BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ARCHITECTURE INFORMATION FOR LLVM COMPILER OPTIMIZATIONS

INFORMACE O ARCHITEKTUŘE PRO OPTIMALIZACE V PŘEKLADAČI LLVM

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. JAN SVOBODA

SUPERVISOR
VEDOUCÍ PRÁCE

Prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Master's Thesis Specification

22492

Student:	**Svoboda Jan, Bc.**
Programme:	Information Technology Field of study: Information Systems
Title:	**Architecture Information for LLVM Compiler Optimizations**
Category:	Compiler Construction

Assignment:

1. Get familiar with Codasip Studio, the LLVM compiler infrastructure and the CodAL language.
2. Get familiar with optimizations on intermediate representation driven by architecture information in the LLVM infrastructure.
3. Propose a mechanism for providing CodAL architecture information to LLVM.
4. Implement the proposed solution in the Codasip LLVM compiler.
5. Test the solution on a set of testing applications.
6. Evaluate the impact of the solution on performance and code size of testing applications.

Recommended literature:

- Bik, A. J. C.: *The Software Vectorization Handbook: Applying Multimedia Extensions for Maximum Performance*. Intel Press, 2004, ISBN: 9780974364926.
- Muchnick, S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 9781558603202.

Requirements for the semestral defence:

- Items 1 - 3 of this specification.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Hruška Tomáš, prof. Ing., CSc.**
Consultant:	Šnobl Pavel, Ing., CODASIP
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2019
Submission deadline:	May 20, 2020
Approval date:	October 29, 2019

Abstract

This thesis deals with the automatic extraction of processor architecture information from the CodAL language. Extracted information is used as the base for a cost model of the optimizer in the LLVM compiler. In this thesis, a new system was implemented, that creates the cost model, transforms it into a C++ code and compiles it into a dynamic library. This library is loaded at run-time by the compiler and used for better decision-making during the optimization process. The system achieves an average reduction in program code size of 14% and up to 68% improvement in the performance of the generated code.

Abstrakt

Tato práce se zabývá automatickou extrakcí informací o architektuře procesoru z jazyka CodAL. Získané informace jsou využity jako základ pro cenový model optimalizátoru překladače LLVM. V rámci práce vznikl nový systém, který vytváří cenový model, převádí jej do C++ kódu a sestavuje do dynamické knihovny. Tato knihovna je za běhu načtena překladačem a využita pro přesnější rozhodování o přínosech jednotlivých optimalizací. Výsledkem práce je průměrné 14% snížení velikosti strojového kódu programů a až 68% zlepšení výkonu generovaného kódu.

Keywords

compiler, optimization, cost model, processor, architecture, Codasip, CodAL, LLVM

Klíčová slova

překladač, optimalizace, cenový model, procesor, architektura, Codasip, CodAL, LLVM

Reference

SVOBODA, Jan. *Architecture Information for LLVM Compiler Optimizations.* Brno, 2020. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Prof. Ing. Tomáš Hruška, CSc.

Rozšířený abstrakt

Tato práce se zabývá poskytováním informací o procesorové architektuře překladači LLVM. Překladač převádí zdrojový kód do vlastní reprezentace, nad kterou provádí optimalizace, jež mají za cíl zefektivnit výsledný strojový kód z hlediska výkonu či velikosti. Některé optimalizace lze aplikovat bez ohledu na cílovou platformu, pro kterou je kód určený. Většina optimalizací ale zpravidla benefituje z detailní znalosti cílového procesoru. Cílem této práce bylo vytvoření automatizovaného systému pro dodávání potřebných informací o procesoru do překladače jazyků C a C++ založeného na LLVM infrastruktuře.

Teoretická část práce se zabývá úvodem do typického procesu návrhu procesorů od prvotní analýzy požadavků až po verifikaci syntetizovaného hardware. Čtenáři je představen způsob automatizace této nákladné činnosti pomocí jazyků pro popis architektury. Následně je popsáno prostředí Codasip Studio pro vývoj procesorů spolu s ukázkami modelu procesoru zapsaného v jazyce CodAL. Teoretickou část uzavírá přehled překladačové infrastruktury LLVM a její integrace do nástroje Codasip Studio. Tento nástroj do překladače LLVM automatizovaným způsobem přidává podporu pro modelovaný procesor.

V rámci praktické části diplomové práce byl analyzován optimalizátor v LLVM a jeho interakce s cenovým modelem cílového procesoru. Výsledkem analýzy je zjištění, že informace o architektuře jsou nejlépe využité při vektorizaci smyček a lineárního kódu. Také byly identifikovány nejčastěji využívané informace o architektuře: počet a šířka registrů, velikost instrukcí, cena operací a preference pro optimalizace inlining, unrolling, interleaving.

Na základě poznatků o chování optimalizátoru byl vytvořen nový nástroj integrovaný do prostředí Codasip Studio, jež analyzuje model procesoru, implementované instrukce a extrahuje užitečné informace. Z těchto informací o procesoru nástroj sestavuje cenový model a serializuje jej do C++ kódu použitelného v LLVM. Vygenerovaný kód je sestaven do dynamické knihovny obsahující kompletní cenový model. Do překladače jazyků C a C++ distribuovaného spolu s nástrojem Codasip Studio byla přidána funkcionalita tuto knihovnu za běhu načíst a využít její obsah pro zlepšení kvality optimalizací.

Nový systém byl otestovaný na dvou procesorech firmy Codasip: uRISC a Codix Berkelium. Pro testování byla využita sada reprezentativních programů i standardních syntetických benchmarků. V případě procesoru uRISC s podporou SIMD operací došlo ke zmenšení výsledných programů v průměru o 14 %, v jednom programu ke zrychlení o 68 % a v dalším ke 36% zpomalení. S procesorem Berkelium postaveném na instrukční sadě RISC-V bylo dosaženo průměrného 8% zmenšení napříč základními programy a 12% zrychlení programu porovnávajícího řetězce.

Výsledkem diplomové práce je tedy systém, který má pozitivní dopad především na velikost programů, což je jedna z klíčových metrik při vývoji vestavěných systémů. Implementovaný systém má praktické využití a bude součástí další hlavní verze nástroje Codasip Studio pro vývoj procesorů.

Architecture Information for LLVM Compiler Optimizations

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Prof. Ing. Tomáš Hruška, CSc. Supplementary information was provided by Ing. Pavel Šnobl. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Jan Svoboda
June 9, 2020

Acknowledgements

I would like to thank Ing. Pavel Šnobl and his compiler team at Codasip for their advice on the implementation, and Ing. Jiří Hynek, Ph.D. for his help with this text.

Contents

Chapter 1

Introduction

During the last decade, the annual growth of CPU performance has slowed down to around 3.5% – a sharp decline from the yearly 52% improvements we have seen since the 1980s until the early 2000s [9]. To overcome the stagnation, various companies have started to develop application-specific processors tailored to perform well when running a particular type of workload. For example, Google recently developed a tensor processing unit (TPU) to accelerate machine learning workflows [8].

The manual development of such processors is labour-intensive, which gave rise to electronic design automation (EDA) tools such as Codasip Studio. Codasip Studio is an integrated development environment (IDE) for modelling custom application-specific processors through the CodAL language. The environment can automatically generate a synthesizable hardware description of the processor, and development tools (i.e. verification testbench, simulator, assembler, C/C++ compiler and more).

To take advantage of new capabilities offered by application-specific processors, compilers of high-level languages need to adapt. Compilers need to have an intimate knowledge of the target architecture specifics to choose suitable optimizations and generate well-performing machine code. Currently, the optimizer in the C/C++ compiler generated by Codasip Studio is not provided with all the necessary architecture information to make informed decisions while optimizing code.

This thesis aims to design and implement a mechanism for providing architecture information to the LLVM optimizer used in the Codasip compiler. The information should be deduced from the CodAL processor model and manually editable by the processor engineers.

Chapter 2 contains an overview of the general processor design workflow. Then, Chapter 3 talks about how Codasip Studio and the CodAL language help to speed up the process. Chapter 4 is dedicated to an introduction of the LLVM compiler infrastructure and its usage within Codasip Studio. Chapter 5 proposes a mechanism for integrating the necessary architecture information into the optimizer used by the Codasip compiler. In Chapter 6, the implementation details of the integration are described. Next, Chapter 7 focuses on using the new infrastructure to improve the quality of code generated for Codasip processors. Finally, Chapter 8 sums up the key takeaways of this thesis and proposes steps that can be taken to improve the optimizer further.

Chapter 2

Processor Design

Nowadays, processors can be found in many kinds of systems scattered throughout our environment – in cities (cameras, traffic control), homes (appliances, security systems), offices (printers, networks), vehicles (driving assistance, navigation) or personal devices (smartphones, computers). Embedded systems play a critical role in our society. As the number of embedded systems and our dependence on them increases, the demands on the processors powering them grow as well.

Companies building embedded systems typically reach for an existing processor model offered by an established vendor (e.g. ARM). However, there are cases where the requirements on power, performance, or chip area are so specific they cannot be satisfied by an off-the-shelf solution. In situations like these, a custom-made processor designed from scratch might be the only way to build the system.

This chapter presents a brief overview of a typical processor design workflow to provide the context necessary for understanding Chapter 3 that introduces the CodAL language for describing processors and the Codasip Studio IDE.

2.1 Manual Process

In this section, we outline the manual approach to processor design. This description should act as a motivational piece that explains the desire for more automation of the workflow and foreshadows the benefits of development tools such as Codasip Studio.

2.1.1 Requirements

The design process typically starts by capturing the functional and non-functional requirements for the processor [19]. The functional requirements can be determined by finding a representative set of algorithms that are eventually going to be executed by the processor, and analyzing the instructions necessary for supporting an effective execution of such algorithms (e.g. through profiling on existing platforms). Additional functional requirements on frequency, throughput or latency may be imposed by the operating system, memory subsystem, the input/output devices attached to the processor and other aspects of the environment.

The processor design is usually further constrained by non-functional requirements such as power consumption, manufacturing cost, or electromagnetic compatibility (EMC). Another aspect to be considered is extensibility – for example even application-specific processors may benefit from general-purpose characteristics to allow for future modifications.

2.1.2 Instruction Encoding & Architecture

After finding the required operations, data types and addressing modes, designers create the instruction encoding scheme (i.e. the binary representation of instructions) [16]. There are different trade-offs to be considered when designing the encoding: maximal utilization of instruction-level parallelism can increase the performance but requires longer instructions to encode all of the operations, whereas short instructions may lead to smaller program size but restrict the possible combinations of resources. Variable instruction length allows for both short memory-efficient instructions and longer parallelism-enabling instruction but also complicates the logic of decoding and consequently, greater chip area.

Instruction encoding has a significant effect on the architecture organization, as operand access and operations require a specific type and number of registers, data transfer mechanisms, control structures and arithmetic/logic units. The initial architecture is usually dictated by the instruction set itself, and what follows is an exploration of alternative variations, typically guided by a performance analysis (e.g. calculating cycle counts and code size of the predefined set of algorithms). This iterative process is repeated until a solution without any critical bottlenecks reaching the desired cycle time is found.

2.1.3 Hardware & Verification

When the final architecture is decided, the processor can be implemented in a hardware description language (HDL) such as Verilog or VHDL. HDLs model the behavior and structure of hardware, which can be synthesized into a circuit and simulated [17]. Listing 2.1 shows an example of a simple hardware block described in VHDL.

```vhdl
entity mux is port(
  in1  : in  std_logic_vector(2 downto 0);
  in2  : in  std_logic_vector(2 downto 0);
  in3  : in  std_logic_vector(2 downto 0);
  in4  : in  std_logic_vector(2 downto 0);
  addr : in  std_logic_vector(1 downto 0);
  res  : out std_logic_vector(2 downto 0));
end mux;

architecture rtl of mux4 is begin
  res <= in1 when (sel = "00") else
         in2 when (sel = "01") else
         in3 when (sel = "10") else
         in4;
end rtl;
```

Listing 2.1: A multiplexer implemented in the VHDL language.

Since processors are often being deployed in safety-critical applications, the correctness of the implementation must be thoroughly verified. Verification efforts can make up to 80% of the total design cost, and there are three general approaches being employed today [10]:

1. Functional verification via simulation is the most commonly used verification technique that simulates the processor in software for some amount of clock cycles. Given an input program, the simulation produces a result which is then compared with the result produced by the golden model (an implementation considered to be correct).

The simulation is usually quick but not very thorough, as it only checks a single scenario in one run. The testing scenarios can be generated by hand, randomly or using techniques such as *random pattern simulation*, which allows to specify patterns that ought to appear more frequently in the randomly generated programs [1].

2. Formal verification encapsulates techniques that mathematically prove the functional correctness of a design. These are the most reliable verification methods, but also the most computationally intensive. The typical example of this category is *model checking*, which explores all reachable states of the processor and checks that a desired property always holds (e.g. an output signal is only set when a register contains a specific value).

3. Semi-formal is the intersection of simulation and formal techniques. A good example of the semi-formal approach is *symbolic simulation*, where some parts of the testing input are concrete values while others are variables. This makes one simulation run representative of many runs of pure simulation.

2.1.4 Software Tools

Besides the verified model of the processor hardware, a set of software tools is necessary for the actual application development. The instruction set simulator was already mentioned in previous sections, but other tools such as assembler, disassembler, linker and compiler of a high-level language (e.g. C/C++) must also be developed.

2.2 Architecture Description Languages

Creating a new processor architecture is a labour-intensive process that requires cooperation between the teams responsible for specification, hardware design, verification, simulation and software support. With strict time-to-market requirements, the opportunity to explore many architecture alternatives is limited, often leading to sub-optimal solutions. Moreover, the manual nature of the work is error-prone and can cause issues such as subtle inconsistencies between the hardware and software representations.

To overcome the drawbacks of the manual approach to processor design, *architecture description languages* (ADLs) were developed [16]. Such languages allow designers to specify some aspects of the processor like its instruction set or the architectural details, and then automatically generate the HDL code, verification environment, compiler, assembler or other software tools. This eliminates the need for cooperation of multiple teams, removes the possibility of inconsistencies between different processor representations and allows for faster and easier exploration of alternative designs.

There are two main categories of ADLs based on the aspect of processor architecture they capture: *structural* and *architectural* [2]. The structural languages such as MIMOLA focus on the components in the architecture and the connectivity between them, which makes them suitable for RTL generation and validation. On the other side, behavioral languages such as ISDL deal with the behavior of the processors' instruction set, and are geared towards simulation and compilation.

