Wright State University
CORE Scholar

Browse all Theses and Dissertations | Theses and Dissertations

2019

LLVM-IR based Decompilation

Ilsoo Jeon
Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Repository Citation

Jeon, Ilsoo, "LLVM-IR based Decompilation" (2019). *Browse all Theses and Dissertations*. 2136.
https://corescholar.libraries.wright.edu/etd_all/2136

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.

LLVM-IR based Decompilation

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

by

Ilsoo Jeon
B.S.C.S., Wright State University, 2015
B.S., Wright State University, 2015

2019
Wright State University

Wright State University
GRADUATE SCHOOL

April 25, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Ilsoo Jeon ENTITLED LLVM-IR based Decompilation BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Meilin Liu, Ph.D.
Thesis Director

Mateen Rizki, Ph.D.
Chair, Department of Computer Science and Engineering

Committee on
Final Examination

Junjie Zhang, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Adam R. Bryant, Ph.D.

Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

Jeon, Ilsoo. M.S.C.E., Department of Computer Science and Engineering, Wright State University, 2019. *LLVM-IR based Decompilation.*

Decompilation is a process of transforming an executable program into a source-like high-level language code, which plays an important role in malware analysis, and vulnerability detection. In this thesis, we design and implement the middle end of a decompiler framework, focusing on Low Level Language properties reduction using the optimization techniques, propagation and elimination. An open-source software tool, *dagger*, is used to translate binary code to LLVM (Low Level Virtual Machine) Intermediate Representation code. We perform data flow analysis and control flow analysis on the LLVM format code to generate high-level code using a Functional Programming Langauge (FPL), Haskell. The result code generated by our decompiler framework is compared with the *sample* source code to verify the correctness of the decompiler framework.

Contents

1 Introduction **1**

2 Background **3**

2.1 Intermediate Representations 3

2.2 Data Flow Analysis 5

2.2.1 Static Single Assignment 7

2.3 Control Flow Analysis 7

2.4 Decompilation 8

3 Methodology **10**

3.1 Dagger and Preprocessing 11

3.2 Propagation 14

3.2.1 Idiom 14

3.2.2 Variable 23

3.3 Elimination 26

4 Evaluation **28**

4.1 Binary Translation and Preprocessing 29

4.2 Propagation 32

4.2.1 Idiom Detection 32

4.2.2 Variable Propagation 36

4.2.3 Double Precision and Naming Variable 37

4.3 Elimination 40

4.4 HLL Conversion and Program's Completeness 42

5 Case Study **45**

5.1 Single Basic Blocks 45

5.2 Conditional Statements 51

6 Conclusion **58**

Abbreviations **60**

Bibliography **61**

Appendix **65**

A Source Code of Algorithms **66**
A.1 Haskell Source code for Algorithm 1: BINARYOP 66
A.2 Haskell Source code for Algorithm 2: BITWISEOP 67
A.3 Haskell Source code for Algorithm 3: IDIOM 68
A.4 Haskell Source code for Algorithm 4: PROPAGATION 69
A.5 Haskell Source code for Algorithm 5: ELIMINATION 70

List of Figures

2.1 Example of DU and UD chains . . . 6
2.2 Comparison of Compiler and Decompiler . . . 8

3.1 Decompiler Process Diagram . . . 10
3.2 Example of Idiom Detection and Propagation . . . 14
3.3 Binary Idiom Propagation in LLIR . . . 15
3.4 Bitwise Idiom Propagation in LLIR . . . 18
3.5 Example of AND i16 $\%v_j, -2^M$ . . . 18
3.6 Example of Variable Propagation . . . 23
3.7 Example of Elimination . . . 26

5.1 Single Basic Block, Compound Assignment case . . . 46
5.2 Increment and Decrement . . . 46
5.3 Single Basic Block, Using Bracket case . . . 47
5.4 Numeric Variables Type Conversion from Short to Long . . . 48
5.5 Numeric Variables Type Conversion from Float to Double . . . 49
5.6 Char Type Variable 1 . . . 49
5.7 Char Type Variable 2 . . . 50
5.8 Char Type Variable 3 . . . 50
5.9 Conditional Statement, If Statement . . . 51
5.10 Conditional Statement, If and Else Statement with Equal condition . . . 52
5.11 Control Flow Graph of Figure 5.10 and its Block Order . . . 53
5.12 Conditional Statement, If and Else Statement with None Equal condition . . . 53
5.13 Conditional Statement, If, Else-If Statement . . . 54
5.14 Conditional Statement, Switch Statement . . . 55
5.15 Conditional Statement, If Else and Switch Statement . . . 56
5.16 Conditional Statement, If Else and Switch Statement . . . 57

