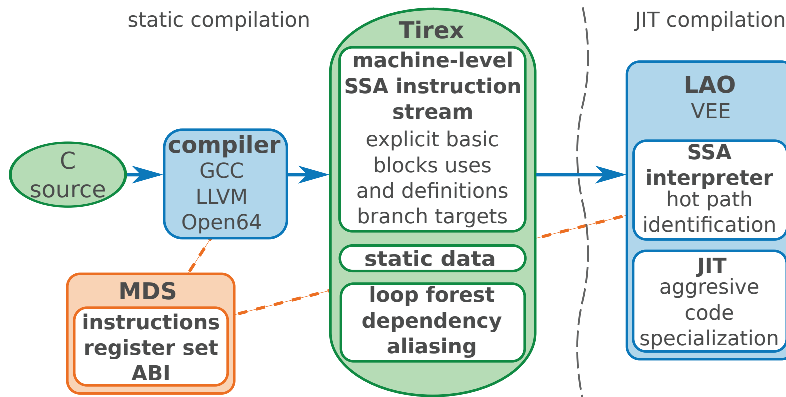


Figure 7.1: Tirex for mixed mode execution and JIT compilation.

bytecodes, or LLVM bitcode, Tirex needs more storage space and gives up the target processor independence, but allows to move the burden of code lowering to processor ABI and instruction selection back to the upstream compiler, with the goal of increasing the budget available for run-time compiler optimizations, thus targeting performance. Also, the target-level IR simplifies interoperability between the interpreted and JITed code, as well as external native libraries by having the same ABI and allowing to share between all three of them the run-time stack and global data.

Motivated by the limitations of both JIT and dynamic optimizers, and by the lack of interpreters of the SSA form, in particular a target-dependent low-level intermediate representation, in this chapter we present our work on a virtual execution environment that fills the gap between these two aforementioned systems. It is designed to execute Tirex by interpretation and run-time compilation. We also discuss the problems related to SSA form interpretation and present our solutions. We provide a *proof-of-concept* JIT compiler for Tirex and talk about issues with its dynamic compilation and code cache management. Finally, we provide a framework (see Figure 7.1) that would allow for experimentation with aggressive dynamic optimizations on a very low, target-level SSA program representation.

The properties of Tirex allow to eliminate the mixed-mode execution problems related to ABI mismatch. Thanks to SSA form interpretation, we avoid SSA construction overhead as well. Tirex is a three-address register-based IR, hence there is no need for conversion from stack-based to register-based during runtime. This leads to a simpler virtual execution environment that can reduce the execution and compilation time.

7.1 Overview

The Tirex runtime is implemented on the top of the *Linear Assembly Optimizer* framework (see Figure 7.2). Its main blocks are the parser, the execution engine, the inter-

preter and dynamic code generator. What is interesting, in such system the run-time stack and the global data can be shared among the interpreter, JITted code and external native functions.

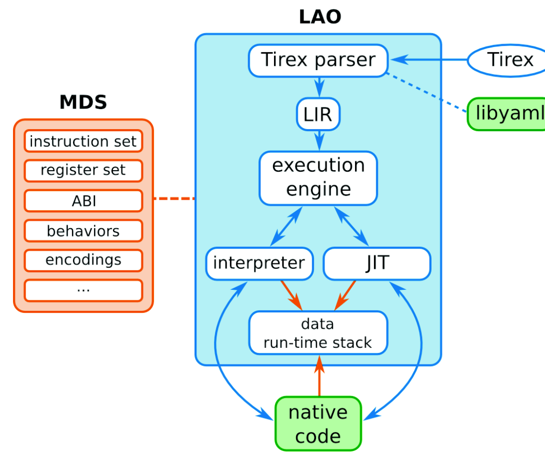


Figure 7.2: The Tirex virtual execution environment.

Parser The Tirex representation, being a YAML document when in textual form, can be easily parsed by standard tools like `libyaml`. However, the LAO only uses `libyaml` to tokenize the Tirex file. A recursive-descent parser directly builds the code and data streams in the LAO internal structures. This design ensures that, by just changing the tokenizer, a binary encoding of a Tirex representation could be read with low overhead.

The Execution Engine The Figure 7.3 shows the initialization process of the execution engine (EE) in JIT-only mode. In case of mixed-mode execution with both interpreted and compiled code, depending on the profiling data it either runs the interpreter or triggers the JIT compilation and executes the resulted code. Its role in the future will be also to decide when to recompile some parts of the code and apply different optimizations.

After loading a program the EE allocates memory space for global data and resolves its addresses. If the data in Tirex program contains initial values the allocated memory is set accordingly. After that, in the JIT-only mode the EE looks up the main function which is used as the entry function and calls the JIT compiler. When the compilation process is over, the main function is called.

Global data If a program contains the (global) data stream, after loading the program and before the interpretation, memory space is allocated and the symbols related to the data objects are resolved. Later, during interpretation, addresses of referenced

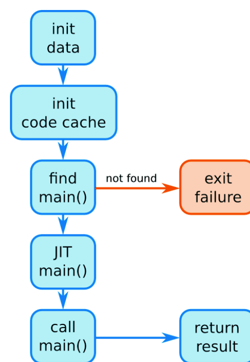


Figure 7.3: Execution engine initialization.

data are provided via the symbols. If a data object contains an initializer, the allocated memory is initialized according to it. As the interpreted and native code use the same ABI, there is no need of conversion when native (JITted or from libraries) functions are called, hence exactly the same memory space is used.

The run-time stack The run-time stack is used to keep the stack frames of functions. It is an array of memory that reflects an internal state of the interpreter. The run-time stack of the interpreter should not be confused with the evaluation stack in stack-based Virtual Machines (Tirex is a target-level IR, hence register-based). Similarly to the global data, the run-time stack is shared by the interpreter and the native code.

7.2 Interpreter

The program interpretation is initialized by the execution engine, depending on given options. A program can be run in the interpreter-only mode, JIT-only mode, or mixed-mode. The interpreter support SSA form of Tirex, but can execute also a *Out-of-SSA* Tirex allowing to keep some parts of the program fully formed. The core of the interpreter consists of:

Behavioral functions The MDS uses the target processor description to generate a set of behavioral functions. These functions are divided into three phases: fetch, execute and write-back. While being initially designed to be used by the instruction set simulator (ISS), these functions are also used by our interpreter, which we describe in more details in Section 7.2.1

The register Set The MDS provides the register set description and ABI register conventions for the LAO. The register set description leads to the creation of an array as a part of the interpreter. Elements of this array are then used by the behav-

ioral functions during interpretation, as if they were the architectural registers, to store the computed values.

The trampoline The trampoline is a special function written in the assembly language of the target architecture to ensure that no stack or register operation is performed by the compiler. As arguments it takes three addresses that point to: the interpreted register set, the interpreted stack, and the called function. It is used to prepare the processors registers and to switch the stack before a function call. We talk about the whole process with an example in Section 7.2.2.