Languages that combine both of the above are referred to as *mixed* ADLs [15]. They capture both behavioral and structural aspects of the processor and can be used to support the use cases of both. Examples of such languages are EXPRESSION, LISA or CodAL. The next Chapter 3 contains a brief introduction to the CodAL language.

Chapter 3

Codasip Studio

Codasip Studio is an *integrated development environment* (IDE) that covers all aspects of *application-specific instruction set processor* (ASIP) design. Based on the processor model, it can generate a *register transfer level* (RTL) description of the architecture, an environment for its functional verification, and a *software development kit* (SDK). Information in Section 3.1 is taken from Codasip user manuals [4] and [6], while Section 3.2 draws from the Codasip SDK guide [7].

3.1 CodAL Language

CodAL is a description language syntactically similar to C, that is used in Codasip Studio for designing processors and expressing their properties:

- architectural resources: program counter and registers,
- instruction set: the names of instructions, operands and their binary coding,
- semantics: the description of how each instruction affects architectural resources,
- implementation: the micro-architectural details such as timing or resources.

The processor model is divided into two sub-models. The *instruction-accurate* (IA) model describes the architectural properties: resources, instruction set and semantics, whereas the *cycle-accurate* (CA) model describes the implementation details of a particular micro-architecture. Since the SDK is currently being generated from the IA model alone, the following sections focus on its essential elements while omitting the CA model altogether.

3.1.1 Elements & Sets

Both architectural resources and the instruction set are described through CodAL's `element` and `set` constructs. Elements describe instructions and their operands, while sets group multiple elements under one name. Each element consists of several sections and each of them defines one property of the particular instruction set component:

- `use` section declares instances of other elements referenced in the following sections,
- `assembly` section specifies the representation in textual assembly code,

- `binary` section defines the element's binary representation,
- `semantics` section specifies the instruction semantics via C code,
- `return` section contains the value that will represent the element after its instantiation in a `use` section.

3.1.2 Architectural Resources

The architectural resources that can be described in a CodAL IA model include registers, register files, address spaces, ports and interfaces. This section briefly shows and explains the example definitions of the above.

Interfaces

Interfaces are used for connections between memory and bus. Listing 3.1 below is an example of an interface to memory containing the program code.

```
interface if_fetch {
  bits = { 32, 32, 8 };
  type = AHB3_LITE:MASTER;
  flag = R;
  endianness = BIG;
  alignment = {
    address = 32;
    data = { 32 };
  };
};
```

Listing 3.1: Example of an interface specified in the CodAL language.

The `bits` attribute specifies the address bus width in bits, word width, and size of the least addressable unit—byte. The `type` field specifies the interface protocol; `flag` marks the interface read-only, and `endianness` can be either `BIG` or `LITTLE`. The `alignment` attribute then specifies the alignment of addresses and the data itself.

Address Space

The address space is an abstract model of the memory available to a processor. Most of today's processors have only one address space, but some architectures have separate address spaces for program code and data (e.g. the family of Harvard architectures). CodAL supports multiple address spaces. The example in Listing 3.2 below shows an example of an address space of a Von Neumann architecture that unifies the program code and data.

```
address_space as_all {
  bits = { 32, 32, 8 };
  interfaces = { PROGRAM : if_fetch, DATA : if_ldst };
  type = ALL;
  endianness = BIG;
};
```

Listing 3.2: CodAL definition of an address space of a Von Neumann architecture.

The bits and endianness attributes have the same semantics as in interface definitions above. The interfaces attribute specifies which interfaces are to be used for accessing the program code and data. The type field then again specifies if the address space can be used for accessing both parts of the memory (value ALL), program code only (PROGRAM), or data only (DATA).

Register Files

Listing 3.3 shows the definition of a register file named rf_gpr. It is an *architectural* register file (i.e. guaranteed to be implemented in hardware and visible to the compiler), contains four 32-bit registers, and has two read data-ports and single write data-ports.

```
arch register_file bit[32] rf_gpr {
  size = 4;
  dataport r0, r1 { flag = R; };
  dataport w0 { flag = W; };
};

// gpr_0, gpr_1, gpr_2 ...

element gpr_3 {
  assembly { "r3" };
  binary { 3:bit[2] };
  return { 3 };
};

set gpr_all : register_class(rf_gpr);
set gpr_all = gpr_0, gpr_1, gpr_2, gpr_3;
```

Listing 3.3: CodAL description of an architectural register file named rf_gpr.

The definition of one register belonging to the register file is shown: it consists of the textual assembly representation, two-bit binary representation and a return value. All four registers are then grouped into a single set named gpr_all that can be used, for example, as operand of an instruction.

3.1.3 Instruction Set

Instruction set definition builds on the principles introduced in the preceding sections. Listing 3.4 shows an example of an operation code for a comparison instruction: it specifies the opcode type with two members, creates an element for each of them and joins them both in a set called opc_cmp, which can be used by an instruction.

Finally, the comparison instruction i_cmp is declared in Listing 3.5. The instruction works on four operands: the operation code opc and three general-purpose registers gpr_src1, gpr_src2 and gpr_dst. The element also specified the form of the assembly code, e.g. r0 = eq r1, r2 for the equals operation.

```
enum opcode : uint8 {
  OPC_EQ,
  OPC_ULT,
};

element opc_eq {
  assembly { "eq" };
  binary { OPC_EQ };
  return { OPC_EQ };
};

element opc_ult {
  assembly { "ult" };
  binary { OPC_ULT };
  return { OPC_ULT };
};

set opc_cmp = opc_eq, opc_ult;
```

Listing 3.4: CodAL description of two operation codes.

```
element i_cmp {
  use opc_cmp as opc;
  use gpr_all as gpr_dst, gpr_src1, gpr_src2;

  assembly { gpr_dst "=" opc gpr_src1 "," gpr_src2 };
  binary { opc gpr_dst gpr_src1 gpr_src2 };

  semantics {
    uint32 src1, src2;

    src1 = rf_gpr[gpr_src1];
    src2 = rf_gpr[gpr_src2];

    switch (opc) {
      case OPC_EQ: rf_gpr[gpr_dst] = src1 == src2; break;
      case OPC_ULT: rf_gpr[gpr_dst] = src1 < src2; break;
    }
  };
};
```

Listing 3.5: CodAL declaration of an `i_cmp` instruction for performing comparisons.

Code in the semantics section may contain only a restricted subset of ANSI C that forbids the use of pointers, structures, `goto` directives and statements that combine variable declaration and initialization. The contents of registers are manipulated through array-like operations on the register file.

3.2 Development Tools

The Codasip Studio IDE can generate a full SDK just from the processor model specified in the CodAL language. As discussed in the previous chapter, this fact eliminates the need for multiple separate teams, significantly reduces the development time, and avoids possible inconsistencies between the software tools and the hardware caused by human error. The SDK is composed of the following tools:

- Assembler converts the textual human-readable assembly code into a binary object file. The Codasip assembler is based on the LLVM compiler infrastructure.
- Linker combines multiple object files into a binary file that can be executed by the processor. Codasip Studio uses a GNU linker.
- Disassembler reads the executable file generated by assembler or linker and transforms it back into the original assembly code. This tool is again built on top of LLVM.
- Simulator mimics the behaviour of the processor – it fetches, decodes and executes instructions stored in the memory. Codasip SDK contains two interpreting simulators: the instruction-accurate (IA) and cycle-accurate (CA).
- Debugger allows developers to inspect the current state of the simulator. The Codasip debugger is a modified version of the LLDB debugger.
- Profiler collects data during the simulation and provides insights such as the number of clock cycles spent in each function and the number of instruction executions.
- Compiler converts code written in a high-level language into an executable binary file. The C/C++ Codasip compiler is a modified version of the Clang compiler – a part of the LLVM infrastructure.
- Standard libraries provide a common set of data structures and algorithms that can be used in C or C++ code to develop applications for the processor.

The rest of this thesis focuses on the compiler and the optimizations it performs. The basic structure of an LLVM-based compiler is described in Chapter 4 along with the compiler generation feature of Codasip Studio.

Chapter 4

The LLVM Compiler Infrastructure

The LLVM compiler infrastructure is an open-source set of modular and reusable compiler and toolchain technologies initially developed by Chris Lattner as a research project at the University of Illinois [11]. It includes tools such as optimizer, code generator, assembler, disassembler, linker, debugger and others.

LLVM builds on the idea of a three-phase pipeline architecture consisting of the *front-end*, the *optimizer* and the *back-end* [3]. All compilation phases work with the LLVM *intermediate representation* (IR), a generic assembly language described in Section 4.1.

The role of the front-end is to parse and analyse the source program and generate semantically equivalent IR. The optimizer then performs a sequence of optimizations on the IR that are driven by information about the target architecture, but still produce a target-independent IR. (Improving the quality of architecture information is the goal of this thesis.) The back-end then takes the optimized IR and emits the final target code. This process is illustrated in Figure 4.1 and Section 4.2 explains it in more detail.

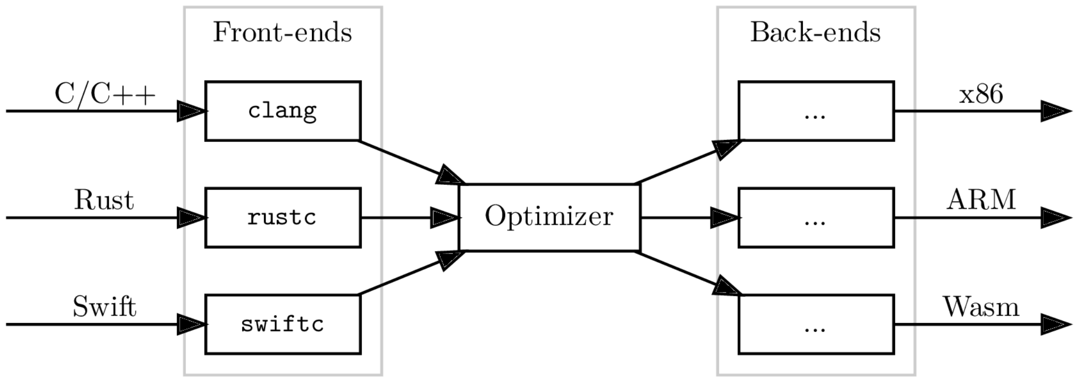


Figure 4.1: The three-phase architecture of LLVM's pipeline decouples language front-ends and architecture back-ends from the generic optimizer.

The translation phases are independent of each other, which enables compiler developers to easily add support for new processor architectures or to create new languages that can take advantages of existing optimizations and architecture support. LLVM is used in many compilers including Clang (C/C++), Swift, Rust, GHC (Haskell) or Flang (Fortran) and supports a wide variety of architectures: x86, ARM, WebAssembly, AMD GPU and more.

4.1 Intermediate Representation

The LLVM IR is a generic assembly language in SSA form [13]. It is designed to be generic with respect to the source programming language and the target processor architecture, but extensible (typically via metadata or target data layout). The SSA (static single assignment) form ensures that each variable is assigned exactly once and defined before its use, which simplifies and improves the results of a variety of optimizations [18]. The IR has three equivalent representations: as an in-memory data structure, as a bitcode stored on a disk, or as a human-readable assembly language.

```c
int factorial(int n) {
  int result = 1;
  for (int i = 1; i <= n; i++)
    result *= i;
  return result;
}
```

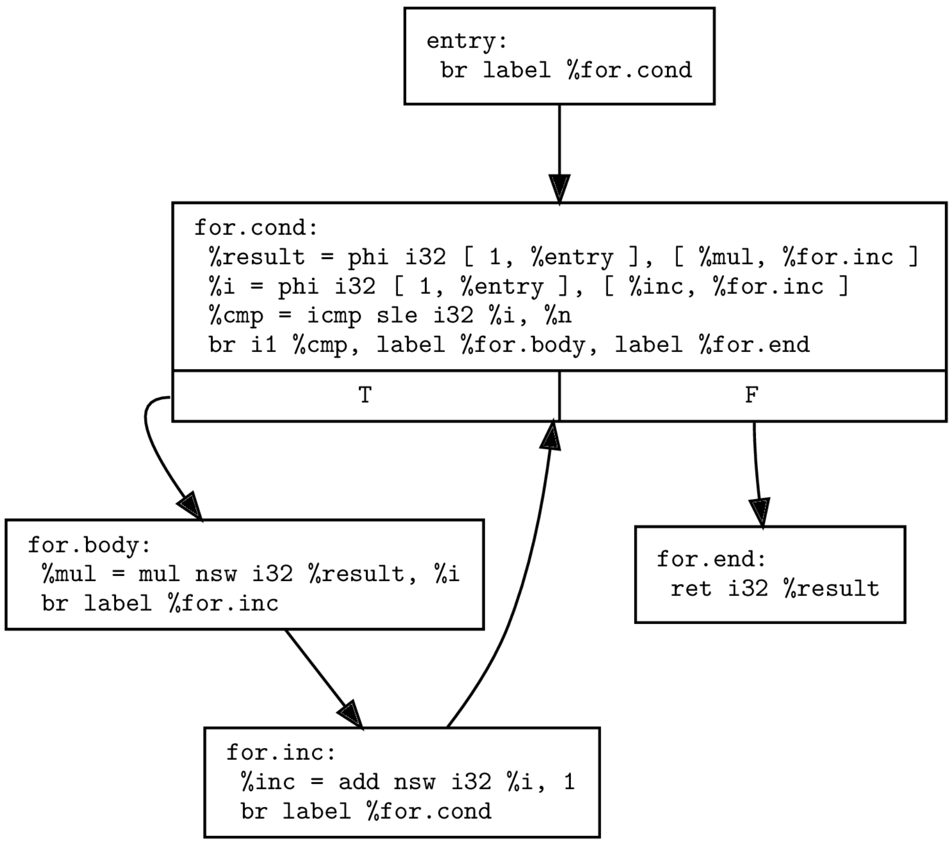
Listing 4.1: Example of an iterative factorial computation written in the C language.