List of Tables

2.1 HLL and LL Comparison 4

List of Code

4.1 Result of *dagger* binary translated, LLVM Low-Level IR Code 29
4.2 Result of PreProcessing initial Low-level Intermediate Representation (LLIR) 30
4.3 IR Code output Summary after *dagger* and PreProcessing 31
4.4 Example *sample* Register List 31
4.5 Example *sample* Variable List 31
4.6 LLVM Low-Level IR code before the Idiom Phase 33
4.7 IR Code after Idiom Conversion 34
4.8 Updated *sample* Variable List after the Idiom Detection 34
4.9 IR Code output Summary after Idiom Detection 35
4.10 IR Code after Propagation Phase 36
4.11 IR Code output Summary after the Variable Propagation 37
4.12 objdump Disassembled Code 38
4.13 objdump Disassembled .rodata section 38
4.14 IR Code after Naming Varaibles and Reading Double Precision 39
4.15 IR Code output Summary after the Variable and Percision Conversion . . . 39
4.16 IR Code after Elimination Phase before removing registers' statement . . . 40
4.17 IR Code after Elimination Phase without registers' DEF and USE statements 41
4.18 Overall Program Summary 42
4.19 High-level Langauge (HLL) Transformation Result 44
4.20 source code of *sample* 44

5.1 HLLC of Source 5.2 46
5.2 Original Code of HLLC 5.1 46
5.3 HLLC of Source 5.4 46
5.4 Original Code of HLLC 5.3 46
5.5 HLLC of Source 5.6 47
5.6 Original Code of HLLC 5.5 47
5.7 IR code of Figure 5.3 after Idiom Detection and Conversion 47
5.8 HLLC of Source 5.9 48
5.9 Original Code of HLLC 5.8 48
5.10 HLLC of Source 5.11 49
5.11 Original Code of HLLC 5.10 49
5.12 Change of variable type a in Figure 5.5's Idiom IR code 49
5.13 HLLC of Source 5.14 49
5.14 Original Code of HLLC 5.13 49
5.15 IR code after the Elimination Phase for Figure 5.6 49
5.16 HLLC of Source 5.17 50
5.17 Original Code of HLLC 5.16 50
5.18 HLLC of Source 5.19 50
5.19 Original Code of HLLC 5.18 50
5.20 HLLC of Source 5.21 51
5.21 Original Code of HLLC 5.20 51
5.22 HLLC of Source 5.23 52
5.23 Original Code of HLLC 5.22 52
5.24 HLLC of Source 5.25 53
5.25 Original Code of HLLC 5.24 53
5.26 HLLC of Source 5.27 54
5.27 code of HLLC 5.26 54
5.28 HLLC of Source 5.29 55
5.29 Original Code of HLLC 5.28 55
5.30 HLLC of Source 5.31 56
5.31 Original Code of HLLC 5.30 56
5.32 HLLC of Source 5.33 57
5.33 Code of HLLC 5.32 57
A.1 Source Code of Binary Idiom 66
A.2 Source Code of Handle Bitwise Idiom 67
A.3 Source Code of Detecting Idiom 68
A.4 Source Code of Propagation 69
A.5 Source Code of Elimination 70

Chapter 1

Introduction

A *decompiler* is a reverse engineering tool which transforms a machine code into a High-level Langauge (HLL) formatted code, which plays an important role in malware analysis. The goal of decompilation is to build an easily readable High-level Language Code (HLLC) without any modifications or changes in the program behaviour typically to support static analysis. It is most useful when the source code ofa program is unavilalbe, in the case of malware analysis.

A debugger and dissembler are great tools for dynamic and static analysis for reverse engineering; it shows both control flow and data flow of the program and gives an idea on the program's semantic and behaviour. On the other hand, using assembly code for static analysis requires knowledge in architectures and is expensive; a decompiler with accessible intermediate representation code is practical to save time for analysis.

Decompiler has been around since 1960s [29, p. 1]; almost every current decompiler supports 32-bit architectures, and some of them additionally cover 64-bit architectures as well. However, many open-source ones either do not support 64-bit, or the result of decompiler still contains low-level instruction(s) or registers which do not belong to high-level language code.

In this thesis, we attempt to develop the middle end of a decompiler framework, focus-

ing on Low-level Language (LL) properties reduction using the optimization techniques, propagation and elimination. Both optimization techniques are frequently used during the data flow analysis; propagation supports feasible code elimination which removes unnecessary statements or unreachable blocks. Thus, they remove LL statements and leave us with a IR code with HLL semantics.

Since our focus is design of middle end analysis of the decompiler framework, we use the open-source translator, dagger, to transform a binary code (or a machine code) to Intermediate Representation (IR) code, i.e., the LLVM (Low Level Virtual Machine) Intermediate Representation code. Then, to support static analysis and program verification with mathematical proof, we use Haskell to design the decompiler framework. We modify propagation and elimination techniques to work with the result LLVM intermediate representation code after binary translation. Finally, we edit control flow analysis technique to generate source-like high-level language code. The result code generated by our decompiler framework is compared with the *sample* source code to verify the correctness of the decompiler framework.