7.2.1 Interpreting Instructions

An age-long debate whether a stack-based architecture or a register-based better suits the interpretation did not provide general conclusions until work of Shi et al. [2008]. He shows that although stack-based code is smaller, the register-based requires fewer executed instructions leading to significant speedups. The Tirex, being target-level, naturally is register-based, thus the interpreter works with SSA variables and interpreted registers and the run-time stack used for local function storage.

The interpreter is implemented as a threaded interpreter, which executes so called *instruction behavioral* functions that correspond to instructions in the Tirex form. These functions are automatically generated by MDS and were designed to be used by the instruction set simulator, but are suitable for interpretation purposes as well.

The Tirex IR does not contain entry directive, but rather the interpreter assumes that a main function is the entry of a program. Hence, after loading Tirex, the interpreter searches for the `main` function and starts the execution from the first instruction of this function. Interpretation of branch instruction is extended to process ϕ-functions after executing behavior function for branch; Details are explained in Section 7.2.3.

ϕ-functions, call and return instructions are not interpreted using automatically generated behavior functions, but by custom code. Call instructions are treated differently depending on whether the target is a function in the interpreted Tirex program or some native code. In the first case a new context (memory space reserved for the values of a function, see Section 7.2.3) is created and the target function is interpreted afterward. When the return instruction is interpreted, the context is destroyed and the interpreter goes back to the caller. If the target is a native function, a trampoline is called to prepare the registers and stack as described in Section 7.2.2.

7.2.2 Calling Native Code

The fact that Tirex is already a target-level representation simplifies significantly interoperability between the interpreter and the native code, also called *mixed-mode* execution. Usually, when the interpreted intermediate representation is target-independent,

the ABI differs for native and interpreted code. This, plus in some cases, different endianness and different sizes of types, require the interpreter to emit special code that prepares the parameters passed to the native code and to get back the correct result.

Tirex, in the other hand, is already target-level with the same endianness, data layout, calling conventions and sizes of parameters, hence the problem of mixed-mode execution simply does not exist. To perform the call, the interpreter uses a trampoline, which has to perform only a small number of tasks before:

1. Store the return address and stack pointer in memory
2. Copy arguments in interpreted registers to processor's.
3. Point the processor's stack pointer to the interpreter stack
4. Call the function (using indirect call, i.e., the address of the function)

and after:

1. Copy the returned values in processor registers to interpreted registers
2. Restore original return address and stack pointer

Depending on the call signature, the target processor architecture and its ABI, parameters can be passed through registers, on the stack or both. An interesting property of interpreting the Tirex representation is that the parameters are already prepared by instructions preceding the call instruction. Parameters required to be in registers will be put in interpreted registers, and those required to be on the stack will be placed on the interpreter stack. Then, during the interpretation of a call instruction, the *trampoline* function is called by the interpreter.

As an example, lets assume that the ABI requires up to four parameters to be passed in registers `r0` to `r3` and the rest on the stack. Similarly a function should return the result in up to four registers `r0` to `r3` or in the buffer allocated by the caller on the stack if the result is bigger. An example of Tirex code just before a call could be seen on Figure 7.4. While interpreting the code, each parameter is prepared in the interpreted (not physical) register set and on the interpreter stack if necessary.

As illustrated on the Figure 7.5, inside the trampoline the processor return address and stack pointer are saved in memory. At this moment the processor stack pointer keeps the frame of the function that interprets the Tirex code. We set it to point to the run-time stack of the interpreter. In other words, the real stack is switched with the virtual run-time stack of the interpreter for the duration of the native call. This allows interpreted and native code to effectively use the same stack. After this, the interpreted registers specified in the ABI to pass the parameters are copied to their processor equivalents. Finally the function is called using the provided address.

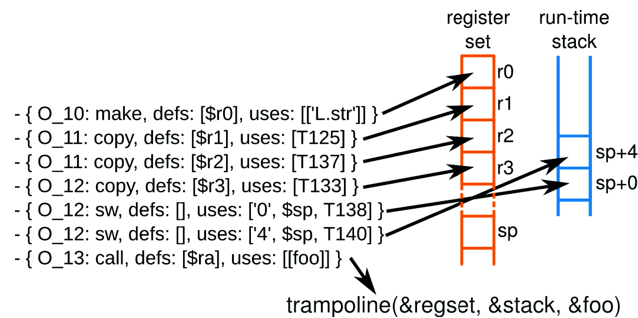


Figure 7.4: Example of a call in Tirex

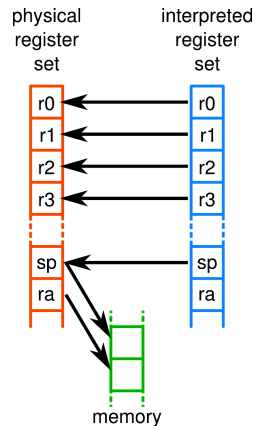


Figure 7.5: Passing arguments before the call, saving stack pointer and return address and switching stack pointers

After returning from the called function, only the return address and original stack pointer are restored, as could be seen on the Figure 7.6. Also the processor registers containing the result are copied to their interpreted equivalents. If the result was passed on stack, there is no need to do anything as the run-time stack was switched with the interpreted stack for the time of call.

The interpretation of target-level IR on the target architecture itself removes the need to take care of any other registers in the trampoline. The caller-save processor registers are already saved by code generated during the compilation of the interpreter itself. As the trampoline does not use any register but the argument passing ones, it does not matter if the called function uses the physical caller-save register or not.

7.2.3 Interpreting the SSA Form

The SSA form can be challenging for the interpretation process. The main property of the SSA is that each variable has exactly one static assignment in the program. In other words, a variable in a program can occur on the left side of only one instruction

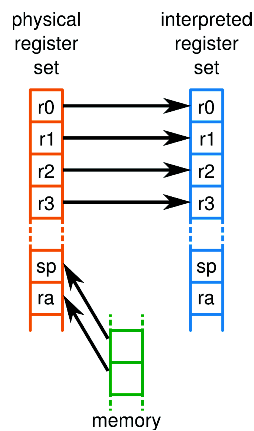


Figure 7.6: Copying the result to the interpreted registers after the call and restoring stack pointer and return address

in a program, possibly leading to a large number of variables. Another difficulty for interpretation is the notion of ϕ-functions introduced to select and assign values at the beginning of a basic block, depending on the incoming edge in the control flow.

Interpreted registers and contexts A source program uses a finite but unbounded number of variables, while native code uses only a fixed (and small) number of registers. The process of mapping those variables to either memory or registers is called register allocation. This step is one of the last performed during compilation, and a program under SSA form still uses variables; Moreover it use even *more* variables than the original program because of the SSA construction. Hence the memory requirements to store all the values during interpretation can be large.

Furthermore, we must ensure that the values in variables do not get erased by mistake. The variables are given unique names so that different functions cannot overwrite the values of their variables. However, these variables are not divided into caller-save and callee-save as the processor registers. So, if a function is called multiple times in different contexts, e.g., in the case of recursive functions, the variables in this function will be overwritten. This makes the memory requirements even bigger, especially in case of recursive calls.