Listing 4.1 shows an example of an iterative factorial function written in the C language. Listing 4.2 then contains the equivalent LLVM IR code that has been translated and slightly optimized by the Clang compiler. The following sections will refer back to this example to describe individual features of the intermediate representation.

```llvm
define i32 @factorial(i32 %n) {
entry:
  br label %for.cond
for.cond:
  %result = phi i32 [ 1, %entry ], [ %mul, %for.inc ]
  %i = phi i32 [ 1, %entry ], [ %inc, %for.inc ]
  %cmp = icmp sle i32 %i, %n
  br i1 %cmp, label %for.body, label %for.end
for.body:
  %mul = mul nsw i32 %result, %i
  br label %for.inc
for.inc:
  %inc = add nsw i32 %i, 1
  br label %for.cond
for.end:
  ret i32 %result
}
```

Listing 4.2: The LLVM IR representation of the code from Listing 4.1.

The LLVM IR code can also be visualized as a control flow graph (CFG) – a graphical representation of the program that groups blocks of code into a directed graph showing the flow of the computation. This form makes it easier for compiler developers to see the effects of non-trivial transformations. The CFG for the LLVM IR code from Listing 4.2 is shown in Figure 4.2.



```
entry:
 br label %for.cond
```

```
for.cond:
 %result = phi i32 [ 1, %entry ], [ %mul, %for.inc ]
 %i = phi i32 [ 1, %entry ], [ %inc, %for.inc ]
 %cmp = icmp sle i32 %i, %n
 br i1 %cmp, label %for.body, label %for.end
```

```
for.body:
 %mul = mul nsw i32 %result, %i
 br label %for.inc
```

```
for.end:
 ret i32 %result
```

```
for.inc:
 %inc = add nsw i32 %i, 1
 br label %for.cond
```

Figure 4.2: The control flow graph of the IR code from Listing 4.2.

4.1.1 Identifiers

Identifiers in the IR are of two kinds: global and local. Global identifiers such as functions or global variables are prefixed with `@` (`@factorial`) and local ones start with `%` (`%result`). The prefixes help to distinguish names from reserved words such as types (`i32`, `void`), opcodes (`add`, `ret`) or constants (`false`, `42`).

4.1.2 Structure

Programs in intermediate representation are structured into several basic components. The following sections explain their purpose and the relationships between them.

Modules

Modules are the top-level containers of the IR. Each module contains a list of global variables, functions, symbol table, target information (*triple*, data layout) and references to other modules it depends on. The scope of modules varies between compilers. In Clang, each translation unit (a single source file with all of the included headers) lives in its own module, while the Rust compiler compiles the whole library into a single module [20].

Functions

Declarations and definitions of functions in LLVM IR consist of the `declare` or `define` keywords, function name, its return type and a list of parameters. Also, several attributes can be associated with the function: linkage type, visibility style, calling convention, inlining preferences and more. The function parameters may also be tagged with various attributes, e.g. alignment, aliasing, non-null. The function definition contains a list of basic blocks, described in the following section.

Basic Blocks

A basic block is the longest sequence of instructions that are always executed in order. They have exactly one entry point and one exit point. Due to these restrictions, jump instructions can only occur at the end of a basic block, and their destination is always the first instruction of a basic block.

Basic blocks form the control flow graph (CFG) of functions and as such, are convenient for program optimizations. In LLVM, a basic block starts with a label and holds a list of instructions that end with a *terminator* instruction.

Instructions

The LLVM IR instruction set is a low-level representation of an abstract virtual machine that expresses the key operations of ordinary processors [11]. The instruction set operates on an infinite set of typed virtual registers that can hold values of primitive types. Instructions in the IR are polymorphic, which means that a single instruction can operate on several types of operands.

4.1.3 Instruction Types

The instruction set of LLVM IR can be divided into several parts. The following sections describe simple versions of the essential instructions used in code samples in later chapters.

Terminator Instructions

Terminator instructions always appear at the end of a basic block and cause a transfer of the control flow. Two essential instructions are the following:

- `ret`: return the control flow from a callee function to the caller. There are two versions of the instruction: `ret void` is used to return from a void function, whereas `ret <type> <value>` returns a value from a non-void function.
- `br`: transfer the control flow from one basic block to another one in the same function. `br label <dest>` performs an unconditional jump to the specified destination and `br i1 <cond>, label <then>, label <else>` performs a conditional jump to one of the two labels based on the condition value.

The remaining instructions in this category include `switch`, `indirectbr` or `unreachable` and instructions necessary for exception handling.

Binary Instructions

Binary instructions execute an operation on two operands of the same type and produce a single value of the same type. They can be divided into two basic categories:

- arithmetic instructions (e.g. `add`, `sub`, `mul`, floating-point `fmul`, ...),
- bitwise operations (e.g. `and`, `or`, `shl`, ...).

Memory Instructions

The SSA representation in LLVM does not model memory [11] – IR instructions treat memory as a single mutable object. There are just a few instructions in LLVM IR that interact with memory:

- `alloca`: allocates memory on the stack frame of the current function and return its address. Memory allocated by this instruction is uninitialized and gets automatically released when the function returns. It may also specify the number of elements to allocate, explicit alignment and address space.
  `<ptr> = alloca <type>`
- `load`: reads from memory at the specified address.
  `<val> = load <type>, <ptr_type>* <ptr>`
- `store`: writes a value to a memory with the specified address.
  `store <type> <val>, <ptr_type>* <ptr>`
- `getelementptr`: calculates the address of a subelement of an aggregate type – an array or a structure.
  `<ptr> = getelementptr <type>, <ptr_type>* <ptr> <el_type> <el_id>`

Aggregate Operations

Aggregate operations work with aggregate types – arrays and structures:

- `extractvalue`: extracts a value at specified index from an aggregate type. It is similar to `getelementptr`, but works on values instead of pointers, and indices are required to be in bounds.
  `<el> = extractvalue <aggregate type> <val>, <el_id>`
- `insertvalue`: inserts a value into a member field of an aggregate type and return the modified aggregate type.
  `<res> = insertvalue <aggregate type> <val>, <el_ty> <el>, <el_id>`

Vector Instructions

LLVM IR supports vector instructions that may get translated into SIMD operations on targets with such feature. Vector types are supported by many instructions thanks to their polymorphic nature and are written in the form `<N x <type>>`. There are a few operations specific to vectors:

- `extractelement`: extracts a scalar element at the specified index in a vector
  `<el> = extractelement <N x <type>> <vec>, <el_type> <id>`

- `insertelement`: inserts a scalar element at the specified index in a vector
  `<vec> = insertelement <N x <ty>> <vec>, <el_ty> <el>, <id_ty> <id>`

- `shufflevector`: combines two vectors of the same type and length N. Elements of the first vector are indexed from 0 to $N-1$ and elements of the second vector from N to $2N-1$. The mask is a vector of length M with `i32` indices. The result of this instruction is the mask with each index replaced with the value from one of the vectors.
  `<r> = shufflevector <N x <ty>> <v1>, <N x <ty>> <v2>, <M x i32> <mask>`

Other Instructions

Other frequently used instructions include:

- `icmp`: returns a boolean result of the specified operation applied to two operands of the same type. The condition operation can be for example `eq` (equal), `ugt` (unsigned greater than), `slt` (signed less than) and other.
  `<result> = icmp <cond> <type> <op1>, <op2>`

- `phi`: the ϕ node in SSA representation. From a list of incoming values takes the one coming from the basic block that executed prior to the current basic block.
  `<result> = phi <type> [<val>, <label>]*`

- `select`: chooses one value based on a condition.
  `<result> = select <type> <cond>, <type1> <value1>, <type2> <value2>`

- `call`: calls a function with the specified arguments.
  `<result> = call <fn> (<args>)`

4.1.4 Intrinsic Functions

LLVM IR also supports *intrinsic functions*. Their semantics is built directly into LLVM, and their name starts with the `llvm.` prefix. Intrinsic functions can serve as a customization point for new front-ends and back-ends alike – adding new intrinsic functions is significantly easier then extending the IR instruction set. [12]

The intermediate representation has already many target-specific and target-independent intrinsic functions built-in, e.g.: garbage collection intrinsics, intrinsics for standard C library functions, bit manipulation, arithmetic with overflow, vector and matrix operations and many more.

4.2 Compilation Phases

This section follows the compilation process of an LLVM-based compiler and explains how code written a high-level language gets transformed into executable binary code.

4.2.1 Front-End

LLVM front-ends usually start the compilation process by parsing the input source file and analysing its semantics. If the input is deemed to be valid, the output of this phase is an *abstract syntax tree* (AST).

Some compilers (including Clang) build the LLVM IR by recursively visiting the AST nodes and emitting the corresponding instructions through LLVM's *IR builder*. Other compilers first transform the tree into a custom intermediate representation (MIR in Rust or SIL in Swift) designed to enable language-specific analyses and optimizations. Some of those could not be performed directly on LLVM IR due to its generic/abstract nature. The language-specific intermediate representation is then mapped onto regular LLVM IR.

4.2.2 Optimizer

The LLVM optimizer consumes the IR generated by a front-end and produces its optimized version. The goal of optimizations is typically to improve the run-time performance of the given code or to reduce its size. The optimizer achieves that by running a sequence of so-called *passes*, which are of two kinds:

- *analysis passes* deduce some properties of the IR code without modifying it,
- *transformation passes* use the information provided by analysis passes to transform the IR code into equivalent IR code that meets the optimizer's goal.

Each pass in the LLVM optimizer runs over a particular unit of the IR: the whole module, a strongly connected component of its call graph, single function, or a loop.

LLVM's optimizer is capable of a wide range of transformations including constant propagation, constant hoisting, loop optimizations (fusion, rotation, unrolling, unswitching, vectorization, ...), inter-procedural optimizations (inlining, dead argument elimination, devirtualization, hot/cold splitting, ...), instruction combining, CFG simplification, and many more.

Pass Manager

The *pass manager* orchestrates and executes the transformation passes. Pass manager receives a sequence of the desired transformation passes, typically specified by the compiler front-end and customizable via command-line flags (e.g. `-O3` or `-Os` in Clang).

A transformation pass can request analysis results for the processed IR unit through the *analysis manager*. The analysis and pass managers work together to avoid recomputing analyses results in situations where an earlier result is still valid. This lazy behaviour is possible since transformation passes report which analyses are invalidated by the performed IR transformations. When a transformation pass reports it invalidated the result of a particular analysis, that analysis pass is not run until another pass requests its result.

One analysis pass may fetch the result of another analysis and, therefore, potentially invoke its evaluation. Transformation passes, on the other hand, cannot invoke other transformation passes. Their ordering is orchestrated exclusively by the pass manager.

Target Information

In generating high-performance code, the optimizer needs to make informed decisions based on the properties of the target architecture. In LLVM, transformation passes may use the *target transform info* (TTI) interface to query information such as:

- cost of instructions, memory operations, intrinsics, function calls, immediate values,
- legality of addressing modes, immediates, masked operations, vector instructions,

- loop unrolling preferences,
- width of vector registers,
- other capabilities of the architecutre.

One example of such transformation pass is *loop unrolling*, which queries the target transform info for detailed preferences regarding this optimization. The target may specify parameters including but not limited to cost threshold (the maximum allowed cumulative cost of instructions in the unrolled loop body), unrolling factor (the maximum number of body copies), flags that signify whether a loop remainder is allowed, or disable the optimization for loops with a dynamic number of iterations. Choice of the parameters may depend on the relative cost of jump/division instructions, cache behaviour or the memory constrains.

Another example is *hot/cold splitting*, where a sequence of instructions is outlined into a separate function only if their cumulative size cost reaches a certain threshold. There are also cases where target transform info prevents optimizations that would cause issues during later stages of the compilation process. For example, *vectorizing loops* naturally does not benefit targets with no vector registers – an information that is again exposed by the TTI interface.

While this part of the optimizer might seem to conflict with the three-phase architecture introduced in the beginning of this chapter, it is essential to note that the optimized code is still valid and generic IR. The optimizer does not produce a target-specific representation of the program and can work in the absence of target information.

4.2.3 Back-End

The LLVM back-end lowers optimized IR into native code. The back-end is similar to the optimizer in that it uses generic algorithms that query a set of interfaces to obtain information about the target. The primary interface is *target machine*[1] which consists of several parts [14]:

- *data layout* specifies memory layout, alignment requirements, pointer size, endianness,
- *target lowering* describes how IR should be transformed into low-level representation, particular registers required by instructions, or natively supported operations,
- *target register info* defines the register file and interactions between registers,
- *target instruction info* describes the target's machine instructions,
- *target frame lowering* exposes details about the stack frame layout,
- *target subtarget* specifies the micro-architectural details of the processor: supported instructions, their latencies, and scheduling details,
- *target JIT info* provides information necessary for enabling just-in-time compilation.

[1]The *target machine* interface is usually also used by the *target transform info* interface in the optimizer.

Code Generation Process

The code generation process *lowers* the target-independent LLVM IR into target machine code and consists of several phases:

1. In the *instruction selection* phase, the program in LLVM IR is converted into target-independent *selection DAG* (directed acyclic graph). The DAG is subsequently transformed to only use types and operations supported by the target.

   The selection DAG is then matched against a specified set of graph *patterns*. Each matched subgraph is replaced by another subgraph containing target-specific operations that can use physical registers. Both the patterns and the target-specific graphs are small instances of selection DAG themselves. Therefore the central part of code generation reduces to matching and replacing pieces of the program's selection DAG.

   Pattern graphs and target-specific graphs are defined in a meta-language *TableGen*. The definitions get transpiled into a table-like structure in C++ used by the LLVM's matching algorithm.

2. During *scheduling and formation*, the DAG nodes are reordered to minimize register pressure and align instruction latencies. The DAG is then transformed into a list of *machine instructions*.

3. Machine instructions then undergo *SSA-based machine code optimizations*. The optimizations are usually target-specific, however, some target-independent optimizations such as loop invariant code motion or duplicate instruction elimination are also run.

4. The *register allocation* phase maps remaining virtual registers onto physical ones.

5. Next, the *prologue and epilogue code* for function calls and stack frame management is inserted and optimized.

6. *Late machine code optimizations* like peephole optimizations and register spill code scheduling form the final version of the machine instructions code.

7. Finally, the result is emitted either as machine code or assembly.

Figure 4.3 depicts the selection DAG of the `for.body` basic block from Listing 4.2. The DAG already underwent the instruction selection phase – the generic `mul` instruction was replaced by `IMUL32rr` specific to the x86 architecture.