Additional information on decompiling process and intermediate representation are described in Chapter 2, including brief summary on data flow and control flow analysis. Our methodology, illustrated in Chapter 3, proposes the modified middle end including the optimization techniques, propagation and elimination, for decompilation; and then, we use a *sample* binary code, originally written in C languages, to evaluate the optimization techniques, propagation and elimination. The result code is compared with the *sample* source code, in Chapter 4. Finally in Chapter 6, we summarize our thesis work and discuss the limitation of our research and future work to improve it.

Chapter 2

Background

Source code is a code originally written by a programmer[1], typically using human-readable programming languages [26], such as C and Python; whereas, a code written for a electronic devices to run and execute is called *Machine Code (MC)*.

In this chapter, we present the basic concepts related to the techniques used in this thesis. In Section 2.1, intermediate representation languages are introduced, and major analysis will be performed in intermediate representation language to recover and generate source-like formatted code from MC. In the thesis, we choose to use *LLVM* (Low Level Virtual machine) intermediate representation. Two analysis methods data flow analysis and control flow analysis are introduced in Section 2.2 and Section 2.3. Then, decompilation process is described in Section 2.4.

2.1 Intermediate Representations

Intermediate Representation (IR), also known as intermediate language, is a representation used in compiler between High-level Langauge (HLL) (source programming language) and

[1]GNU and FSF argue that "Obfuscated 'source code' is not real source code and does not count as source code." [27]. However, in this paper, we will uses 'originally written code' definition by LINFO for convenient.

Low-level Language (LL) (target machine instruction language)). The purpose of IR is to make a compiler retargetable from one HLL to multiple architectures and to perform code analysis and optimization during compilation without changing program behaviour.

A single source code can result in different MC with the same behaviour, depending on hardware architectures, like Instruction Set Architecture (ISA) (see Table 2.1); while transforming HLL source code to a target MC, compilers use a universal language, IR, to support multiple LL.

Features	High-level Language (source)	Low-level Language (target)
Dependency	Language dependency	Machine Instruction dependency
Abstraction	High Abstraction (eg. type, variable, and semantic statements)	Low Abstraction (eg. registers, use of offset, and branch/return instructions)

Table 2.1: HLL and LL Comparison

Thus, IR needs to have some abstraction but is still able to be implemented in machine-like instructions; in the article *"The increasing significance of intermediate representations in compilers"* [6], Chow points out the following important features of intermediate languages: completeness, semantic gap, neutrality, programmable, extensible, simplicity, program information, and analysis information.

In the thesis, we choose to use *LLVM* (Low Level Virtual Machine) intermediate representation for multilingual supports [18] through active open projects [2]. Also, it has some features [6] that are suitable for our flow analysis and source-like code generation, illustrated as follows:

- LLVM IR has 31 opcode [19] (**simplicity**) and is able to write a code and modify it (**programmable**) [3]
- The representation uses low-level instruction set and memory models, but contains high-level abstraction, such as `switch`, except for *class, inheritance, and exception*

[2] *LLVM Project* http://llvm.org/ProjectsWithLLVM/, *Open Project* https://llvm.org/OpenProjects.html
[3] able to use command to convert or execute it, see https://llvm.org/docs/CommandGuide/

handling [19]. However, the syntax is not related to any specific hardware or source-languages (**Independency**)

- It uses general type-based system which is free from predefined fixed-sized definition in architecture and HLL. For example, *int* can refers to an integer with different bit-size depending on HLL and LL, where as LLVM uses *i#* format to indicate #-bit size of integer. This type system is helpful for pointer analysis and data transformation.
- Use of *Static Single Assignment Form* (Subsection 2.2.1) and *control flow graph* with *basic block* (Section 2.3) makes analysis easier to apply compiler optimization techniques, in Chapter 3

2.2 Data Flow Analysis

Data Flow Analysis (DFA) is a technique to observe variables' information, how they are defined and used in the program. Primary operation, in decompiler, is converting LLIR code into High-level Intermediate Representation (HLIR) code [9]; we use DFA data structure for our compiler optimization techniques. Here is a list of terms and definition of data structures that we use in this thesis:

- **Left Hand Side** and **Right Hand Side** indicate an expression on a left side and the one on a right side of an equation, typically, centered on equal sign ($=$ or $:=$).
- A variable v is **defined** when v is on the left hand side of a statement. However, in high-level language, v is defined when a memory-location is assigned to it. For instance, in C, 'int x;' is not an equation statement but a memory address is assigned to the variable x.
- **Definition** of a variable v is an expression on the right side of a statement, defining v. For instance, given $x = y + 5$, definition of x is $y + 5$, denoted as $\textbf{DEF}(x) = y + 5$.