Although a program in SSA could have large number of variables, which of them will be actually used depends on the control-flow. So the amount of storage space, obviously with a cost of dynamic allocation, could be limited to only required values. Each time a function is entered, we create a new context on the context stack in similar fashion as a new stack frame. A context is destroyed when control reaches return instruction. A storage space for a value is provided on demand when needed in a current context. As the amount and order of values inserted into a context could differ, a context is implemented as a hash table to allow fast access to a required value.

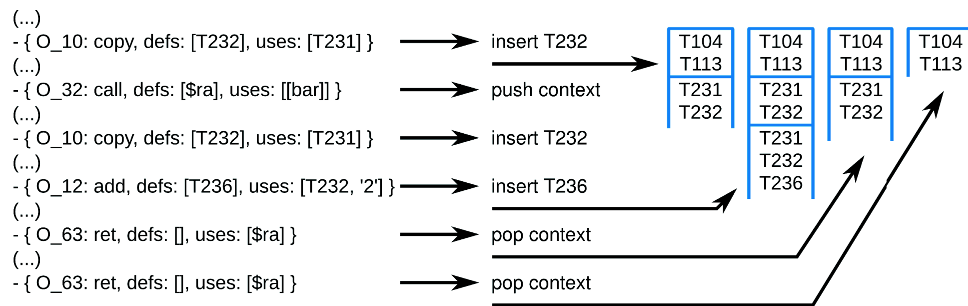


Figure 7.7: An example of recursive function interpretation trace and its context stack in different execution states.

Figure 7.7 shows a trace of instructions during interpretation of a recursive function call (reduced only to two instructions per call for readability) and the context stack in different execution states. After first instruction, variable `T232` is inserted to the current context, where already `T231` was previously inserted. When interpreting a call instruction, a new context is pushed on the context stack, on which variables will be inserted. At this point, SSA variables can have multiple storage spaces, one for each existing context. When return from function occurs, its context is popped from the context stack.

ϕ-functions In his Interpretable SSA (ISSA) von Ronne et al. [2004] intermediate representation, Von Ronne proposed to extend the instruction set with a "`pfe`" instruction marking the end of ϕ-functions in a block, and an auxiliary CFG-Edge Number (CEN) register. The CEN register is set when interpreting branching instructions to store the path taken by the program. The ϕ-functions are then interpreted one by one but storing the results in temporary values. Finally, when encountering the `pfe` instruction, results are copied into the correct variables. This solution allows for real direct imperative interpretation but requires extensions of the classical SSA form with a special instruction and register, which we believe is unnecessary.

In our case, instead of setting an auxiliary register, interpretation of a branch instruction simply checks if there are any ϕ-function in the target basic block. If so, they are interpreted and the results are also stored in temporary storage. Our SSA form only allows ϕ-functions to be placed at the beginning of a basic block (as it is usually the case), so when there is no more ϕ-function left, the values are copied to their correct interpreted registers.

7.3 The JIT Compiler

In previous section we have shown that Tirex is suitable for interpretation of the SSA form target-level program representation. However, execution only by interpretation

does not make the virtual execution environment efficient, especially on a VLIW processor where multiple instructions could be executed in parallel. In this section we describe the *proof-of-concept* JIT compiler.

7.3.1 Code generation

The compilation process of the Tirex representation is very simple compared to compilation of JVM or CLI bytecodes. It does not require lowering of calling conventions and laying out the data. The instruction selection process was also done by the front-end static compiler, thus the code generator has to perform only few tasks to emit a binary code, including:

- register allocation if in SSA form,
- gathering call sites of other Tirex functions,
- request memory space from the code cache,
- encode the function.

Call sites To avoid keeping track of all call sites and the need for patching the native code every time one of the called functions within is recompiled, all the calls are done via a special callback function. For this purpose, during the compilation of a function all the call instructions to Tirex functions are gathered, and the return address for each of them is computed. Each call target is set to the callback function. Gathered pairs of addresses and symbols related to the called functions are kept in a hash table as shown on the Figure 7.8 and used later to identify the callee from within the callback function.

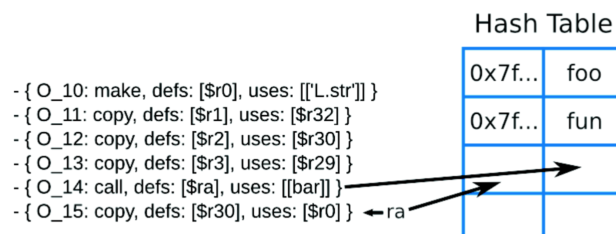


Figure 7.8: Gathering call sites during compilation.

Figure 7.9 shows how a call occurs from within a compiled code. The callback function is written in the target assembly to allow stack modification and registers manipulation without loosing the call context of the callee itself. At the beginning a new temporary stack frame is created where the call arguments are stored. Then the symbol of the callee is looked up in the hash table using the given return address. If

the callee was already compiled, a pointer to the native code is returned, otherwise the compilation is triggered.

After obtaining the pointer, call arguments previously stored on the stack frame are restored, the temporary stack frame is destroyed and finally the callee function is executed.

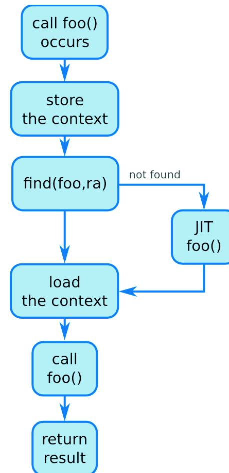


Figure 7.9: A function call from compiled code.

All the calls happen without giving back the control to the execution engine, hence simplifying the process and chaining together the functions in a similar fashion as bypassing the main translation loop in some virtual machines [Cmelik and Keppel, 1994; Lattner et al., 2002].

Instruction encoding The LAO relies on the MDS, which automatically generates encoding functions and bundle templates based on the architecture description. During the final, code generation phase, the LAO emits the binary code directly to the memory provided by the code cache, using the encoding functions.

7.3.2 Code cache

The code cache is an important part of a JIT environment. It is a memory space designated to keep the generated native code and to execute this code from within. As opposed to a static program, in a JIT environment the code is generated only for parts of a program. Moreover, it is not always kept in memory during the whole execution of that program because of the space constraints as well as a possibility of its recompilation with a different optimization scheme applied.