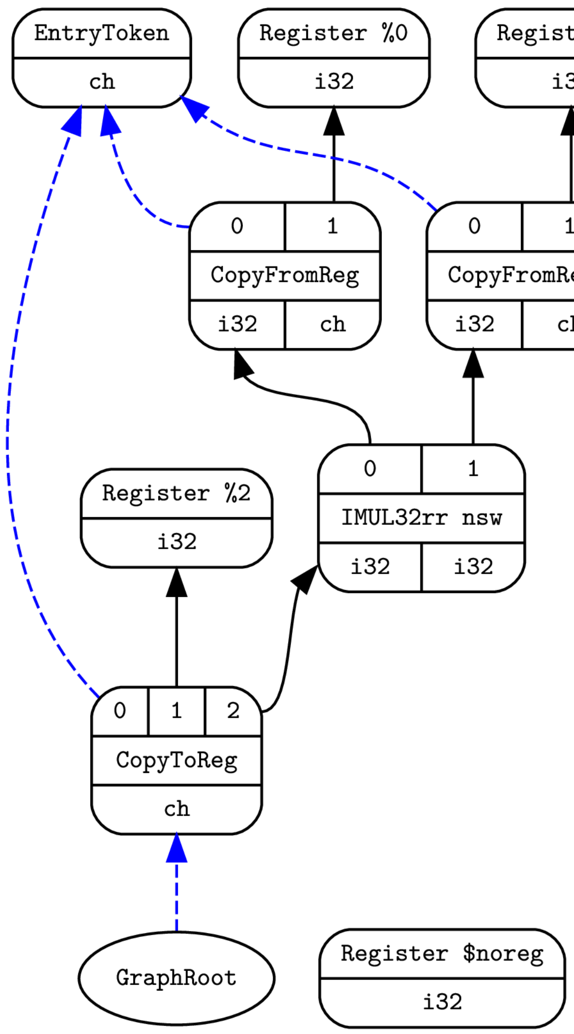


Figure 4.3: Selection DAG of the code from Listing 4.2 after instruction selection.

4.3 LLVM in Codasip Studio

Chapter 3 listed the tools contained in the SDK generated from the CodAL description of a processor. Among other tools, the SDK includes a C/C++ compiler Clang based on the LLVM compiler infrastructure introduced in previous sections. The rest of this chapter briefly explains the process of compiler generation – a prerequisite for Chapter 5 that describes the proposed system for providing architecture information to the optimizer.

4.3.1 Back-end Generation

The compiler generation process is depicted in Figure 4.4 and begins with *semantic extractor* pre-processing the CodAL model. The output from the extractor is then consumed by the *back-end generator* that creates C++, and TableGen source files specific to the target architecture. These files are translated by the LLVM build system into the `llc` back-end executable. Later, during the compilation of a C/C++ source code for the target architecture, `llc` is invoked by the `clang` driver and outputs the machine code for architecture described through the CodAL model [5].

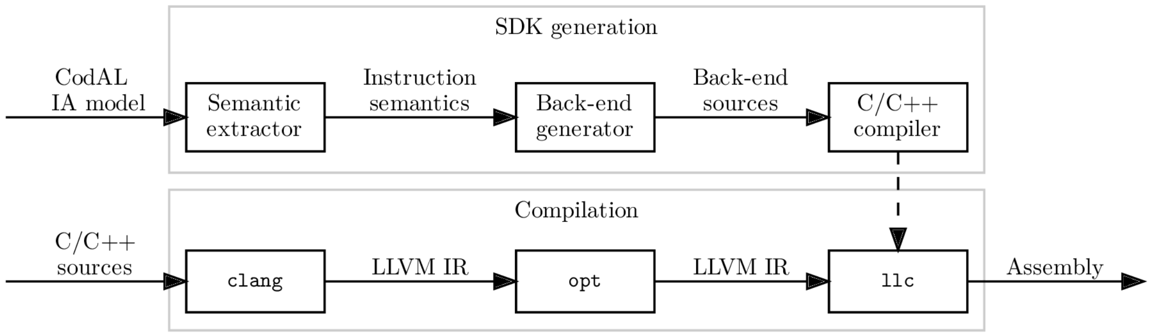


Figure 4.4: Besides other things, the SDK generation process generates the `llc` executable that is later used to compile C/C++ code.

Internally, the semantics of each instruction get translated into LLVM IR and subsequently into a selection DAG. The selection DAG can be used by the LLVM's instruction selector to match patterns in the selection DAG of the program [6] and replaced by the target's binary representation of the instruction. Therefore, the DAGs get serialized into a TableGen file and compiled with the rest of the target-specific source files into a complete LLVM back-end as described earlier in this chapter.

4.3.2 Front-end Arguments

The specifications of both the C and C++ languages depend on the target architecture – the pointer size, the width of fundamental types and the required/preferred alignment is derived from the target's data layout. Thus, generating only the compiler back-end is not enough to create a fully working compiler. Therefore, the Codasip back-end generator also emits a list of command-line arguments that are applied to each invocation of the compiler and specify the processor's data layout.

4.3.3 Optimizer Hints

Moreover, the back-end also generates a `simd_info.txt` file. The file contains the number and width of registers, legal data types and a matrix of operations and data types (supported combinations of types and operations are denoted by `1`, unsupported combinations with `-1`). An excerpt from the file with optimizer hints is shown in Listing 4.3. This file is loaded by the `TargetMachine` class in the compiler and mapped onto the TTI interface.

```
processor codix_berkelium.ia
widest_scalar_registers 31:32
widest_vector_registers 0:0
legal_types i32

EntryToken i256:1 i512:1 i1024:1 i2048:1 v2i128:1 v256i1:1 ...
TokenFactor i256:1 i512:1 i1024:1 i2048:1 v2i128:1 v256i1:1 ...
...
add Other:-1 i1:-1 i8:-1 i16:-1 i32:1 i64:-1 i128:-1 f16:-1 f32:-1 f64:-1 ...
sub Other:-1 i1:-1 i8:-1 i16:-1 i32:1 i64:-1 i128:-1 f16:-1 f32:-1 f64:-1 ...
mul Other:-1 i1:-1 i8:-1 i16:-1 i32:1 i64:-1 i128:-1 f16:-1 f32:-1 f64:-1 ...
...
fadd Other:-1 i1:-1 i8:-1 i16:-1 i32:-1 i64:-1 i128:-1 f16:-1 f32:-1 f64:-1 ...
```

Listing 4.3: Excerpt from the `simd_info.txt` file with optimizer hints.

While this amount of information can improve a handful of optimizations (e.g. simple vectorization), it is far from ideal. The optimizer often asks complex questions (e.g. whether it is viable to unroll some particular loop) that cannot be answered based on the provided information. Moreover, users of Codasip Studio do not have the option to hand-tune the generated information or add their own logic for helping the optimizer make good decisions.

Chapter 5

Design of Architecture Information Integration

In Chapter 4, the importance of architecture information for the LLVM optimizer has been established. Without detailed information about the target, the optimizer cannot make sensible decisions, and the quality of the generated code can be far from optimal. This leads to worse characteristics of the resulting system – slower execution speed, greater code size and higher resource usage.

The goal of this thesis is to design a mechanism for providing architecture information to Clang (the LLVM-based compiler used for application development within Codasip Studio) and its optimizer. Most of the information should be automatically extracted from the CodAL processor model. However, users of Studio should have the possibility to fine-tune the information and extend it with their knowledge of the platform at hand.

5.1 Integration into Codasip Studio

Section 4.3 described the compiler generation process in Codasip Studio. The back-end source files are generated from the CodAL model and compiled into the `llc` executable, as shown in Figure 4.4. The back-end generator also emits a text file that is dynamically processed by the compiler driver and provides the essential target information to the front-end and optimizer.

I decided to follow the principles behind the back-end generator in order to achieve the desired level of customizability. First, the SDK generation inside Codasip Studio needs to be extended with a new subprocess for analysing the CodAL model and deducing the architecture information relevant for optimizations. Second, the architecture information should be emitted as a set of C++ source files to allow arbitrary modifications of the logic driving LLVM optimizations. Finally, the source files should be compiled and used within the LLVM optimizer.

However, re-linking the whole compiler front-end and optimizer each time the CodAL model changes would be inefficient and time-consuming. Therefore, the architecture information shall be compiled into a lightweight shared library and dynamically loaded by the custom version of Clang that ships with Codasip Studio.

The shared library shall export the implementation of the target transform info (TTI) interface – the main customization point of LLVM's optimizer described in Chapter 4. The proposed new part of the SDK generation process and its interaction with the compilation pipeline is depicted in Figure 5.1.

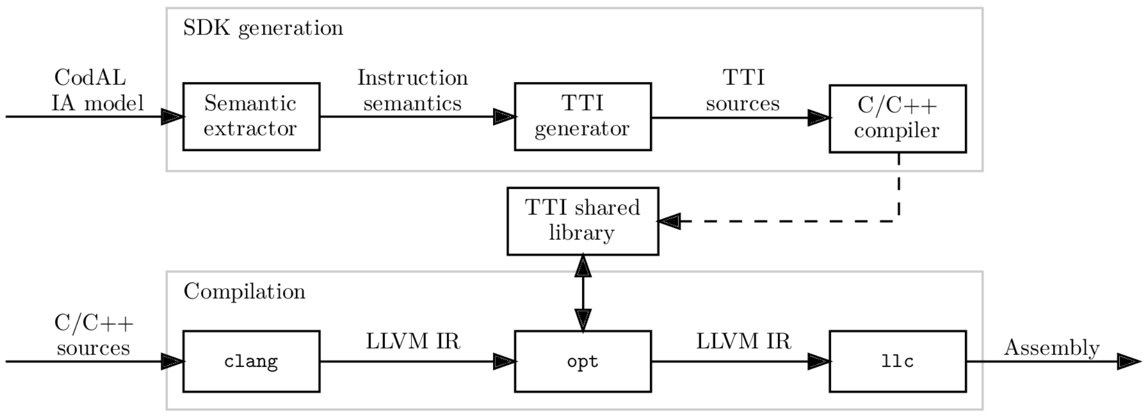


Figure 5.1: The SDK generation process produces a shared library of the TTI implementation that is loaded and queried by the optimizer.

Several steps need to be carried out to implement the proposed solution:

1. Analyze LLVM's TTI interface and its usage within the optimizer. The number of functions exposed by the interface is substantial, but not all of them are necessarily relevant for the processors and applications of a typical Codasip Studio user. Finding a set of functions that have the greatest impact on the quality of generated code can be achieved by statically analysing the optimizer source code, studying established architectures within LLVM and their TTI customizations, or monitoring most frequently used TTI functions during the compilation of a representative application.

2. Create a new tool for the automatic extraction of the information deemed useful from the processor CodAL model. This should build on the infrastructure already present in the Codasip Studio codebase. The information may include various aspects of the processor such as the number and size of registers, natively supported operations, latency and size of instructions and more.

3. Generate C++ code that exposes the extracted architecture information through the TTI interface and make it easily editable by processor designers.

4. Compile the code into a shared library that exposes a simple interface. Seamlessly integrate the build process into the Codasip Studio IDE.

5. Customize the Clang compiler so that it locates the shared TTI library, loads the generated TTI implementation and uses it in the optimizer.

The details of the above steps are elaborated in Chapter 6. Chapter 7 tests the infrastructure by fine-tuning the TTI code generated for specific processors and evaluates the impact of better architecture information on code size and performance.

Chapter 6

Implementation of the TTI Generator

This chapter describes the integration of processor architecture information into the Codasip compiler. In the first part, the usage of the target transform info (TTI) interface by the LLVM optimizer is analysed. Next, the extraction of useful processor properties from the CodAL model is outlined. Later, the generation of C++ source files of TTI is described. The following section deals with compiling the shared TTI library and loading it in the compiler, and finally, the integration of the solution into the Codasip Studio IDE is dealt with.

6.1 TTI Usage Within the LLVM Optimizer

The TTI interface was introduced in LLVM 3.2[1] (released in early 2013) and originally consisted of only seven functions deciding the legality of immediate operands, addressing modes, types, and reporting the size and alignment of the jump buffer. Over the years, many more functions were added to support new kinds of optimizations on new target architectures. The TTI interface in LLVM 9.0 (used by the Codasip compiler) contains over 120 functions providing target information to the optimizer. A deeper analysis is necessary to identify what parts of the interface are relevant for most of the optimizations and processor architectures.

6.1.1 Source Code Analysis

To gain a basic insight into the usage of TTI inside the optimizer, the LLVM source code was analysed. Overall, the LLVM cost model interface is queried from 260 distinct source code locations across 43 transformation and analysis passes.

TTI Usage in Passes

Table 6.1 shows an overview of the most frequent users of the interface, of which the most prominent ones are the loop and SLP[2] vectorizers. This suggests that the vectorization passes could benefit the most from an accurate cost model.

[1] `https://github.com/llvm/llvm-project/commit/e10328737`

[2] SLP (superword-level parallelism) vectorizer merges consequent scalar operations into vector operations.

LLVM pass	**Queries**
Loop vectorization	68
SLP vectorization	51
Loop strength reduction	31
Load/store vectorization	13
Function inlining	12
Phi speculation	9
Constant hoisting	7
CFG simplification	6
Loop data prefetching	5
...	...

Table 6.1: Number of cost model queries in the source code of LLVM passes.

TTI Function Queries

Table 6.2 then lists the TTI functions that are referenced most frequently across all passes. The names of shown functions usually capture the basic idea behind their semantics: the cost of an IR construct (user) is reported by the `getUserCost` function, other functions report the costs of arithmetic instructions, vector shuffle operations, memory operations, specific vector instructions, type conversions, compare and select operations, or the cost of an integer immediate value.

TTI function	**Queries**
`getUserCost`	14
`getArithmeticInstrCost`	13
`getShuffleCost`	11
`getMemoryOpCost`	9
`getVectorInstrCost`	9
`getCastInstrCost`	8
`getCmpSelInstrCost`	7
`getInstructionCost`	7
`getIntImmCost`	7
...	...

Table 6.2: Number of calls to TTI functions in the source code of LLVM passes.

Types of Cost Models

All of the listed TTI functions represent the cost of an operation as a single unsigned integer. In reality, however, the cost of an operation highly depends on the optimization goal. If the goal is code size reduction, long instructions should have a greater cost than short ones. On the other hand, when optimizing for run-time performance, the cost of operations should be based on properties such as the instruction latency.