- A variable v is **used** when v is either on the right side of a statement or appears on a non-equation statement (eg. `return v`). For example, given a statement $s : x = y+5$, y is used in s. It is denoted as **USE**(y) = [s] OR [$x = y + 5$] and **IN-USE**(s) = y.

- **Define-Use chain** is a list (or a table) storing a statement which defines a variable and a list of statement where the variable is used, denoted **DU**(v) = (**DEF**(v), **USE**(v)).

- **Use-Define chain** is a list (or a table) storing a statement using a variable and a list of a possible definition(s) of the variable, denoted **UD**(v) = (current statement using v, [**DEF**(v)]) or (current statement using v, [statement of **DEF**(v)]).

```
s1:    x = α
s2:    if(x > β)
s3:      then
s4:        x = 2
s5:        y = 1
s6:      else
s7:        y = 0
s8:    z = x + y
```

DU(x) = (α, [s_2, s_8]), (2, [s_8])
DU(y) = (1, [s_8]), (0, [s_8])
DU(z) = ($x + y$, [])

UD(x) = (s_2, α) or (s_2, s_1)
UD(x) = (s_8, [α, 2]) or (s_8, [s_1, s_4])
UD(y) = (s_8, [1, 0]) or (s_8, [s_5, s_7])

Figure 2.1: Example of DU and UD chains

- For $i, j \in \mathbb{N}, i < j$, let a variable v is defined at a statement s_i (denoted v_{s_i}). We say that **DEF**(v_{s_i}) is **killed** at a statement s_j which redefined the variable v.

- A variable v is stated as a **live** variable at a program point (or a statement) p, if v will be used in the future without being killed.

- A variable v is called, a **dead** variable, at a program point (or a statement) p, if v will not be used in the future.

2.2.1 Static Single Assignment

Static Single Assignment Form (SSA) is a intermediate representation form where every variables are defined only once; the variable is renamed when the variable's definition is reassigned. For instance, a variable x is not renamed the initial declaration, but any re-declaring x is renamed as x_1, x_2, x_3, etc. sequentially.

SSA makes program analysis more feasible [29, section 4.2-4.3] since there is no need to worry about killed variables and change of definitions. In our thesis, we use the static single assignment format in LLVM IR to track variables to modify and delete duplicated information in a IR code, using DU and UD chain.

2.3 Control Flow Analysis

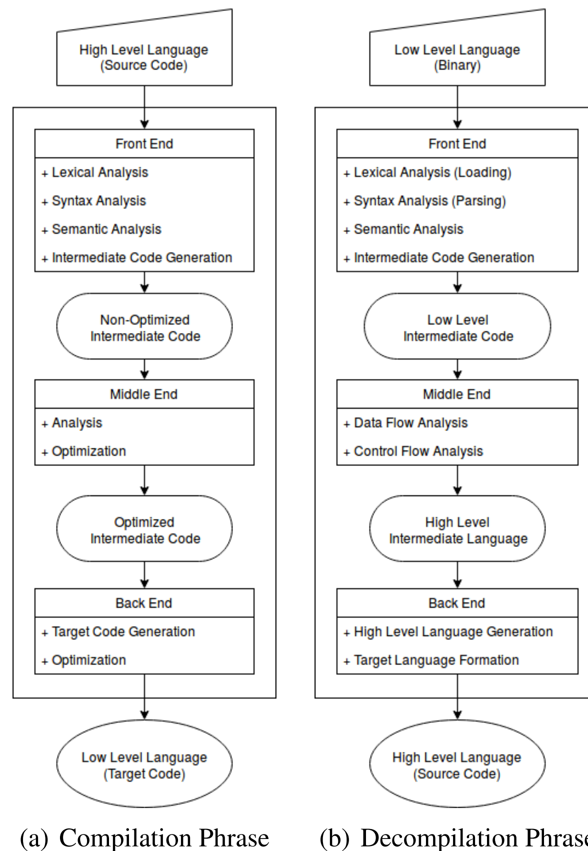
Control Flow Analysis (CFA) is another analysis technique, illustrating the program's running path. It is used to detect and define loop, conditional (or unconditional) statements, function(s), and functional call and return by construction Control Flow Graph (CFG). To draw a CFG, we starting from diving code into *basic blocks* based on *terminator* instructions.

Basic Block (BB) is a linear sequence of instructions (or statements) which runs straight (from top to bottom) without any branching interrupt; branching only occurs at the top of the block (from predecessor), called *entry point*, and at the end of block, called *exit point* to pass the program control to another basic block, successor. Every BB ends with either conditional (or unconditional) instruction, functional call and return, or indirect tail or pointer call [17, p. 111]; the last instruction of each basic block is called *terminator* instruction[4]. After the block of code is divided into basic blocks, CFA is assigned to find direct predecessor(s) and successor(s) of each blocks, depending on the terminator instructions. The CFG is provided to rearrange the IR code and generate the source-like HLLC.