The possibility and in some cases the need for removing the code because of recompilation or running out of memory space leads to memory fragmentation, bad code locality and cash misses. These problems are a direct result of the code eviction

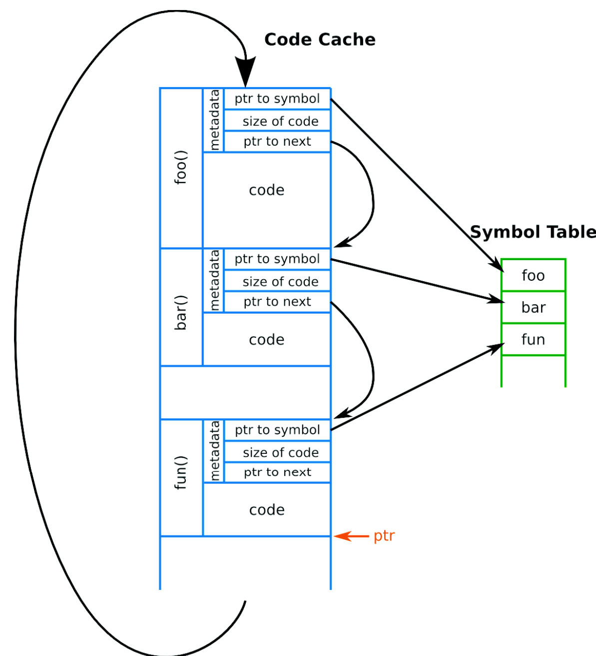


Figure 7.10: The Code Cache.

algorithm used for the cache management. When the code cache is full, in order to provide memory space for newly generated code a decision has to be made which parts of existing code should be removed. This decision is difficult as it is hard to predict when and how often the existing fragment of code will be executed. In addition to that difficulties, some parts of the code in the cache could be marked as unremovable due to its current execution.

In LAO we use a single, circular buffer-like scheme shown on Figure 7.10, which provides good performance with relatively cheap bookkeeping [Hazelwood and Smith, 2002]. In the future, when the profiling mechanisms and the interoperability with the interpreter are on place, it will be extend to three-level generational code cache as proposed by Hazelwood and Smith [2003], similar to shown on Figure 7.11.

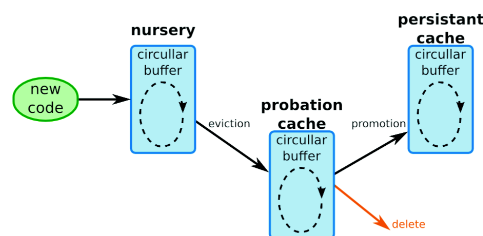


Figure 7.11: Three level, generational Code Cache.

Each entry in the code cache starts with a metadata which contains a pointer to

the symbol attached to this entry, size of entry, and a pointer to next entry in the code cache.

If the JIT compiler requires memory space to place the code, the code cache, starting from current pointer, is gathering entries of space greater or equal than required memory size plus the metadata. If one of the entries is marked as non-removable (i.e., being a function during execution), the process restarts from next entry. If the end of buffer is reached, process restarts from the beginning. When sufficient entries are found simply the metadata of the first one is overwritten to point to the next valid entry and skipping the ones that will be overwritten without removing them. All the values of symbols related to the entries being removed are set to NULL, to tell the call back function that these functions are not yet compiled.

7.4 Summary

By choosing a target-level intermediate representation, we give up program portability across processor architectures. However, this allows to keep the same processor calling conventions and data layout rules (ABI) for both interpreted and native code. As our implementation demonstrates, this choice dramatically simplifies the tasks that the virtual execution environment has to do before and after a native function call in order to provide compatibility of function arguments and results. In practice, mixed-mode execution is no longer a problem.

A dynamic compiler is a natural complement of a interpreter-based virtual machine. Although our JIT compiler is not yet finished, we have shown that Tirex is suitable for dynamic code generation and perfect for a mixed-mode execution with an interpreter and external native libraries.

We also show how the SSA form of Tirex once loaded in the virtual execution environment internal structures provides a unified program representation which is directly used for interpretation and JIT compilation. Keeping only one intermediate representation, in the SSA form, for both compilation and interpretation, assuming a SSA interpreter exists, simplifies the virtual machine and reduces the memory requirements compared to keeping both, non-SSA and SSA, at the same time; and removes the need for run-time SSA form construction.

In the following chapter we discuss future perspectives of the Tirex in static analysis and dynamic execution environment and we summarize the work presented in this thesis.

Chapter 8

Conclusion

In this thesis we have presented a new, target-level intermediate representation called Tirex, as well as its various applications. The work on Tirex, however, is not yet complete, but also creates new opportunities. In the following sections we discuss the future work and ideas that can be realized using Tirex and existing infrastructure, and summarize this thesis.

8.1 Future work

8.1.1 Intermediate representation

Although the current state of the intermediate representation allows for generating fully functional program, as well as passing additional information used by the optimizer and code generator, there is still some work left on important functionality, including passing debug information and encoding the IR in a binary form, as well as providing some more extensions.

Debug information One of the first reasons of creating Tirex was connection of some popular compilers with an advanced, target-specific optimizer. This step interferes with a regular compilation chain. It is already not trivial task to provide accurate debug information when aggressive optimizations are applied in a single compiler, thus tracing the faulty code back to the source code while providing additional level of optimizations outside the main compiler could be even more difficult. Nevertheless, in an industrial quality software development toolkit this functionality is required. In its current state, however, Tirex does not specify debug information, but we plan to extend the specification and implement it in the both Tirex code generator and the optimizer in the nearest future.

Binary form The current, textual form of the Tirex representation, although being excellent for experiments and testing parts of the compilation toolchain with the ability of its inspection and modification by hand, is not well suitable for virtual execution environment due to the overhead of text parsing and its size. Because of that we need to encode Tirex in a binary which was already considered during the design of the current parser. Also, this can speed up the static compilation process of bigger projects with the aggressive optimizations enabled by reducing the time of writing and reading of a text file.

Polyhedral optimizations Polyhedral representation of static control loops is a significantly more powerful representation of loop behavior and memory dependencies than the current loop scoped information with regards to program transformations, but it is restricted to special cases, the so-called "static control program" parts. As we plan to use optimizations based on the polyhedral model, again, we will need to find a way to efficiently encode this information in Tirex.

8.1.2 The runtime

The Tirex runtime is already partially in place, however in the current state being more *proof-of-concept* than production-grade, requires some cleanup and improvements focused on the interpreter and the dynamic compiler.

Instruction interpretation The core of the interpreter was build using the behavior functions generated automatically from the Machine Description System. It allowed to save lot of time while building a working interpreter, however not without price. These function were designed to be part of the instruction set simulator, thus to provide correct results of instructions on any architecture. They use extensively casting up for the internal computation purpose and casting back down while returning computed values, as well as bit manipulation and calling helper functions, making the interpretation process inefficient.

Tirex, however, is a target-level representation, and as such must be interpreted on a specific architecture. In such case it is more natural to interpret the instruction set with the underlying processor's instruction set. In the nearest future we will replace the behavior functions by automatically generated inline assembly that would execute correct instructions directly by the processor and improving the interpretation efficiency drastically.

Profiling The main reason of using interpreter is to avoid the startup overhead introduced by the JIT compiler and relatively easy prepare the ground for that compiler to work. We plan to implement mechanisms of gathering profile and dynamic information of a program to use them to drive the JIT compilation process and deciding

what and how to optimize. Thanks to a unified, target-level intermediate representation for both interpretation and JIT, the control-flow after Out-of-SSA in generated (not-optimized) binary will be similar to that in the interpreted code, thus the profile gathered by the interpreter will more accurately reflect the native code and hopefully be more useful for optimizations.

JIT compiler The current *proof-of-concept* JIT compiler does not do much more than compiling one function at a time when the threshold is exceeded in the interpreter mode and managing the code cache. Tirex with the memory dependencies annotated along with a program has the potential for aggressive runtime optimizations. The future work will focus mainly on enabling and experimenting with these optimizations and dynamic instruction scheduling on a VLIW processor.