While LLVM 9.0 does understand the concept of different cost model types (i.e. code size, latency, and reciprocal throughput), it is mainly used to choose one from several preconfigured optimization pipelines. In TTI, it is only used to parametrize a single function (`getInstructionCost`) that is used in four optimization passes. The rest of the TTI interface tries to express the different costs of operations as a single number. This is clearly an aspect of the optimizer and cost model that should be corrected. In April 2020, an RFC[3] (request for comments) was submitted to the LLVM-dev mailing list proposing being explicit regarding the used cost model in the TTI interface.

6.1.2 Case Study: Commonly Customized TTI Functions

In finding what TTI functions should be customized for improving the cost model accuracy, two well-developed target architectures were analyzed: x86 and 64-bit ARM. The TTI behaviour is not customizable directly in the `TargetTransformInfo` class, and target implementers are encouraged to create a new class inheriting from `BasicTTIImplBase<T>` instead. This class (along with *its* base classes `TargetTransformInfoImplCRTPBase<T>` and `TargetTransformInfoImplBase`) provides the default cost model and ability to guess the cost of a complex operation by splitting it into more basic operations whose cost is known. The main TTI interface stores an instance of this class hierarchy and forwards all queries. The UML diagram for the TTI infrastructure of x86 architecture is depicted in Figure 6.1 below.

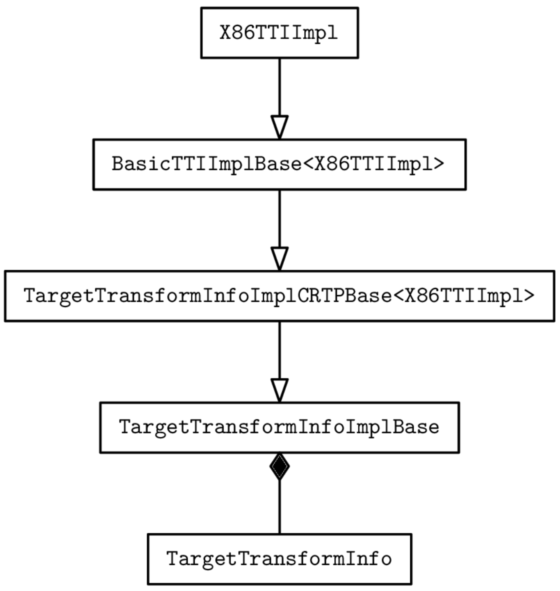

X86TTIImpl
BasicTTIImplBase<X86TTIImpl>
TargetTransformInfoImplCRTPBase<X86TTIImpl>
TargetTransformInfoImplBase
TargetTransformInfo

Figure 6.1: The UML diagram of the x86 `TargetTransformInfo` hierarchy.

Both targets customize the functions listed in Table 6.2 and some of their dependencies. In addition, they tweak the interleaving decisions, provide costs of intrinsics, address computation, arithmetic reduction, and report the level of popcount support: whether it is implemented in hardware or needs to be emulated by software.

It is worth noting that most of the customization logic deals with vector capabilities of both respective architectures. For the 64-bit ARM TTI, this means the NEON extension. On the x86 side, extensions like AVX512, AVX2, SSE2 are handled in addition to various micro-architectural specifics. These findings fall in line with results from the previous chapter: the TTI interface in LLVM 9.0 serves mainly the needs of loop, and SLP vectorization passes.

[3]http://lists.llvm.org/pipermail/llvm-dev/2020-April/141263.html

6.1.3 Dynamic Analysis

Previous sections inspected optimizer's usage of the TTI interface by analysing the LLVM source code. I have looked into the optimizer run-time behaviour to confirm the most viable customization points in the TTI interface. A logging system for the TTI was developed (described in following Section 6.3) to capture which functions are being called, what are the arguments and return values, and what transformation passes made the calls. This mechanism was applied to functions customized by either the x86 or ARM target. Three sets of programs were tested:

- Basic: a battery of basic algorithms: bitcount, CRC, Dhrystone, Dijkstra's algorithm, square root calculation, quicksort, SHA, string comparison, and Pratt-Boyer-Moore string search.
- Dhrystone: a widely-used synthetic benchmark focused on measuring processors' integer performance.
- CoreMark: another synthetic integer benchmark. Unlike with Dhrystone, the work in CoreMark cannot be optimized away by the compiler.

Table 6.3 shows the number of calls to each TTI function during optimization of a program set with different optimization goals (`-O3` for performance, `-Os` for size). The default LLVM TTI implementation was used. The most frequently called function is by far `getUserCost`, which internally falls back on `getOperationCost` during non-trivial queries (i.e. not a phi node, `alloca` instruction or similar). This function is also used for querying the size cost of an operation from `getInstructionCost`. Less frequently used hooks deal with target memory intrinsics, the unrolling of loops, inlining of functions, memory interleaving, popcount support, arithmetic instruction cost and the number of registers.

TTI function	**Basic**		**CoreMark**		**Dhrystone**	
	`-O3`	`-Os`	`-O3`	`-Os`	`-O3`	`-Os`
`getUserCost`	21939	16276	47271	22745	2619	2186
`getOperationCost`	8786	7122	20323	12254	1249	940
`getTgtMemIntrinsic`	2690	2473	3605	2299	428	417
`getUnrollingPreferences`	190	202	376	234	26	28
`getNumberOfRegisters`	114	112	152	152	56	56
`getMaxInterleaveFactor`	57	56	76	76	28	28
`areInlineCompatible`	53	48	100	108	17	19
`getInliningThresholdMultiplier`	41	36	106	120	17	19
`getPopcntSupport`	35	25	92	48	8	8
`getArithmeticInstrCost`	2	2	40	36	4	4

Table 6.3: Number of cost model queries performed during optimization with the default TTI implementation in LLVM.

6.2 Extracting CodAL Model Properties

The Codasip Studio codebase includes *semantic extractor* – a tool for working with the instruction semantics. Semantic extractor analyses the instruction element and its semantics (written in simplified ANSI C), generates multiple specializations if the instruction uses a `set` (e.g. an operation code), and converts the results into the LLVM selection DAG representation. The output of semantic extractor can be traversed via provided tree visitation facilities.

To derive the cost model from the CodAL processor model, the output of semantic extractor is analysed with the help of the provided infrastructure. The tree data structure is traversed, and the following information is extracted:

- number and width of scalar/vector registers,
- cost of scalar/vector memory access,
- cost of scalar/vector arithmetic/logic binary operations,
- cost of scalar/vector comparison operations,
- support & cost of vector shuffle operations.

Register Information

The number and width of registers is extracted from the list of processor's register classes through the semantic extractor interface. An example of a register class was shown back in Chapter 3. However, register classes do not directly declare what data types they store. This is necessary for deciding whether the registers are scalar or vector and satisfying the TTI interface.

In instruction semantics, the type of each register is implicitly scalar. If the designers want to treat it as a vector type, they need to cast their contents into a vector type explicitly. The TTI generator leverages this to identify register classes. For each code fragment matching the pattern in Listing 6.1, it marks the register class of `reg` as a vector class.

```
%1 = v4i32 regop(reg);
```

Listing 6.1: The pattern for matching vector register operands.

Memory Access

The cost of memory access can differ a lot between different platforms due to the memory type, bus speed and latency, used protocol, misalignment penalties, and more. To find out what data types can be loaded from and stored to the memory, the instruction semantics are searched for the patterns in Listings 6.2 and 6.3.

```
%x = i32 regop(reg);
store(i32 %x, i32 %y, 0);
```

Listing 6.2: The pattern for matching store operations.

```
%x = i32 load(i32 %y, 0);
regop(reg) = i32 %x;
```

Listing 6.3: The pattern for matching load operations.

Processor designers should explicitly specify the latency of memory operations to keep the IA model consistent with the CA model. If the latency is missing, it is implicitly set to 1 clock cycle. In the generated cost model, the cost of load/store operation is the same as the instruction latency.

Arithmetic/Logic Operations

Arithmetic/logic binary operations implemented by the processor can be discovered by visiting all binary operations where the left operand is a register, and the right operand is a register or an immediate value. Listing 6.4 shows the pattern for matching arithmetic/logic operations with two register operands.

```
%x = i32 regop(reg_1);
%y = i32 regop(reg_2);
%z = i32 <op>(i32 %x, i32 %y);
regop(reg_0) = i32 %z;
```

Listing 6.4: The pattern for matching arithmetic/logic operations with register operands.

Comparison Operations

The selection DAG representation of comparison operators is distinct from that of arithmetic/logic binary operations and are therefore handled separately. Comparison operations are typically cheap, and are thus implicitly assigned the cost of 1. The pattern for matching comparison operations with two registers is shown in Listing 6.5.

```
%a = i32 regop(reg_2);
%b = i32 regop(reg_1);
%c = i1 <op>(i32 %a, i32 %b);
%d = i32 zero_extend(i1 %c);
regop(reg_0) = i32 %d;
```

Listing 6.5: The pattern for matching comparison operations with two register operands.

Vector Shuffles

The support of the shuffle operation is essential for any processor with support for vector data types: it allows changing the order of vector elements in place. The LLVM loop vectorizer checks the cost of the *reverse* shuffle operation when analysing a loop that iterates in reverse order. When the target does not support the reverse shuffle, the vectorization may fail due to the high cost of scalarization of the operation. The extractor looks for the pattern in Listing 6.6 and assigns a low cost to the shuffle operation when found to prevent such failures.

```
%a = v4i32 BUILD_VECTOR(3, 2, 1, 0);
%d = v4i32 shuffle(v4i32 %b, v4i32 %c, v4i32 %a);
```

Listing 6.6: The pattern for matching shuffle operations.

6.3 Generating TTI Source Code

As outlined in Chapter 5, the intermediate output of the newly developed system should be a set of C++ files customizing the TTI interface. This allows engineers working with a CodAL model to hand-tune the cost model logic through a language they are most likely already familiar with. The generation of C++ source code in the Codasip Studio tools is usually achieved in one of two ways:

- The *templgen* tool consumes a file written in a templating language where regular C++ code can be enriched by a markup that gets expanded based on the provided data.
- The *ocstream* library provides an interface to declare a class (its name, member attributes and functions, included header files) and define the functions via output streams. The library then generates the header and implementation files.

The TTI source code generation is handled by the ocstream library, as it offers a higher degree of flexibility compared to templgen.

Cost Tables

The generated functions returning costs of operations use an internal cost table to answer queries, same as the TTI customizations of upstream targets. In case the function receives a query referring to an operation or a data type not supported by the processor, the query is forwarded to the basic TTI implementation, and its result is made more expensive. This should make the optimizer avoid placing such operation/type into the program. The back-end then has a more straightforward job lowering and legalizing the code, and the output should be of higher quality. An example of a TTI function implemented using an internal cost table is shown in Listing 6.7.

```
unsigned CodasipTTIImpl::getArithmeticInstrCost(unsigned Opcode, Type *Ty, ...) {
    std::pair<int, MVT> LT = TLI->getTypeLegalizationCost(DL, Ty);
    int ISD = TLI->InstructionOpcodeToISD(Opcode);

    static const CostTblEntry CostTable[] = {
        { ISD::ADD, MVT::i32, 1 },
        { ISD::SUB, MVT::i32, 1 },
        { ISD::MUL, MVT::i32, 2 },
        { ISD::SDIV, MVT::i32, 32 },
        // ...
    };

    if (const auto *Entry = CostTableLookup(CostTable, ISD, LT.second))
        return LT.first * Entry->Cost;

    return 4 * BaseT::getArithmeticInstrCost(Opcode, Ty, ...);
}
```

Listing 6.7: Code generated by the TTIGen tool.

Besides the natively supported types (`i32` in the case above), the cost table holds entries for narrower types (`i16` and `i8`). The reason for that is the fact that the narrower types are, in many cases, extended to the natively supported type without any performance penalties. This aspect of cost modelling is usually encoded in the *target lowering info* (TLI), which allows querying the cost of type legalization. This aspect of the target definition is beyond the planned scope of the implementation, and is thus solved through larger cost tables. The call to `getTypeLegalizationCost` of the default TLI always returns the cost of `1` and does no legalizations on the provided type.

Logging

To analyse the run-time behaviour of the optimizer and its TTI usage, a logging system was developed. It provides an insight into what TTI functions are being called, their arguments, return values and even the calling transformation pass. This can be enabled by building the TTI generator in a special mode, which makes it generate code that uses the `unwind` library to retrieve the stack trace and extract the calling transformation pass and prints it to a file along with the arguments and return value of the original code.

The TTI code generated for the Codasip uRISC processor is shown in Appendix B.

6.4 Compiling the TTI Library

In line with the integration design proposed in the previous chapter, the generated TTI code is compiled into a shared library and linked dynamically at the run-time of the Codasip compiler. That preserves the ability to ship the complete compiler binary and only compile the small amount of TTI code for a CodAL processor model.

C++ compiler implementations are allowed to change symbol names through a process called *name mangling*. Mangling may cause that the symbol representing the generated TTI constructor in the shared library to change based on the used compiler. To provide a stable interface, the shared TTI library exposes a single C function shown in Listing 6.8 that instantiates the class implementing the TTI interface.

```
extern "C" EXPORT_API
TargetTransformInfo TTIConstructor(const TargetMachine* TM, const Function& F) {
    return TargetTransformInfo(CodasipTTIImpl(TM, F));
}
```

Listing 6.8: Call to the TTI constructor wrapped in a C API.

The CMake build script producing the shared library uses the `add_library` function with the `SHARED` specifier. The list of compiled source files contains only the C interface from Listing 6.8 and the generated TTI implementation. Definitions of other LLVM functions called from the implementation are not included, as they are already present in the compiler binary executable. The dynamic linker will still be able to find their definitions, which enables the library itself to be very small. For the tested processor models, the TTI library has around 120 kB in size, depending on the model complexity and the number of instructions.

6.5 Codasip Studio Integration

Within Codasip Studio, users have the ability to generate the C/C++ compiler by clicking a button. Internally, two Python scripts are run: one that launches the back-end generator tool and the other to copy C++ headers provided with the Studio installation into the appropriate SDK directory. Several steps need to be carried out to generate and compile the TTI C++ sources into a shared library that can be loaded by the compiler.

Working Directory & Static Assets

First, a working directory is created in the file system. The directory is going to store all intermediate output from the library generation process. Next, the working directory is populated with static assets shipped as part of the Codasip Studio installation: the `CMakeLists.txt` file containing the build script and the `CodasipTTI.cpp` source file with the definition of the C function acting as the library interface.