[4] LLVM 6 IR Documentation URL: https://releases.llvm.org/6.0.1/docs/ReleaseNotes.html

2.4 Decompilation

A *compiler* is a program that translates source code (written in HLL) to a target language (Low-level Language Code (LLC)) for a processor to read and run. In contrast to a compiler, a program translating a LLC to a HLLC is called a *decompiler*. Since the decomiler reverses the compiling process, it has the same transforming steps as a compiler, as shown in Figure 2.2.



(a) Compilation Phrase (b) Decompilation Phrase

Figure 2.2: Comparison of Compiler and Decompiler

The first step of a decompiler is the *front-end* that performs code analysis on LLC, typically MC. The objective of this phase is constructing a CFG and generating a machine-independent intermediate representation code, called LLIR. LLC has a strong machine dependency such as operator instructions and data formats. Different architectures use different operator instructions and data formats than others; the front-end runs two linguistic

analysis to remove machine dependency from the code [8, Ch 4]: *syntax analysis*, and *semantic analysis*.

- ***syntax analysis***, also called *parsing*, follows the control flow and create a CFG; it generates LLIR and passes it to semantic analysis step.
- ***semantic analysis*** divides a block of code into basic blocks and does *idiom analysis*, *propagation* and *non-referred block elimination*.

After the linguistic anlaysis, the next phase is *middle-end* which handles DFA (Section 2.2) and CFA (Section 2.3). In a decompiler, the primary goal of DFA is transforming the output of front-end, LLIR, into a HLIR code for HLLC generation (see Figure 2.2(b)). Low-level concept symbols, flags and registers, need to be removed as follow [9]:

- Applying use/define chain analysis on the conditional branch instructions, **register flags** are removed and abstract expression (comparison symbols, like $<$, or $==$) statement is edited.
- Using *backward-flow* (or *bottom-top*) analysis, the content of a defined register is substituted for the register; it is called *forward substitution*.

Once the LLIR code has no more low-level symbols, CFA is applied to convert low-level instructions into high-level abstracted statements, HLL structures. It forms a BB and observes predecessors/ successors. Thus, `if-else, switch, for/while/repeat loops` are used instead of conditional/unconditional branching and multi-branching instructions; it results a HLIR.

Finally, a decompiler takes the HLIR and translates it into a C-like programming language code, in the *back-end* stage. After the code conversion, different decompilers might edit the format of the result HLL code, as an additional option [9]; formats includes renaming functions, variables, branching label, adding related libraries (e.g. <stdio.h> in C) statements, and more. In the end, the decompiler outputs a source-like HLLC which is more suitable to read for static analysis.

Chapter 3

Methodology

In this chapter, we first illustrate how to use the open-source program ***dagger*** [30] for binary translation and apply two common compiler optimization techniques, *propagation* and *elimination*, to convert binary code to HLLC. As introduced in Section 2.4, decompilation has three major passes (front-end, middle-end, and back-end). Section 3.1 introduces the front-end phase, and it runs the *dagger* program and preprocesses LLIR contents before code analysis.

Then, we illustrate the middle-end stage, which applies the optimization techniques, *propagation* and *elimination*, to generate HLIR code from LLIR (as shown in Figure 3.1). *Propagation* (as illustrated in Section 3.2), converts a single variable or set of instructions into another form. *Elimination* (explained in Section 3.3) removes any unnecessary statements such as dead-variables (or dead-statements).

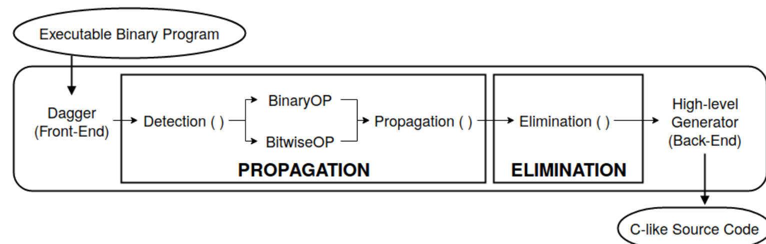


Figure 3.1: Decompiler Process Diagram

3.1 Dagger and Preprocessing

Dagger [24] is an open-source binary translator which transforms binary code to intermediate representation code. It is based on LLVM IR, and uses an *MC Layer* API (open project in LLVM), to convert a binary code to an LLIR code. In this project, we use **dagger** as our front-end phase to obtain the intermediate representation code (as illustrated in Section 2.4).

After applying *dagger* on the input binary code, the output LLIR code includes registers and functions from the following Executable and Linkable Format (ELF) attribute sections [15].

.plt (Procedure Linkage Table) It is an offset table for dynamically linked functions, including ones from a standard library.

.text Contains a list of the functions from user's executable code, including functions for setting up registers (pointers) and dynamic objects.