Also, our JIT system is running on a massively parallel processor, thus exploring multicore code generation and execution with provided memory dependencies, as well as interpretation, code generation and optimizations in parallel, is a natural step forward in gaining performance.

There is also space left for improving the code cache in terms of efficient processor's cache and memory management especially challenging in a heavily parallel environment.

8.2 Perspectives

8.2.1 Execution traces and program analysis

Tirex with its runtime can be also used as a tool for program analysis, which could be useful not only for further optimizing of a given program, but also for finding problems in the compilers itself. In this section we present an idea of such a use case of Tirex proposed by Fabrice Rastello, that we plan to implement in the nearest future.

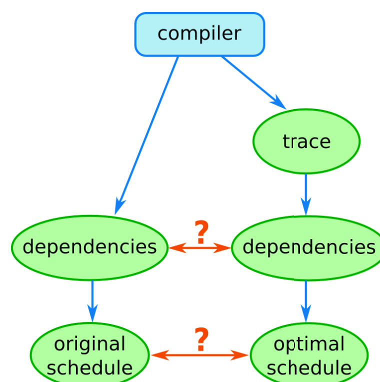


Figure 8.1: Dependencies verification and optimal schedule search.

The idea is based on a simple use of the Tirex interpreter for tracing memory loads and stores, and arithmetic instructions that are accessing registers, all of them discriminated by loops, hence directly profiting from the loop nesting forest kept in the Tirex form.

Each loop is equipped with a counter set to zero while passing by the entry node or exit edge, and incremented while passing by a back-edge. To avoid having to big traces, the number of iterations traced is limited to n. Also, to get a unique iteration vector for each line of trace, each instruction in the trace has a unique identifier.

Such a trace could be used, among others, for two purposes, as shown on Figure 8.1. One of them is to construct data dependencies that could be compared with dependencies computed by a static compiler. Further analysis of both dependencies can be useful for investigating why compiler decided not to perform some optimizations and potentially verify correctness of the computed dependencies. Finally, this knowledge can also be used for instruction scheduling in a dynamic execution environment, thus allowing to obtain more optimal schedule, than the one generated during static compilation.

Later, for the purpose of trace, Tirex could be equipped with more information than simple loop nesting forest, but also with data dependencies, aliasing and other if needed to perform more sophisticated tracing and program analysis.

The trace The trace will most likely be also represented using YAML, similarly to Tirex, profiting from its human-readable form and making it easy to write analyses using scripting languages like Perl, Python or Ruby.

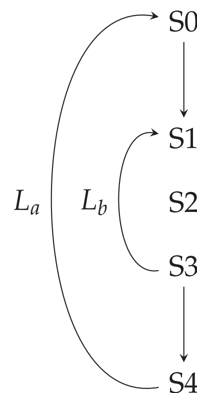


Figure 8.2: Simplified example of loop L_b nested inside loop L_a. $S0 - S4$ correspond to some statements inside the loops.

On Figure 8.2 we give a very simple example of two loops L_a and L_b. For the simplicity we omit the division to basic blocks leaving only statements $S0 - S4$ that correspond to some arithmetic or memory access instructions inside the loops.

Example of the trace gathered by the interpreter is given on Listing 8.1. The field `Index` is a iteration vector, where the numbers are accordingly L_a counter, L_b counter and statement identifier. `Statement` corresponds to an actual operation that could be specified directly or renamed in the header of the trace. `Def` and `Uses` contains given operation results and arguments. In the example `$A(), $B(), $C()` are arrays of memory, that should be specified in the header of trace. Instead an address could be used directly.

Listing 8.1: Example of trace memory and register accesses.

```
Trace:
  – { Index: "(0,0,0)", Statement: S0, Def: $r12 }
  – { Index: "(0,0,1)", Statement: S1, Uses: [@A(0)], Def: $r10 }
  – { Index: "(0,0,2)", Statement: S2, Uses: [@B(0)], Def: $r11 }
  – { Index: "(0,0,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(0,1,1)", Statement: S1, Uses: [@A(1000)], Def: $r10 }
  – { Index: "(0,1,2)", Statement: S2, Uses: [@B(1)], Def: $r11 }
  – { Index: "(0,1,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(0,2,1)", Statement: S1, Uses: [@A(2000)], Def: $r10 }
  – { Index: "(0,2,2)", Statement: S2, Uses: [@B(2)], Def: $r11 }
  – { Index: "(0,2,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(0,0,4)", Statement: S3, Uses: [$r12], Def: @C(0) }
  – { Index: "(1,0,0)", Statement: S0, Def: $r12 }
  – { Index: "(1,0,1)", Statement: S1, Uses: [@A(1)], Def: $r10 }
  – { Index: "(1,0,2)", Statement: S2, Uses: [@B(0)], Def: $r11 }
  – { Index: "(1,0,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(1,1,1)", Statement: S1, Uses: [@A(1001)], Def: $r10 }
  – { Index: "(1,1,2)", Statement: S2, Uses: [@B(1)], Def: $r11 }
  – { Index: "(1,1,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(1,2,1)", Statement: S1, Uses: [@A(2001)], Def: $r10 }
  – { Index: "(1,2,2)", Statement: S2, Uses: [@B(2)], Def: $r11 }
  – { Index: "(1,2,3)", Statement: S3, Uses: [$r12, $r10, $r11], Def: $r12 }
  – { Index: "(1,0,4)", Statement: S3, Uses: [$r12], Def: @C(1) }
```

8.2.2 WCET

Worst case execution time or *WCET* is an important program analysis used as an input for schedulability in real-time systems. WCET is a maximal time that a given task needs to complete the computation for any input data on a specific hardware.

Some of the main WCET timing computation techniques rely on the static analysis [Wilhelm et al., 2008]. As high-level languages lack the context and the target-processor details, they are inadequate to calculate the accurate time. Because of that, the analysis is done on the assembly to capture all effects of the given processor, however often with some annotations passed from the higher-level language [Li and Malik, 1995; Mok et al., 1989].

Generalized approach to WCET requires three steps:

control-flow analysis Also called *high-level analysis*, tries to identify possible paths of execution and bounding the loops.

processor-behavior analysis Or *low-level analysis*, determining the effects of the architecture on the execution time.

WCET computation Combining the two above to obtain the overall WCET.

One of the main approaches to such a WCET computation is technique called *IPET* (*Implicit Path Enumeration Technique*) proposed by Li and Malik [1995] and implemented in the Otawa [Ballabriga et al., 2010] project. In this method, each basic-block and program flow edge has given time coefficient corresponding to its upper bound execution time, and a counter corresponding to the number of times the basic-block is executed or flow edge taken. Then the sum of products of the execution times and counts is used to computed the WCET.