Source Generator

In the following step, the integration script runs the TTI generator tool, which produces the `llvm_CodasipTTIImpl.cpp` and `llvm_CodasipTTIImpl.h` files that contain the TTI customizations and puts them in the working directory. A complete manual for the tool can be found in Appendix A.

Source Customization

Codasip Studio users should be able to customize the generated TTI implementation. This is possible by copying the generated files from the working directory into a directory in the model sources (`model/ia/optimizer/CodasipTTI`). These files will not be overwritten by the SDK build, unlike the volatile work directory. If the user did create some custom files, they are copied into the work directory, and the name of the original generated TTI source files will gain the `.generated` extension. This allows users to quickly compare the original code with their custom implementation.

Compilation & Library Naming

Finally, the source files present in the working directory are compiled via the CMake build tool with the same compiler and linker flags used for the rest of the SDK. The last step copies the resulting dynamic library into the `bin` SDK directory, where it can be automatically located by the customized Clang compiler. The final name of the library follows the naming scheme of other binaries: `{prefix}{processorModel}-{binaryName}.{suffix}`. In this particular case, `binaryName` is `CodasipTTI`, while `prefix` and `suffix` follow the platform naming scheme for shared libraries. (This means `lib` prefix and `.so` suffix for Unix systems, `.dll` for Windows systems.)

6.6 Using the TTI Library

The Clang compiler must load the compiled dynamic library, locate the `TTIConstructor` function, instantiate the TTI and provide the instance to the optimizer. In the compiler, the `TargetTransformInfo` class is always instantiated through the `TargetMachine` instance, which encompasses all aspects of the target architecture. This makes it a suitable place for loading the library.

Path Deduction

In the `TargetMachine` constructor, the shared library is loaded from the file system and dynamically linked to the compiler executable. The first task is to locate the library on the file system. The compiler gained a new command-line argument that allows developers to specify the absolute path pointing to the library located anywhere in the file system. However, by default, the Studio IDE invokes the compiler without explicitly defining the path, meaning the compiler needs to be able to deduce it on its own.

In the previous section, the last integration step copied the library alongside the compiler binary in the `bin` directory. This is deduced from the compiler executable path by replacing the tool name with the `CodasipTTI` library name and appropriate platform prefix and suffix. For working with the platform information, LLVM's `Triple` utility was used – it encodes the processor architecture, sub-architecture, vendor, operating system and ABI.

Library Usage

Working with the dynamic library is handled by LLVM's `sys::DynamicLibrary` module with multi-platform support. The library is loaded by the `LoadLibraryPermanently` function, and the exported function is located via the `SearchForAddressOfSymbol` function. The result is a `void` pointer that is treated as a function returning the `TargetTransformInfo` instance and taking a pointer to the `TargetMachine` instance and reference to a `Function`.

The function pointer is stored as an attribute of the `CodasipTargetMachine` class and used whenever a call to the `getTargetTransformInfo` method occurs.

6.7 Summary

In this chapter, the details of the implementation system were described. First, the current state of the cost model usage in the optimizer was analysed. Based on the analysis, I have developed a command-line tool for analysing important characteristics of CodAL processor models. The tool generates a C++ code that implements the cost model (i.e. the TTI interface) used by the LLVM optimizer. The code is compiled into a shared library, automatically loaded by a modified Clang compiler and used during the compilation process. All of the above was integrated into the Codasip Studio IDE.

Chapter 7

Testing Generated TTI

In this chapter, the implementation of the proposed system is tested. The criteria for evaluating the effectiveness of the optimizer with the new cost model are the number of clock cycles spent by the computation and the application code size. The number of spent clock cycles is determined by running an application in the CA-accurate simulator generated by Codasip Studio, while the code size is the number of bytes occupied by the resulting binary. The metrics were evaluated on tests from the previous chapter: the battery of basic programs, CoreMark and Dhrystone benchmarks. Processors used for the tests are Codasip uRISC and a processor from the Codix Berkelium family.

7.1 Codasip uRISC

Codasip uRISC is a custom 32-bit core with support for integer operations, including division, and basic vector processing capabilities. The processor supports a custom RISC ISA and has a 4-stage single-issue in-order pipeline. It is used for internal testing of new Codasip Studio features and in the hands-on examples in Codasip user manuals.

The benchmarks were run with the default LLVM cost model containing no architecture information at all, and with a cost model generated by the new tool and further hand-tuned to reach better results. The tweaked cost model assigns the cost of 4 to multiplication and the branch instruction instead of the default cost 1, and also enables interleaved vectorization with the factor of 2.

Figure 7.1 shows a graph comparing the program code size achieved when using the two cost model setups with the `-Os` compiler flag to optimize for code size. The Empty program containing only prologue and epiloque of the standard C library is 34.17% smaller with the new cost model. The rest of the basic programs show between 26.28% reduction and a 0.9% increase in code size. On average, the battery of basic programs sees a code size reduction of 13.63%.

Figure 7.2 depicts the impact of the new cost model on clock cycles. For most programs, the difference is lower than 1%. However, the string comparison benchmark consumed 36.16% more clock cycles when compiled with the new cost model, due to a missed loop unrolling opportunity. This was most likely caused by the conflation of instruction code size and latency into a single cost model. On the other side of the spectrum is the string search benchmark, where the new cost model achieved a 67.64% reduction in clock cycles. With the architecture information on various vector operations, the optimizer was able to decide that vectorization of two hot loops inside the algorithm will be beneficial. The measured values can be seen in their completeness in Table 7.1.

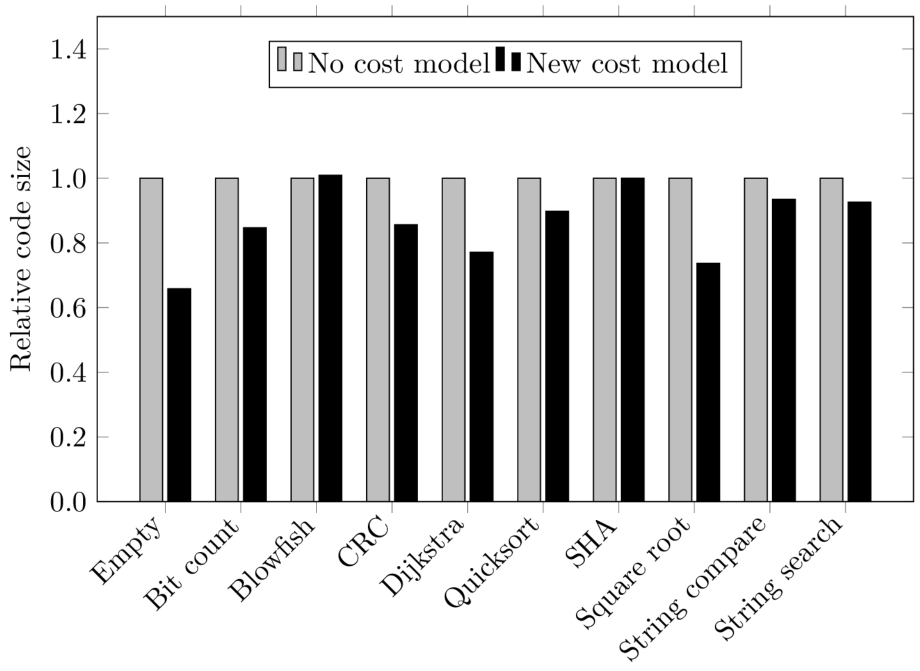


Figure 7.1: Relative code size when using no cost model and the new cost model on the Codasip uRISC processor.

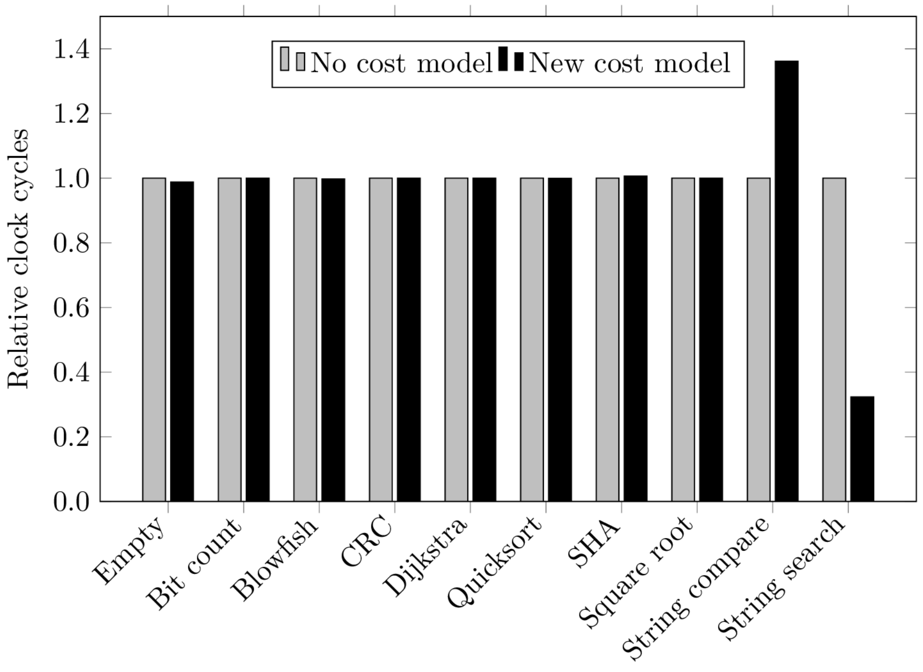


Figure 7.2: Relative clock cycles when using no cost model and the new cost model on the Codasip uRISC processor.

Program	Code size (`-Os`)			Clock cycles (`-O3`)		
	None	New	Diff %	None	New	Diff %
Empty	960	632	-34.17	84	83	-1.19
Bit count	2140	1812	-15.33	547979	547978	0.00
Blowfish	83124	83868	+0.90	17170785	17122599	-0.28
CRC	2280	1952	-14.39	3300109	3300108	0.00
Dijkstra	1432	1104	-22.91	922466	922465	0.00
Quicksort	1412	1268	-10.20	3184380	3182898	-0.05
SHA	2060	2060	0.00	7140	7140	0.00
Square root	1248	920	-26.28	717102	717101	0.00
String compare	5052	4724	-6.49	2713	3694	+36.16
String search	4004	3708	-7.39	1238969	400879	-67.64
CoreMark	8404	8372	-0.38	30833551	30648618	-0.60
Dhrystone	9052	9008	-0.49	758561	759091	+0.07

Table 7.1: Comparison of the improved cost model (new) and the default (none) on the Codasip uRISC processor.

To sum up the results, the optimizer was able to leverage the provided uRISC architecture information mainly when optimizing for size. The average code size reduction of 13.63% is a great result in the context of compiler optimizations, where even 1% improvements are considered a moderate success. As for performance, the loop vectorization optimization makes excellent use of the cost model, causing a 67.64% faster execution of the string search benchmark by employing the vector instructions. This seems to be in line with the hypothesis from the previous section, which argued that vectorization passes make the best use of the TTI interface.

7.2 Codix Berkelium

Codix Berkelium is a family of single-core processors with 5-stage single-issue in-order pipeline built on the open RISC-V architecture.[1] The Bk5-series cores come in 32-bit and 64-bit variants with optional support for multiple extensions. The tests were run on the 32-bit version (RV32I) of Bk5 supporting integer multiplication and division (RISC-V M extension) and 16-bit compressed instructions (RISC-V C extension).

The cost model in the generated TTI implementation was slightly modified. The cost of division operations was halved from 32 to 16. The original value was extracted from the instruction latency in the CodAL specification, which describes the worst case, but the division usually consumes fewer clock cycles depending on the divisor. As with the uRISC processor, the cost of a jump was set to 4. The cost of vector operations queried through the `getVectorInstrCost`, and `getShuffleCost` TTI functions was made extremely expensive, as the Berkelium processor does not support them. Surprisingly, enabling interleaving with factor 2 appeared to be beneficial during testing.

The comparison of code size achieved with the new hand-tuned cost model, and none cost model is shown in Figure 7.3. The greatest difference can once again be observed in the Empty program, where the code size decreased by 15.61%, while the average was an 8.08% decrease in code size.

[1]More information on the RISC-V ISA can be found at `https://riscv.org/`.

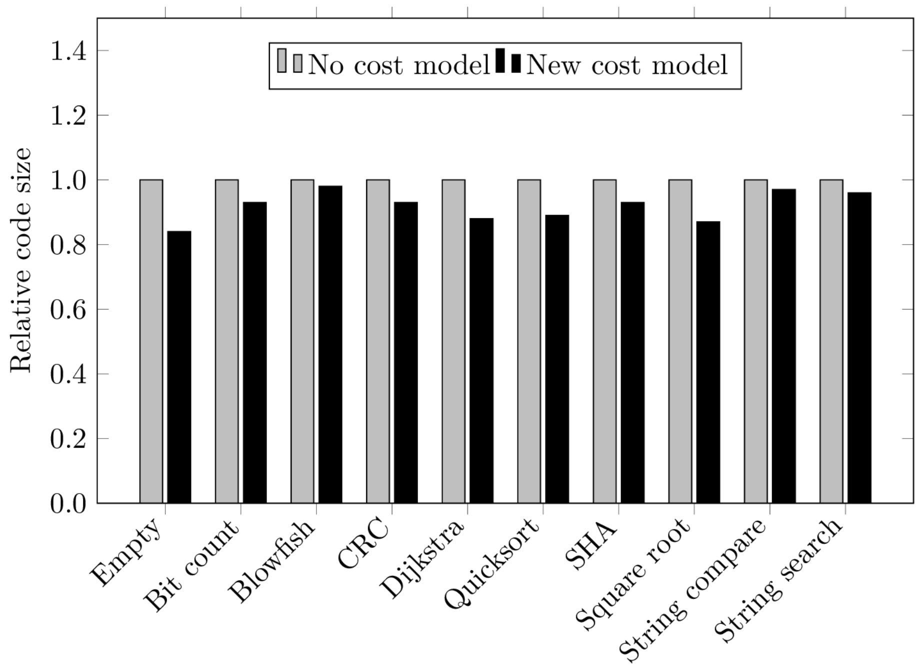


Figure 7.3: Relative code size when using no cost model and the new cost model on the 32-bit Codix Berkelium processor.