.init Runtime initial code block which runs before the user's executable code (e.g. *main* in C); it includes dynamic objects.

.fini Runtime terminal code block which runs after the user's executable code; it includes dynamic objects.

Description 3.1: Attribute functions in disassembled code

Although the attribute functions are critical to analyze and understand a program's behaviour, we only focus on handling non-attribution functions to reduce the size of LLIR content since they are not a part of typical source code. Based on the outputs of multiple test code segments, attribute functions are located after the @MAIN function in the *dagger* IR. We make an assumption that library functions -including static and dynamic linked libraries- and user-typed functions are listed before the defining @MAIN statement. Thus, the preprocessing method keeps the functions defined before the `define @main` line and discards the rest of the codes.

In addition to attribute functions, unlike the HLLC, IR contains the following formatted blocks at every function, starting at an address(#):

entry_fn_# First block which runs when the function is called. It initialises necessary register pointers for the function, such as %IP (instruction pointer) before the executable block(s). Also, called *entry* block in this paper.

exit_fn_# Last block runs right before the function exit to restore initial registers' value (before the entry initialisation). Also, called *exit* block in this paper.

bb_# (a functions initial block) Immediate successor of *entry_fn_#* block. It is a part of an executable code which holds high-level information, like idiom (Section 3.2.1). Also, called *initial* block in this paper.

bb_#$_n$ General block(s) which is a part of executable code.

Description 3.2: Different type of basic blocks in a function

Even though *entry_fn_#* and *exit_fn_#* are not directly part of a source code, they initialise and set up the register and pointer values, for the function. They are essential for pointer analysis and analysis on function call(s) and return(s)[1]; we do not delete them.

Furthermore, the output of dagger keeps track of the value of the registers, like a disassembled code (from *objdump* or IDA Free), and it tracks all 8-bit, 16-bit, 32-bit, and 64-bit registers by converting the type of variables (e.g. from 32-bit integer to 16-bit integer). Since 32-bit or 64-bit architectures are more common, we make assumption where 8-bit and 16-bit registers are not useful and unnecessary. Thus, our preprocessing method reads every line and eliminates any statement which defines or uses 8-bit or 16-bit registers; this also reduces the size of the workload.

Meanwhile, we split the IR-code into a chunk of functions and create two lists, *vList* and *pList*, with some categories (see List 3.3). The middle-end optimization program reads each function's IR-code and adds defined variables and its DEF information to the list; these tables function as a lookup table for variables during the code analysis.

[1]The project does not cover the pointer analysis and calling functions; however, they are for a future implementation.

- *vList* : List of defined variables (eg. $\underline{x}$ = **AND** i32 a, b)

$variable$	defined variable	x
$vtype$	type of variable	i32
$instruction$	instruction/operator of a statement	AND
$state$	DEF(variable)	AND i32 a, b

- *pList* : List of defined registers (eg. $\underline{esp_3}$ = **ADD** i32 esp_1, α)

$rname$	defined register variable	esp_3
$rbase$	1st operand, register variable before SSA format	esp_1
$ridx$	2nd operand, a numeric value (default = 0)	α
$rstate$	DEF(register)	ADD i32 esp_1, α

Description 3.3: Description for defined variable and registers lists

Since general variables are defined and named as unique numbers in `ssa` format (eg. %1, %2, %3, ...); we can parse each individual code line to store variables and registers in the list. However, we backtrack each register to find the base register in SSA format in LLVM-IR.

For example, registers are numbered after their base variable (eg. %RSP_0 is derived from the base %RSP register) in SSA form. %RSP_1 can hold the same value as %RSP (%RSP_0 := %RSP_1 := %RSP), or it can hold a different value (%RSP_1 := %RSP_0-8 where %RSP_0 := %RSP-4). The scanning process attempts to define the variable with the base register. So, in this case, %RSP_1 := %RSP-4-8 $\Rightarrow$:= %RSP-12, and this new information gets stored at $pList$ as a *rstate*.

In summary, binary transformation and the preprocessing output a modified LLIR, such that the LLIR has no 8-bits or 16-bits registers or attribute functions; also, it creates two lists, $vList$ and $pList$.