The IPET method, as well as other static methods require, as mentioned before, control-flow analysis and processor-behavior analysis. The tools like Otawa, operating on binaries, have to reconstruct the CFG, identify basic blocks and loops before starting the computation. Also, as the Otawa authors state, some information needed for the WCET computation cannot be automatically extracted from the binary, hence must be provided by the programmer as special annotations. This information is called *flow-facts* and in Otawa contains mainly loop bounds information specified in a special file.

We believe that Tirex could be successfully used in WCET computation process instead of the target binary and flow-fact file. First, it is target-level, hence suitable for *processor-behavior* analysis, secondly containing explicit program structure, including basic-blocks and loops, thus simplifying the CFG construction, and finally by allowing for passing additional information, which in that case would be loop bounds information as well as any other required for the purpose of WCET technique.

8.2.3 Native software simulation

Native software simulation [Yoo et al., 2003] comes from a very simple idea applied very often by embedded software developers: software compilation on the host machine instead of cross-compiling, in order to perform fast, functional validation. Thus, as opposed to classical *Instruction Set Simulators* (ISS) where each instruction is interpreted one by one, in native software simulation a program is compiled and executed on the host machine with some wrappers to connect event-driven simulation environment.

Pure native simulation, however, allows only for functional simulation, without any accurate target-specific information, that would allow for estimate timings. Petrot et al. [2011] discuss different approaches to simulation, including native simulation.

One of the approaches discussed by Pétrot and his colleagues, uses *binary translation* techniques. The process starts directly from the target binary code. The binary is

translated statically or during the simulation (*JIT simulation*) to the instruction-set of the underlying machine. Everything that cannot be provided by the host-platform is modeled in the simulation framework. However, to obtain accurate simulation results, the host-platform code has to be annotated with the target-specific details. The target binary has to be inspected in order to recreate the program structure, the CFG, and to extract information such as number of instructions per basic block, number of memory accesses, etc.

We believe, that Tirex can be used as an input for the binary translation method. It is already target-dependent and contains explicit program structure that allows for easy CFG construction. Also, the ability for passing additional information when needed can be useful for the simulation purpose.

8.3 Summary

We have presented Tirex, a new intermediate representation, characterized mainly by the following properties:

Human-readability The textual, YAML foundation makes it human-readable, thus useful for fast inspection and modification 'by hand'. This property makes the tools prototyping and testing processes, as well as program analysis very simple. YAML simplifies also the parsing process and allows for processing using many different scripting languages.

Target-level We have intentionally given up target-independence, to be able to move as much of work as possible to the frontend compiler. The backend compiler has to focus only on the target-level optimizations.

Explicit program structure Although target-level, Tirex is much more useful for further processing then assembler, thanks to the explicit program structure, including functions, loops and basic blocks, as well as uses and definitions of instructions and branch targets.

Static single assignment Tirex supports programs in SSA and mixed-forms, hence avoiding the need of SSA construction in the backend compiler.

Flexibility One of the design goals was to allow adding more information when needed, without the need of implementing its support in all the tools. We have presented an example of such extension, data dependencies. If a backend compiler or other tool does not know what to do with this information, it could simply ignore it.

Thanks to these properties, although created to simplify connection between mainstream, frontend compilers, with proprietary, specialized, target-specific backend op-

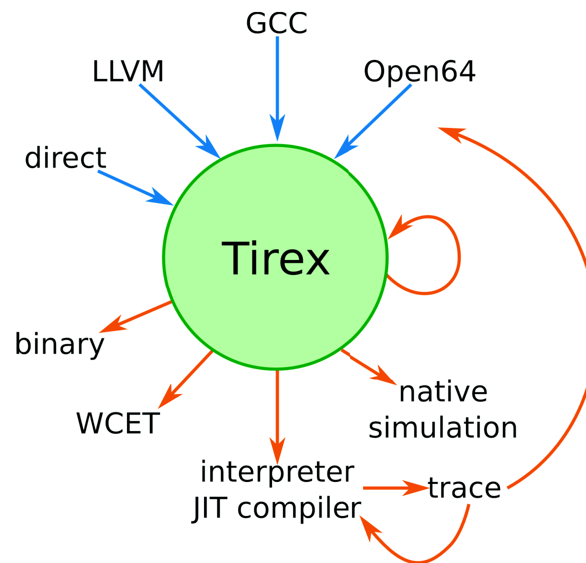


Figure 8.3: Tirex in many applications.

timizer, Tirex have proven to be useful also for writing backend tests 'by hand' or modifying compiler-generated files, by providing very clear and extensible structure.

Our interest in dynamic compilation techniques and the search for gaining more performance lead us also to use of Tirex in a virtual execution environment, consisting of a interpreter and JIT compiler. Tirex, being very low-level, moves the burden of lots of compilation phases that do not really profit from dynamic information, to the frontend compiler. Of course, this comes with the price of loosing target-independence in contrast to Java JVM or CLI, but as in our environment the frontend languages are mostly C and C++, that is languages allow for pointer arithmetics and direct operations on memory, the target-independence would be difficult or even impossible to achieve. In return, however, we gain the time to perform during runtime more aggressive, target-dependent optimizations that could further benefit from additional information included in the IR, such as data dependencies discriminated by loops.

We have proven that Tirex is suitable for both, interpretation (also in SSA) and JIT compilation, and what is even more important in case of mixed-mode execution, by using exactly the same stack and global data as well as the ABI for interpretation and native code, the process of calling native code from within the interpreter, and interpreted from native, does not require very special care from the VEE, hence making the switch between interpretation, JIT and the code from native libraries very cheap. Furthermore, the same IR in SSA form for both interpretation and JIT removes the need for IR conversion and makes the CFG in both cases very similar, thus making the profiling information very accurate.

With the additional information, such as data dependencies, and its ability for extending that information (e.g., aliasing) in combination with the interpreter, Tirex

opens lots of possibilities for tracing memory and register accesses and its further analysis potentially allowing for finding problems in frontend compilers or simply amend dynamic optimizations in the VEE. The explicit program structure, including loop nesting forest with other information could be used in static analysis, program execution analysis and in other tasks including WCET computation and native simulation, hence giving unlimited possibilities of program analysis on a very low level.

List of Figures

1.1 Static compilation 3
1.2 Dynamic compilation 4
1.3 Simplified JIT structure 6
1.4 Simplified dynamic optimizer structure 8

2.1 Instructions with the defined values and used parameters. 13
2.2 Example procedure, its control-flow graph and a dominator tree. 15
2.3 Example of C code and its representation in SSA. 17
2.4 Swapping of two variables in SSA form using ϕ-procedures. 18
2.5 Example of data dependencies between statements. 19
2.6 Example of data dependence graph. 19
2.7 An example of a loop tree for given CFG and its spanning tree. 21

4.1 GCC-CIL compilation and execution flow 34
4.2 CIL Code sizes normalized by CIL code size of GCC-CIL `-Os`. 47
4.3 Metadata sizes normalized by metadata size of GCC-CIL `-Os`. 48

5.1 The framework 49
5.2 MDS (Machine Description System) work-flow. 52
5.3 C to Tirex compilation using LLVM 55
5.4 Program processing in the LAO 58