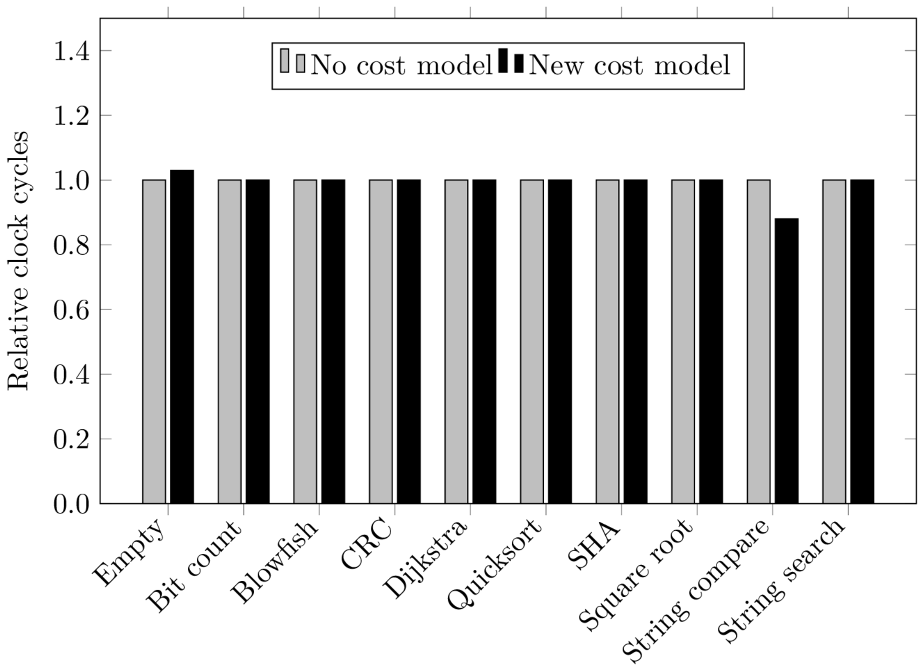


Figure 7.4: Relative clock cycles when using no cost model and the new cost model on the 32-bit Codix Berkelium processor.

Similar to the uRISC results, the majority of the clock cycle results in Figure 7.4 show a small change under 1% or none at all. A notable exception is the string comparison benchmark, which consumed 11.78% fewer clock cycles compared to the default LLVM cost model after unrolling two hot loops. The empty program required an additional 2.76% of clock cycles with the new cost model due to worse optimization decisions made in the exit procedure of *newlib*, the minimal C library linked by Codasip Studio.

Program	**Code size** (`-Os`)			**Clock cycles** (`-O3`)		
	None	**New**	**Diff %**	**None**	**New**	**Diff %**
Empty	948	800	-15.61	1194	1227	+2.76
Bit count	2076	1928	-7.13	395535	395566	+0.01
Blowfish	54724	53860	-1.58	14394695	14396499	+0.01
CRC	2172	2024	-6.81	3101786	3101824	0.00
Dijkstra	1260	1112	-11.75	661131	661245	+0.02
Quicksort	1376	1228	-10.76	1975863	1975853	0.00
SHA	2060	1912	-7.18	8475	8441	-0.40
Square root	1144	996	-12.94	612361	612361	0.00
String compare	4824	4676	-3.07	5287	4664	-11.78
String search	3780	3632	-3.92	771671	771615	-0.01
CoreMark	7008	7008	0.00	26426841	26472556	+0.17
Dhrystone	7224	7220	-0.06	441300	441981	+0.15

Table 7.2: Comparison of the improved cost model (new) and the default (none) on the 32-bit Codix Berkelium processor.

The complete results are presented in Table 7.2. Results of additional benchmarks tested on both Codasip uRISC and Codix Berkelium are located in Appendix C.

7.3 Summary

In conclusion, the testing suggests that the LLVM optimizer is able to leverage the architecture information primarily when optimizing for code size – tests on Codasip uRISC show an average improvement of 13.63% and 8.08% on Codix Berkelium. The code size reductions were most likely caused by the frequent usage of the TTI function responsible for reporting operation size, as observed in Chapter 6.

As for performance, the greatest improvement was seen with the architecture supporting vector operations – the string search benchmark was sped up by 67.64% on Codasip uRISC thanks to the auto-vectorization feature of the optimizer. This finding is in line with the fact that the TTI implementations of the most prominent targets analysed in previous chapter focus on customizing the cost of vector instructions. Overall, the cost model in LLVM's optimizer seems to benefit processors with more complex instruction sets which allow greater flexibility during code transformations.

After testing many versions of cost models, it has been observed that changes to code size are usually uniform and consistent between tested programs, whereas a change leading to a performance improvement in one benchmark usually causes a regression in another.

Chapter 8

Conclusion

This thesis focused on improving compiler optimizations through better cost modelling of custom processors. The importance of bespoke processors was explained along with the typical design workflow. Codasip Studio and the CodAL language were introduced as examples of tools automating the manual design process. The necessity of good optimizing compilers was argued, and the industry-standard LLVM compiler infrastructure was described, including details on its intermediate language, compilation pipeline and optimizations.

We have analysed the usage of the cost model in the LLVM optimizer and found that the most critical information is the number and size of registers, the cost of ordinary arithmetic, logic and comparison operations, and the support of vector operations. Based on these findings, a new system was proposed to automatically extract the architecture information from a CodAL processor model.

A new tool was developed to analyse the CodAL processor model, extract the information and transform it into a C++ code. The code implements the LLVM cost model interface (*target transform info*) and gets compiled into a small shared library. We have modified the Codasip compiler to automatically locate and load the library at run-time and use the cost model in the optimizer. The whole system was seamlessly integrated into Codasip Studio.

The implemented system was tested on two processors: Codasip uRISC with a SIMD support and Codix Berkelium 5 built on the 32-bit RISC-V architecture. We have further hand-tuned the generated cost model for each processor and measured the difference between the default LLVM cost model. We have seen an improvement in code size across all tested programs. On uRISC, the average reduction in code size was 13.63%, and for the Berkelium processor, 8.08%. The measured results are substantial, as even a 1% improvement is considered a good result in the context of compiler optimizations.

The performance of the new cost model was determined by running cycle-accurate simulations of both processors and measuring the number of spent clock cycles. In most testing programs, the new cost model did not have any effect. Notable exceptions are the 36.16% increase in clock cycles on uRISC due to failed loop unrolling, 67.64% reduction due to vectorization, and 11.78% reduction for Berkelium thanks to better unrolling decision.

Overall, the implemented systems works well, has a positive impact on code size, and it is going to be used in the industry when it gets shipped as a part of the next major release of Codasip Studio.

8.1 Future Work

There are a few aspects of the system that could be improved in the future. The fact that the created TTI generator tool works only with the instruction-accurate processor model limits the quality of the deduced information. If the tool had access to the cycle-accurate model, it could use the simulator to determine the latency of each instruction with different operands and produce more accurate cost model.

The target transform info interface within LLVM should be simplified and the documentation improved, as there are multiple functions seemingly serving the same purpose. Furthermore, the API should be explicit about the optimization goal, as the current implementations are often forced to conflate the latency, size and reciprocal throughput characteristics into a single number.

Another idea worth expanding on is the separation of LLVM target information into dynamically linkable libraries. During our work, the dynamic library approach allowed us to reduce the build time of the compiler for the end-users. Outside of Codasip, this approach could prove useful when distributing the compiler: the package maintainers would not have to choose which target architectures to support and link into a single binary. Instead, they could separately compile the base compiler, shared libraries of all targets and distribute them as distinct packages. The end-users then could choose which architecture they need to (cross-)compile for and download only the necessary targets, saving disk space and connection bandwidth.

Bibliography

[1] Bailey, B., Martin, G. and Anderson, T. *Taxonomies for the Development and Verification of Digital Systems*. 1st ed. Springer US, 2005. ISBN 9780387240213.

[2] Brandner, F., Ebner, D. and Krall, A. Compiler Generation from Structural Architecture Descriptions. In: *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. New York, NY, USA: Association for Computing Machinery, 2007, p. 13–22. ISBN 9781595938268.

[3] Brown, A. and Wilson, G. *The Architecture of Open Source Applications*. 1st ed. Lulu Press, 2012. ISBN 978-1257638017.

[4] *CodAL Language Reference Manual*. Codasip s.r.o., Nov 2019.

[5] *Codasip Compiler Generation Tutorial*. Codasip s.r.o., Nov 2019.

[6] *Codasip Instruction Semantics Language Manual*. Codasip s.r.o., Nov 2019.

[7] *Codasip Studio SDK User Guide*. Codasip s.r.o., Nov 2019.

[8] Dean, J. *The Deep Learning Revolution and Its Implications for Computer Architecture and Chip Design*. 2019.

[9] Hennessy, J. and Patterson, D. *Computer Architecture: A Quantitative Approach*. 6th ed. Elsevier Science, 2017. ISBN 978-0128119068.

[10] Ienne, P. and Leupers, R. *Customizable Embedded Processors: Design Technologies and Applications*. 1st ed. Elsevier Science, 2006. ISSN. ISBN 9780080490984.

[11] Lattner, C. and Adve, V. *The LLVM Instruction Set and Compilation Strategy*. Technical Report UIUCDCS-R-2002-2292. Computer Science Department, University of Illinois at Urbana-Champaign, Aug 2002.

[12] *Extending LLVM: Adding instructions, intrinsics, types, etc.* The LLVM Foundation, 2020. Accessed: Jan 2020. Available at: https://llvm.org/docs/ExtendingLLVM.html.

[13] *LLVM Language Reference Manual*. The LLVM Foundation, 2020. Accessed: Jan 2020. Available at: https://llvm.org/docs/LangRef.html.

[14] *The LLVM Target-Independent Code Generator*. The LLVM Foundation, 2020. Accessed: Jan 2020. Available at: https://llvm.org/docs/CodeGenerator.html.

[15] Mishra, P. and Dutt, N. *Processor Description Languages*. 1st ed. Elsevier Science, 2011. ISSN. ISBN 9780080558370.

[16] Nurmi, J. *Processor Design: System-on-Chip Computing for ASICs and FPGAs*. 1st ed. Springer, 2007. ISBN 978-1-4020-5529-4.

[17] Pedroni, V. A. *Circuit Design with VHDL*. Cambridge, MA, USA: MIT Press, 2004. ISBN 0262162245.

[18] Rosen, B., Wegman, M. and Zadeck, F. Global Value Numbers and Redundant Computations. In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. ACM, Jan 1988, p. 12–27. ISBN 0-89791-252-7.

[19] Rowen, C. *Engineering the Complex SOC: Fast, Flexible Design with Configurable Processors*. 1st ed. Pearson Education, 2004. ISBN 0-13-145537-0.

[20] Woerister, M. *Closing the gap: cross-language LTO between Rust and C/C++*. The LLVM Foundation, 2019. Accessed: Jan 2020. Available at: http://blog.llvm.org/2019/09/closing-gap-cross-language-lto-between.html.

Appendix A

TTI Generator Manual

The TTI generator is a command-line tool integrated in the Codasip Studio IDE. Its purpose is to analyse the processor model, extract architecture information useful for the LLVM cost model, and output a set of C++ source files implementing the LLVM TTI interface. The inputs are the XML file containing the CodAL model, and the output of the semantics extractor. The tool places a file named `llvm_CodasipTTIImpl.cpp` and corresponding header file into the specified output directory. The usage manual is shown in Listing A.1.

```
$ ttigen --help
Copyright (C) 2020 Codasip s.r.o.

Generate TargetTransformInfo implementation.

USAGE:

   -o <directory> -m <file> -s <file> [--] [--version] [-h]

Where:

   -o <directory>,  --output <directory>
     (required)  Output directory for TTI source files.

   -m <file>,  --model <file>
     (required)  Path to the ASIP XML model.

   -s <file>,  --semantics <file>
     (required)  Path to the instruction semantics file.

   --,  --ignore_rest
     Ignores the rest of the labeled arguments following this flag.

   --version
     Displays version information and exits.

   -h,  --help
     Displays usage information and exits.
```

Listing A.1: Usage of the new TTI generator tool.