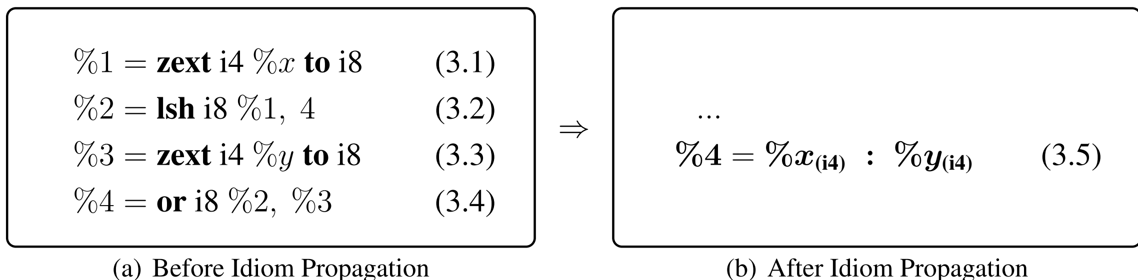
3.2 Propagation

Propagation is an optimization technique to substitute a variable with another, especially for redundant information (or variable). A declared variable is reusable or is able to be restated in HLLC, but MC needs to repeatedly load and store a variable from memory as necessary. Although IR is not MC, binary translation in the *front-end* does not remove machine properties. So, LLVM LLIR code contains machine-like instruction syntax, including `load` and `store` operators, and registers; *propagation* technique is used to transform remaining machine properties in IR code into HLL, especially in the following conditions: *idiom* and redundant variable.

3.2.1 Idiom

Binary code is a list of instructions (or operations) which form a program; it is hard to know the semantics of an individual instruction by itself. However, a sequence of instructions can give us more information about the program semantics and behaviour; these sequence of instructions are called *idioms*.

The code shown in Figure 3.2(a) is a LLIR code in LLVM format, converted from a binary program. There are four sequential instructions `zext`, `lsh`, `zext` and `or` in statement 3.1, 3.2, 3.3 and 3.4 respectively.



(a) Before Idiom Propagation

(b) After Idiom Propagation

Figure 3.2: Example of Idiom Detection and Propagation

Instruction `zext` converts variable type by adding zeros on the left side of the vari-

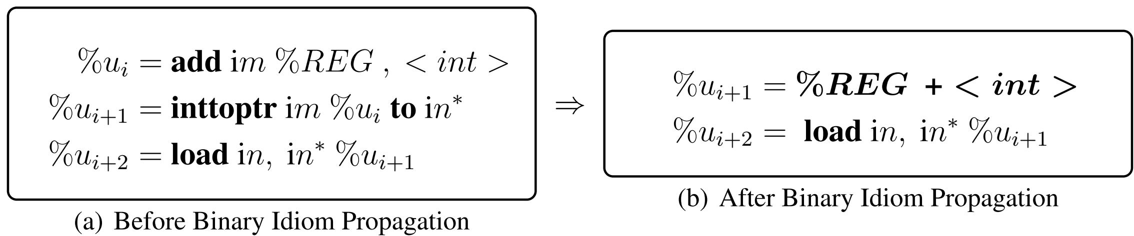
able, and lsh performs a logical left shift on $\%1$ four times. Statement (3.1) and (3.3) do not change the value, but statement (3.2) itself is an idiom and can be converted to $\%2 =$ **mul** i8 $\%x,\ 16$, without changing the program semantics or behaviour.

In addition, evaluating all four statements, we can tell $\%4$ is an 8-bit integer which has $\%x$ value in high 4-bit and $\%y$ value in low 4-bit. In statement 3.1, $\%1$ holds value 8-bit integer $0000 : \%x_{i4}$[2]. Then, lsh operator performs left shift on $\%1$, resulting $\%2 \leftarrow \%x_{i4} : 0000$. Next, third statement results $\%3 \leftarrow 0000 : \%y_{i4}$, similar to statement 3.1. Performing bit-wise *or* on $\%2$ and $\%3$, statement 3.4 is the same as saying ($\%4 \leftarrow x_{i4} : 0000$ **or** $0000 : y_{i4}$) $\Rightarrow$ ($\%4 \leftarrow x_{i4} : y_{i4}$).

Therefore, the sequence of four instructions [zext, lsh, zext, and or], from Figure 3.2(a), is an idiom which can be converted into a single statement with a *bitwise concatenation* colon (:). This type of conversion is called *idiom propagation* (see Figure 3.2).

Binary Idiom

In *dagger* LLIR, there are two common idioms, [add(sub), inttoptr, load(store)] and [zext, and, or]. First [add, inttoptr, load] idiom is called *Binary-idiom*; it is used to get a memory location to load data from the memory location into variable or to store a value in the memory location. In Figure 3.3, we present a generalized binary-idiom in LLIR form from the result of dagger.



(a) Before Binary Idiom Propagation

(b) After Binary Idiom Propagation

Figure 3.3: Binary Idiom Propagation in LLIR

NOTE: [] indicates other additional information in LLVM load and store form

---

[2] **:** is a Bitwise Concatenation Instruction

In this idiom, `add` instruction requires to have either a register pointer or a register variable ($\%REG$) (e.g. %eax or %rbp) and an index of an integer type. Since it has $\%REG$ and an index, we can convert the LLIR formatted statement into a HLL format to help generate HLL code (eg. $x = \texttt{add}\ a,\ b \mapsto x = a + b$).