6.1 Tirex in our toolchain. The MDS supplies target-specific files to build the upstream compilers and LAO code generator. The path from GCC is not yet functional. 61
6.2 Shortnames are created by using operand types to disambiguate instructions with the same mnemonic. 64
6.3 A global structure in C on the left and the corresponding object in Tirex on the right. The initialization is not unique, the `short` -1 uses two bytes to form `0xffff`, but could have been "`s16: -1`" or "`u16: 65535`." The "`space`" where added to satisfy alignment constraints of the pointer (4) and the double float (8). 71

7.1 Tirex for mixed mode execution and JIT compilation. 79
7.2 The Tirex virtual execution environment. 80
7.3 Execution engine initialization. 81
7.4 Example of a call in Tirex 84
7.5 Passing arguments before the call, saving stack pointer and return address and switching stack pointers 84
7.6 Copying the result to the interpreted registers after the call and restoring stack pointer and return address 85
7.7 An example of recursive function interpretation trace and its context stack in different execution states. 86
7.8 Gathering call sites during compilation. 87
7.9 A function call from compiled code. 88
7.10 The Code Cache. 89
7.11 Three level, generational Code Cache. 89

8.1 Dependencies verification and optimal schedule search. 93
8.2 Simplified example of loop L_b nested inside loop L_a. $S0 - S4$ correspond to some statements inside the loops. 94
8.3 Tirex in many applications. 98

Listings

6.1 Example of a MinIR program from the MinIR project 63
6.2 Dummy example of a Tirex program 65
6.3 Example of section definitions. 69
6.4 Example of a nested loop in C. 72
6.5 Loop scoped dependences example. 72
8.1 Example of trace memory and register accesses. 95

Bibliography

Ali-Reza Adl-Tabatabai, Michał Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. *SIGPLAN Not.*, 33:280–290, May 1998. ISSN 0362-1340.

Ole Agesen and David Detlefs. Mixed-mode bytecode execution. Technical report, Mountain View, CA, USA, 2000.

B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561. URL http://doi.acm.org.gate6.inist.fr/10.1145/73560.73561.

Joel Auslander, Matthai Philipose, Craig Chambers, Susan J. Eggers, and Brian N. Bershad. Fast, effective dynamic compilation. *SIGPLAN Not.*, 31:149–159, May 1996. ISSN 0362-1340.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35:1–12, May 2000. ISSN 0362-1340.

Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In *SEUS*, pages 35–46, 2010.

Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML 1.2 Specification, 2009. http://www.yaml.org/spec/.

Benoit Boissinot. *Towards an SSA-based Compiler Back-end: Some Interesting Properties of SSA and Its Extensions*. PhD thesis, École Normale Supérieure de Lyon, 2010.

Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont De Dinechin, and Fabrice Rastello. Fast Liveness Checking for SSA-Form Programs. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code Generation and Optimization*, pages 35–44, New York, NY, USA, 2008. ACM.

Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoît Dupont De Dinechin, and Christophe Guillon. Revisiting Out-of-SSA Translation for Correctness, Code Quality and Efficiency. In *CGO '09: Proceedings of the 2009 international symposium on Code Generation and Optimization*, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

Florent Bouchez. *A Study of Spilling and Coalescing in Register Allocation as Two Separate Phases*. PhD thesis, École Normale Supérieure de Lyon, 2009.

Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28:859–881, July 1998. ISSN 0038-0644. doi: 10.1002/(SICI)1097-024X(19980710)28:8<859::AID-SPE188>3.0.CO;2-8. URL http://dl.acm.org/citation.cfm?id=295545.295551.

Christian Bruel. If-Conversion SSA Framework for partially predicated VLIW architectures. In *ODES 4*, pages 5–13, March 2006.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000. doi: 10.1177/109434200001400404. URL http://hpc.sagepub.com/content/14/4/317.abstract.

Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5.

Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.

Michal Cierniak and Wei Li. Briki: an optimizing java compiler. *Computer Conference, IEEE International*, 0:179, 1997. ISSN 1063-6390.

Bob Cmelik and David Keppel. Shade: a fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM. ISBN 0-89791-659-X. doi: http://doi.acm.org/10.1145/183018.183032. URL http://doi.acm.org/10.1145/183018.183032.

Robert S. Cohn, David W. Goodwin, and P. Geoffrey Lowney. Optimizing alpha executables on windows nt with spike. *Digital Technical Journal*, 9:3–20, 1997.

Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 145–156, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

Marco Cornero, Roberto Costa, Ricardo Fernandez Pascual, Andrea Ornstein, and Erven Rohou. An Experimental Environment Validating the Suitability of CLI as an Effective Deployment Format for Embedded Systems. In *HiPEAC International Conference*, 2008.

Roberto Costa and Erven Rohou. Comparing the Size of .NET Applications with Native Code. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–104, New York, NY, USA, 2005. ACM.

Roberto Costa, Andrea Ornstein, and Erven Rohou. CLI Back-End in GCC. In *Proceedings of the GCC Developers' Summit*, 2007.

Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *IEEE Micro*, 17(3):36–43, May 1997. ISSN 0272-1732. doi: 10.1109/40.591653. URL http://dx.doi.org/10.1109/40.591653.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. ISSN 0164-0925.

François de Ferrière. Improvements to the psi-ssa representation. In *Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, SCOPES '07, pages 111–121, New York, NY, USA, 2007. ACM.

Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher. Deli: a new run-time control point. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 35, pages 257–268, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 0-7695-1859-1.

L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800542. URL http://doi.acm.org/10.1145/800017.800542.

Benoît Dupont De Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2), 2004.

Benoît Dupont De Dinechin. Time-IndexedFormulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In *3rd Multidisciplinary International Scheduling conference: Theory and Applications (MISTA)*, 2007.

Benoît Dupont De Dinechin. Inter-block Scoreboard Scheduling in a JIT Compiler for VLIW Processors. In *Euro-Par*, pages 370–381, 2008.

Benoît Dupont De Dinechin, François de Ferrière, Christophe Guillon, and Artour Stoutchinin. Code Generator Optimizations for the ST120 DSP-MCU Core. In *CASES'00: Proceedings of the 2000 international conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 93–102, New York, NY, USA, 2000. ACM.

ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. 4 edition, 2006. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm.

Chris Fraser and David Hanson. lcc, A Retargetable Compiler for ANSI C. http://www.cs.princeton.edu/software/lcc/.

Andreas Gal, Christian W. Probst, and Michael Franz. Structural encoding of static single assignment form. *Electron. Notes Theor. Comput. Sci.*, 141(2):85–102, 2005. ISSN 1571-0661.

Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996. ISBN 0201634511.

K. John Gough. *Compiling for the .Net Common Language Runtime*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130622966.

Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. An evaluation of staged run-time optimizations in dyc. *SIGPLAN Not.*, 34:293–304, May 1999. ISSN 0362-1340.

David R. Hanson. lcc.NET: targeting the .NET Common Intermediate Language from Standard C. *Software Practice and Experience*, 34(3):265–286, 2004.

Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4), 1997.

Kim Hazelwood and Michael D. Smith. Code cache management schemes for dynamic optimizers. *Interaction between Compilers and Computer Architecture, Annual Workshop on*, 0:102, 2002. doi: http://doi.ieeecomputersociety.org/10.1109/INTERA.2002.995847.