Appendix B

Generated TTI Code

Listing B.1 shows the code produced by the TTI generator for the Codasip uRISC processor.

```
CodasipTTIImpl::CodasipTTIImpl(const TargetMachine *TM, const Function &F)
    : BaseT(TM, TM->getDataLayout()) {
  ST = TM->getSubtargetImpl(F);
  TLI = ST->getTargetLowering();
}
unsigned CodasipTTIImpl::getNumberOfRegisters(bool Vector) {
  return Vector ? 16 : 32;
}
unsigned CodasipTTIImpl::getRegisterBitWidth(bool Vector) const {
  return Vector ? 128 : 32;
}
unsigned CodasipTTIImpl::getArithmeticInstrCost(
    unsigned Opcode, Type *Ty, TargetTransformInfo::OperandValueKind Opd1Info,
    TargetTransformInfo::OperandValueKind Opd2Info,
    TargetTransformInfo::OperandValueProperties Opd1PropInfo,
    TargetTransformInfo::OperandValueProperties Opd2PropInfo,
    ArrayRef<const Value *> Args) {
  std::pair<int, MVT> LT = TLI->getTypeLegalizationCost(DL, Ty);
  int ISD = TLI->InstructionOpcodeToISD(Opcode);

  static const CostTblEntry CostTable[] = {
      {ISD::ADD, MVT::i32, 1},   {ISD::ADD, MVT::i16, 1},
      {ISD::ADD, MVT::i8, 1},    {ISD::ADD, MVT::i1, 1},
      {ISD::ADD, MVT::v4i32, 1}, {ISD::SUB, MVT::i32, 1},
      {ISD::SUB, MVT::i16, 1},   {ISD::SUB, MVT::i8, 1},
      {ISD::SUB, MVT::i1, 1},    {ISD::SUB, MVT::v4i32, 1},
      {ISD::MUL, MVT::i32, 1},   {ISD::MUL, MVT::i16, 1},
      {ISD::MUL, MVT::i8, 1},    {ISD::MUL, MVT::i1, 1},
      {ISD::MUL, MVT::v4i32, 1}, {ISD::AND, MVT::i32, 1},
      {ISD::AND, MVT::i16, 1},   {ISD::AND, MVT::i8, 1},
      {ISD::AND, MVT::i1, 1},    {ISD::AND, MVT::v4i32, 1},
      {ISD::OR, MVT::i32, 1},    {ISD::OR, MVT::i16, 1},
      {ISD::OR, MVT::i8, 1},     {ISD::OR, MVT::i1, 1},
      {ISD::OR, MVT::v4i32, 1},  {ISD::XOR, MVT::i32, 1},
      {ISD::XOR, MVT::i16, 1},   {ISD::XOR, MVT::i8, 1},
      {ISD::XOR, MVT::i1, 1},    {ISD::XOR, MVT::v4i32, 1},
      {ISD::SHL, MVT::i32, 1},   {ISD::SHL, MVT::i16, 1},
      {ISD::SHL, MVT::i8, 1},    {ISD::SHL, MVT::i1, 1},
      {ISD::SHL, MVT::v4i32, 1}, {ISD::SRA, MVT::i32, 1},
      {ISD::SRA, MVT::i16, 1},   {ISD::SRA, MVT::i8, 1},
      {ISD::SRA, MVT::i1, 1},    {ISD::SRA, MVT::v4i32, 1},
      {ISD::SRL, MVT::i32, 1},   {ISD::SRL, MVT::i16, 1},
      {ISD::SRL, MVT::i8, 1},    {ISD::SRL, MVT::i1, 1},
      {ISD::SRL, MVT::v4i32, 1},
  };
  if (const auto *Entry = CostTableLookup(CostTable, ISD, LT.second))
    return LT.first * Entry->Cost;

  return 4 * BaseT::getArithmeticInstrCost(Opcode, Ty, Opd1Info, Opd2Info,
                                           Opd1PropInfo, Opd2PropInfo, Args);
}
unsigned CodasipTTIImpl::getMemoryOpCost(unsigned Opcode, Type *Src,
                                         unsigned Alignment,
                                         unsigned AddressSpace,
                                         const Instruction *I) {
  std::pair<int, MVT> LT = TLI->getTypeLegalizationCost(DL, Src);
  int ISD = TLI->InstructionOpcodeToISD(Opcode);

  // Make vector reads/writes with unsupported alignment extremely expensive.
  if (Src->isVectorTy() && Alignment % DL.getABITypeAlignment(Src) != 0)
    return LT.first * 1000 * Src->getVectorNumElements();

  static const CostTblEntry CostTable[] = {
      {ISD::LOAD, MVT::i32, 2},   {ISD::LOAD, MVT::i16, 2},
      {ISD::LOAD, MVT::i8, 2},    {ISD::LOAD, MVT::i1, 2},
      {ISD::LOAD, MVT::v4i32, 4}, {ISD::STORE, MVT::i32, 1},
      {ISD::STORE, MVT::i16, 1},  {ISD::STORE, MVT::i8, 1},
      {ISD::STORE, MVT::i1, 1},   {ISD::STORE, MVT::v4i32, 4},
  };

  if (const auto *Entry = CostTableLookup(CostTable, ISD, LT.second))
    return LT.first * Entry->Cost;

  return 4 * BaseT::getMemoryOpCost(Opcode, Src, Alignment, AddressSpace, I);
}
unsigned CodasipTTIImpl::getCmpSelInstrCost(unsigned Opcode, Type *ValTy,
                                            Type *CondTy,
                                            const Instruction *I) {
  std::pair<int, MVT> LT = TLI->getTypeLegalizationCost(DL, ValTy);
  int ISD = TLI->InstructionOpcodeToISD(Opcode);

  static const CostTblEntry CostTable[] = {
      {ISD::SETCC, MVT::i32, 1},   {ISD::SETCC, MVT::i16, 1},
      {ISD::SETCC, MVT::i8, 1},    {ISD::SETCC, MVT::i1, 1},
      {ISD::SETCC, MVT::v4i32, 1}, {ISD::SELECT, MVT::i32, 1},
      {ISD::SELECT, MVT::i16, 1},  {ISD::SELECT, MVT::i8, 1},
      {ISD::SELECT, MVT::i1, 1},   {ISD::SELECT, MVT::v4i32, 1},
  };

  if (const auto *Entry = CostTableLookup(CostTable, ISD, LT.second))
    return LT.first * Entry->Cost;

  return 4 * BaseT::getCmpSelInstrCost(Opcode, ValTy, CondTy, I);
}
unsigned CodasipTTIImpl::getOperationCost(unsigned Opcode, Type *Ty,
                                          Type *OpTy) {
  if (llvm::EVT::getEVT(Ty, /*HandleUnknown=*/true) == llvm::MVT::Other)
    return BaseT::getOperationCost(Opcode, Ty, OpTy);

  if (Opcode == Instruction::Br)
    return 1;

  std::pair<int, MVT> LT = TLI->getTypeLegalizationCost(DL, Ty);
  int ISD = TLI->InstructionOpcodeToISD(Opcode);

  static const CostTblEntry CostTable[] = {
      {ISD::ADD, MVT::i32, 1},      {ISD::ADD, MVT::i16, 1},
      {ISD::ADD, MVT::i8, 1},       {ISD::ADD, MVT::i1, 1},
      {ISD::ADD, MVT::v4i32, 1},    {ISD::SUB, MVT::i32, 1},
      {ISD::SUB, MVT::i16, 1},      {ISD::SUB, MVT::i8, 1},
      {ISD::SUB, MVT::i1, 1},       {ISD::SUB, MVT::v4i32, 1},
      {ISD::MUL, MVT::i32, 1},      {ISD::MUL, MVT::i16, 1},
      {ISD::MUL, MVT::i8, 1},       {ISD::MUL, MVT::i1, 1},
      {ISD::MUL, MVT::v4i32, 1},    {ISD::AND, MVT::i32, 1},
      {ISD::AND, MVT::i16, 1},      {ISD::AND, MVT::i8, 1},
      {ISD::AND, MVT::i1, 1},       {ISD::AND, MVT::v4i32, 1},
      {ISD::OR, MVT::i32, 1},       {ISD::OR, MVT::i16, 1},
      {ISD::OR, MVT::i8, 1},        {ISD::OR, MVT::i1, 1},
      {ISD::OR, MVT::v4i32, 1},     {ISD::XOR, MVT::i32, 1},
      {ISD::XOR, MVT::i16, 1},      {ISD::XOR, MVT::i8, 1},
      {ISD::XOR, MVT::i1, 1},       {ISD::XOR, MVT::v4i32, 1},
      {ISD::SHL, MVT::i32, 1},      {ISD::SHL, MVT::i16, 1},
      {ISD::SHL, MVT::i8, 1},       {ISD::SHL, MVT::i1, 1},
      {ISD::SHL, MVT::v4i32, 1},    {ISD::SRA, MVT::i32, 1},
      {ISD::SRA, MVT::i16, 1},      {ISD::SRA, MVT::i8, 1},
      {ISD::SRA, MVT::i1, 1},       {ISD::SRA, MVT::v4i32, 1},
      {ISD::SRL, MVT::i32, 1},      {ISD::SRL, MVT::i16, 1},
      {ISD::SRL, MVT::i8, 1},       {ISD::SRL, MVT::i1, 1},
      {ISD::SRL, MVT::v4i32, 1},    {ISD::SETCC, MVT::i32, 1},
      {ISD::SETCC, MVT::i16, 1},    {ISD::SETCC, MVT::i8, 1},
      {ISD::SETCC, MVT::i1, 1},     {ISD::SETCC, MVT::v4i32, 1},
      {ISD::SELECT, MVT::i32, 1},   {ISD::SELECT, MVT::i16, 1},
      {ISD::SELECT, MVT::i8, 1},    {ISD::SELECT, MVT::i1, 1},
      {ISD::SELECT, MVT::v4i32, 1}, {ISD::LOAD, MVT::i32, 2},
      {ISD::LOAD, MVT::i16, 2},     {ISD::LOAD, MVT::i8, 2},
      {ISD::LOAD, MVT::i1, 2},      {ISD::LOAD, MVT::v4i32, 4},
      {ISD::LOAD, MVT::isVoid, 4},  {ISD::STORE, MVT::i32, 1},
      {ISD::STORE, MVT::i16, 1},    {ISD::STORE, MVT::i8, 1},
      {ISD::STORE, MVT::i1, 1},     {ISD::STORE, MVT::v4i32, 4},
      {ISD::STORE, MVT::isVoid, 4}, {ISD::TRUNCATE, MVT::i32, 0},
      {ISD::TRUNCATE, MVT::i16, 0}, {ISD::TRUNCATE, MVT::i8, 0},
      {ISD::TRUNCATE, MVT::i1, 0},
  };

  if (const auto *Entry = CostTableLookup(CostTable, ISD, LT.second))
    return LT.first * Entry->Cost;

  return 4 * BaseT::getOperationCost(Opcode, Ty, OpTy);
}
unsigned CodasipTTIImpl::getCastInstrCost(unsigned Opcode, Type *Dst, Type *Src,
                                          const Instruction *I) {
  return BaseT::getCastInstrCost(Opcode, Dst, Src, I);
}
bool CodasipTTIImpl::areInlineCompatible(const Function *Caller,
                                         const Function *Callee) const {
  const TargetMachine &TM = getTLI()->getTargetMachine();
  const FeatureBitset &CallerBits =
      TM.getSubtargetImpl(*Caller)->getFeatureBits();
  const FeatureBitset &CalleeBits =
      TM.getSubtargetImpl(*Callee)->getFeatureBits();
  return (CallerBits & CalleeBits) == CalleeBits;
}
unsigned CodasipTTIImpl::getInliningThresholdMultiplier() {
  return BaseT::getInliningThresholdMultiplier();
}
unsigned CodasipTTIImpl::getVectorInstrCost(unsigned Opcode, Type *Val,
                                            unsigned Index) {
  return BaseT::getVectorInstrCost(Opcode, Val, Index);
}
unsigned CodasipTTIImpl::getShuffleCost(TargetTransformInfo::ShuffleKind Kind,
                                        Type *Tp, int Index, Type *SubTp) {
  return BaseT::getShuffleCost(Kind, Tp, Index, SubTp);
}
bool CodasipTTIImpl::enableInterleavedAccessVectorization() {
  return BaseT::enableInterleavedAccessVectorization();
}
unsigned CodasipTTIImpl::getMaxInterleaveFactor(unsigned VF) {
  return BaseT::getMaxInterleaveFactor(VF);
}
```

Listing B.1: The TTI implementation generated for the Codasip uRISC processor.

Appendix C

Additional Cost Model Tests

Tables C.1 and C.2 show the test results for an additional set of programs. With the Codasip uRISC processor, we can see the same behaviour as in Chapter 7 – the code size dropped significantly across all test cases, whereas the performance changed only for some programs.

Program	**Code size** (`-Os`)			**Clock cycles** (`-O3`)		
	None	**New**	**Diff %**	**None**	**New**	**Diff %**
McGill/exptree	109408	108896	-0.47	17764	16756	-5.67
McGill/queens	121568	119552	-1.66	201687883	201685972	0.00
MiBench/stringsearch	20112	20128	0.08	2898307	967002	-66.64
Misc/lowercase	960	632	-34.17	84	83	-1.19
Misc/richards	7360	7004	-4.84	29622671	29622670	0.00
Misc/salsa20	2360	2268	-3.90	96231298	96231297	0.00
Shootout/fib2	1052	724	-31.18	84	83	-1.19
Shootout/lists	7796	7440	-4.57	98470346	98450345	-0.02
Shootout/methcall	7124	6768	-5.00	61000608	61000607	0.00
Shootout/nestedloop	960	632	-34.17	84	83	-1.19
Shootout/objinst	4644	4288	-7.67	566	567	0.18
Shootout/sieve	3640	3448	-5.27	56877605	56881354	0.01
Stanford/bubblesort	1364	1036	-24.05	132550216	132550215	0.00
Stanford/int-mm	2016	2100	4.17	11792168	12858075	9.04
Stanford/perm	1460	1132	-22.47	174937194	174937193	0.00
Stanford/puzzle	3200	3204	0.12	79724198	78344957	-1.73
Stanford/queens	1648	1320	-19.90	116496301	116496300	0.00
Stanford/quicksort	1560	1232	-21.03	108815712	108815711	0.00
Stanford/towers	1968	1640	-16.67	142436588	142460811	0.02
Trimaran/enc-md5	8708	8568	-1.61	83408855	83411067	0.00
Trimaran/enc-pc1	5820	5680	-2.41	17459439	17459438	0.00

Table C.1: Comparison of the improved cost model (new) and the default (none) on the Codasip uRISC processor.

The results for Codix Berkelium are in general worse with the new cost model. The degradation in performance and code size is most likely a manifestation of the issue discussed in the thesis: the conflation of multiple cost model dimensions into a single number. This also shows that a cost model tuned to give good results for one set of programs does not necesserily perform well on another set.

Program	**Code size** (`-Os`)			**Clock cycles** (`-O3`)		
	None	**New**	**Diff %**	**None**	**New**	**Diff %**
McGill/exptree	66704	67464	+1.14	17451	17177	-1.57
McGill/queens	74840	74728	-0.15	172128609	172128229	0.00
MiBench/stringsearch	15396	15604	+1.35	1826589	1494372	-18.19
Misc/lowercase	948	948	0.00	1194	1194	0.00
Misc/richards	5044	5044	0.00	21730518	21730518	0.00
Misc/salsa20	2120	2120	0.00	91460758	91460758	0.00
Shootout/fib2	1004	1004	0.00	1183	1183	0.00
Shootout/lists	5356	5356	0.00	74634889	74635079	0.00
Shootout/methcall	5016	5016	0.00	33669203	33669203	0.00
Shootout/nestedloop	948	948	0.00	1194	1194	0.00
Shootout/objinst	3296	3296	0.00	2412	2412	0.00
Shootout/sieve	2836	3044	+7.33	37209023	39855486	+7.11
Stanford/bubblesort	1268	1268	0.00	106422073	106422073	0.00
Stanford/int-mm	1380	1528	+10.72	3915182	6508691	+66.24
Stanford/perm	1268	1268	0.00	155760029	155760029	0.00
Stanford/puzzle	2456	2664	+8.47	69534542	64912523	-6.65
Stanford/queens	1392	1392	0.00	103794254	103794254	0.00
Stanford/quicksort	1384	1384	0.00	89392557	89392557	0.00
Stanford/towers	1588	1588	0.00	120733302	120729420	0.00
Trimaran/enc-md5	6900	6900	0.00	77148849	77148849	0.00
Trimaran/enc-pc1	4140	4140	0.00	17002800	17002775	0.00

Table C.2: Comparison of the improved cost model (new) and the default (none) on the 32-bit Codix Berkelium processor.