Kim Hazelwood and Michael D. Smith. Generational cache management of code traces in dynamic optimization systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 169–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL http://dl.acm.org/citation.cfm?id=956417.956551.

Urs Hölzle. Adaptive optimization for self: Reconciling high performance with exploratory programming. Technical report, Mountain View, CA, USA, 1995.

Urs Hölzle and David Ungar. A third-generation self implementation: reconciling responsiveness with performance. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 229–243, New York, NY, USA, 1994. ACM. ISBN 0-89791-688-3. doi: http://doi.acm.org/10.1145/191080.191116. URL http://doi.acm.org/10.1145/191080.191116.

Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. Vmad: An advanced dynamic program analysis and instrumentation framework. In Michael O'Boyle, editor, *Compiler Construction*, volume 7210 of *Lecture Notes in Computer Science*, pages 220–239. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-28651-3.

Chandra Krintz. Improving mobile program performance through the use of a hybrid intermediate representation. In *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, PPPJ '02/IRE '02, pages 175–180, Maynooth, County Kildare, Ireland, Ireland, 2002. National University of Ireland. ISBN 0 901519 87 1. URL http://dl.acm.org/citation.cfm?id=638476.638511.

Chris Lattner, Misha Brukman, and Brian Gaeke. Jello: a retargetable just-in-time compiler for llvm bytecode, 2002.

Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *ISPASS*, pages 175–183. IEEE Computer Society, 2010. ISBN 978-1-4244-6022-9. URL http://dblp.uni-trier.de/db/conf/ispass/ispass2010.html#LaurenzanoTCS10.

Julien Le Guen, Christophe Guillon, and Fabrice Rastello. Minir, a minimalistic intermediate representation. In Florent Bouchez, Sebastian Hack, and Eelco Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 5–12, 2011.

Mark Leone and R. Kent Dybvig. Dynamo: A staged compiler architecture for dynamic program optimization, 1997.

Allen Leung and Lal George. Static single assignment form for machine code. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 204–214, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5.

Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, LCTES '95, pages 88–98, New York, NY, USA, 1995. ACM. doi: 10.1145/216636.216666. URL http://doi.acm.org/10.1145/216636.216666.

Serge Lidin. *Expert .NET 2.0 IL Assembler*. Apress, 2006.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065034. URL http://doi.acm.org/10.1145/1064978.1065034.

Microsoft. Unix custom application migration guide, 2006. URL http://technet.microsoft.com/en-us/library/bb496996.aspx.

Sun Microsystems. The java hotspot virtual machine, 2001. URL http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html.

A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight execution time bounds of programs by annotations. *IEEE Real-Time Syst. Newsl.*, 5(2-3):81–86, May 1989. URL http://dl.acm.org/citation.cfm?id=87662.87681.

Mono. Linear intermediate language, 2012. URL http://www.mono-project.com/Linear_IL.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007. ISSN 0362-1340. doi: 10.1145/1273442.1250746. URL http://doi.acm.org/10.1145/1273442.1250746.

Oracle. The java hotspot performance engine architecture, 2010. URL http://java.sun.com/products/hotspot/whitepaper.html.

Frederic Petrot, Nicolas Fournel, Patrice Gerin, Marius Gligor, Mian-Muhammed Hamayun, and Hao Shen. On mpsoc software execution at the transaction level.

IEEE Design & Test of Computers, 28:32–43, 2011. ISSN 0740-7475. doi: http://doi.ieeecomputersociety.org/10.1109/MDT.2010.118.

Artur Pietrek, Florent Bouchez, and Benoît Dupont De Dinechin. Tirex: A target-level intermediate representation for compiler exchange. In Florent Bouchez, Sebastian Hack, and Eelco Visser, editors, *Proceedings of the Workshop on Intermediate Representations*, pages 13–20, 2011.

G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5), 2002.

Fabrice Rastello, François de Ferrière, and Christophe Guillon. Optimizing Translation Out of SSA Using Renaming Constraints. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, pages 265–278, 2004.

Bob Ramakrishna Rau. Levels of representation of programs and the architecture of universal host machines. In *MICRO 11: Proceedings of the 11th annual workshop on Microprogramming*, pages 67–79, Piscataway, NJ, USA, 1978. IEEE Press.

B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562. URL http://doi.acm.org.gate6.inist.fr/10.1145/73560.73562.

Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *In IEEE Workshop on Binary Translation*, 2001.

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4:2:1–2:36, January 2008. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1328195.1328197. URL http://doi.acm.org/10.1145/1328195.1328197.

Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18:649–658, November 1996. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/236114.236115. URL http://doi.acm.org/10.1145/236114.236115.

Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating Out of Static Single Assignment Form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 29(6):196–205, June 1994. ISSN 0362-1340. doi: 10.1145/773473.178260. URL http://doi.acm.org/10.1145/773473.178260.

Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimization at link-time, 1992.

Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical report, 1993.

Artour Stoutchinin and François de Ferrière. Efficient Static Single Assignment Form for Predication. In *MICRO: Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 172–181, 2001.

Artour Stoutchinin and Guang Gao. If-conversion in ssa form. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 336–345. Springer Berlin / Heidelberg, 2004.

T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Syst. J.*, 39(1):175–193, January 2000. ISSN 0018-8670. doi: 10.1147/sj.391.0175. URL http://dx.doi.org/10.1147/sj.391.0175.

Gabriele Svelto, Andrea Ornstein, and Erven Rohou. A Stack-Based Internal Representation for GCC. In *International Workshop on GCC Research Opportunities (GROW)*, 2009.

David Ungar and Randall B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA ’87, pages 227–242, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0. doi: 10.1145/38765.38828. URL http://doi.acm.org/10.1145/38765.38828.

Kapil Vaswani and Y. N. Srikant. Dynamic recompilation and profile-guided optimisations for a .net jit compiler. *IEE Proceedings - Software*, 150(5):296–302, 2003.

Jeffery von Ronne, Ning Wang, and Michael Franz. Interpreting programs in static single assignment form. In *Proceedings of the 2004 workshop on Interpreters, virtual machines and emulators*, IVME ’04, pages 23–30, New York, NY, USA, 2004. ACM. ISBN 1-58113-909-8. doi: http://doi.acm.org/10.1145/1059579.1059585. URL http://doi.acm.org/10.1145/1059579.1059585.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem–overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389. URL http://doi.acm.org/10.1145/1347375.1347389.

Am Wolfram, Niall Dalton, Jeffery von Ronne, and Michael Franz. Safetsa: a type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 137–147, New York, NY, USA, 2001. ACM. ISBN 1-58113-414-2. doi: http://doi.acm.org/10.1145/378795.378825. URL http://doi.acm.org/10.1145/378795.378825.

Sungjoo Yoo, Iuliana Bacivarov, Aimen Bouchhima, Yanick Paviot, and Ahmed A. Jerraya. Building fast and accurate sw simulation models based on hardware abstraction layer and simulation environment abstraction layer. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10550, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2.