Next, `inttoptr` instruction converts the register value type to a pointer for the next `load` statements. Although the type is changed, the value itself remains the same. We define Left Hand Side (LHS) variable $\%u_{i+1}$ as the HLL formatted `add` statement (see Figure 3.3(b)) and remove $\%u_i$ variable and its statement.

The optimization program is aware of the binary-idiom when it encounters a statement with an `add` instruction during the scanning process. Although `add` operator might be used in a different idiom or for general computation, the binary-idiom can be identified sicne one of the operand must be a register (eg. $\%REG$) to be a part of a binary-idiom, followed by an `inttoptr` operator statement. Therefore, when the `add` operator is found, we convert the LLIR statement into the HLL format (eg. `add` v, 5 $\mapsto$ v + 5), and then, we call BINARYOP function.

Algorithm 1, is a designed to performs idiom propagation when the sequence of statements is BINARYOP. BINARYOP, in Algorithm 1, is a sub-function which handles binary-idioms; it determines whether the `add` statement is a part of a binary-idiom, based on the operator next statement, $nextLine$. If the statement is a part of the idiom, the function performs idiom propagation, as shown in Figure 3.2(b).

In Algorithm 1, the BINARYOP function receives a LHS variable and a Right Hand Side (RHS) statemen of `add` statement, next-line statement, ad a list of variables, $vList$. First of all, it extracts LHS variable from the next-line, and called nv; it is equivalent to the $\%u_{i+1}$ variable from the binary-idiom example. Then, it checks, if the LHS variable and RHS definition from next-line fit the binary-idiom format (line 8 of Algorithm 1).

If the next-line fits the pattern of a binary-idiom, it removes the current LHS `add` variable from the list since it is defined for `inttoptr` statement, as a part of the idiom.

It writes a new line, called $addLine$, such that DEF(nv) is the `add` right-side definition (line 10 of Algorithm 1) and returns the new `add` line, along with an updated variable-list.

Algorithm 1 Binary operation with register pointer

1: **caller:** DETECTIDIOM (Algorithm 3)
2: **Input:**
  v := LHS variable of `add` statement
  $addHLL$:= converted `add` statement in $[ptr + k]$ HLL format for $k \in \mathbb{Z}$
  $nextLine$:= next line of `add` statement
  $vList$:= List of defined variable (generated during the preprocess, Section 3.1)
3: **Output:**
  $addLine$:= idiom-propagated DEF(u_{i+1}) OR unmodified DEF(u_i) `add` statement
  $nextLine$:= an empty string OR an unmodified `inttoptr` statement
  $vList$:= updated variable list

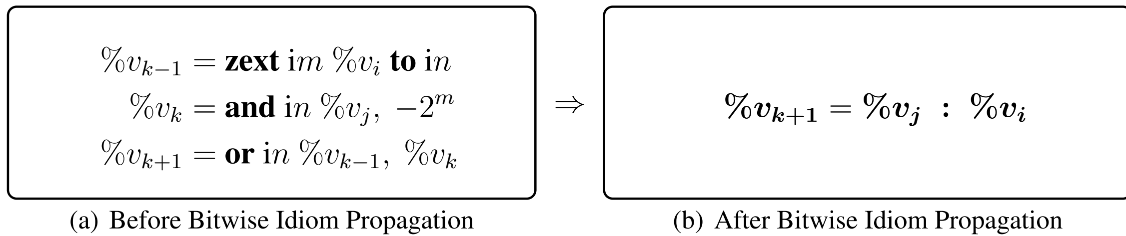
4: **function** BINARYOP(v, $addHLL$, $nextLine$, $vList$)
5: $v_{current} \leftarrow v$ data from $vList$
6: $nv \leftarrow nextLine$ LHS variable
7: $v_{next} \leftarrow nv$ data from $vList$

8: **if** nv not NULL & $nextLine$ operator is **`IntToPtr`** **then**
9:   **remove** $v_{current}$ from $vList$
10:   $addLine \leftarrow$ CONCAT(nv, "=", $addHLL$)
11:   **set** v_{next}'s instruction to Empty-String and state to be $addHLL$
12:   $vList \leftarrow$ **update** v_{next}
13:   $nextLine \leftarrow$ ""
14: **else**
15:   $addLine \leftarrow$ CONCAT(v, "=", $addHLL$)
16:   **set** $v_{current}$'s instruction to Empty-String and state to be $addHLL$
17:   $vList \leftarrow$ **update** $v_{current}$
18: **end if**
19: **return** ($addLine$, $nextLine$, $vList$)
20: **end function**

However, if the next-line is not a part of the idiom, it updates the definition of current variable v to be the input RHS statement (line 15) and returns the updated v statement, unmodified next-line statement, and the updated variable-list.

Bitwise Idiom

The second idiom, the [`zext, and, or`] idiom is a bit-wise operating idiom which is similar to the idiom example Figure 3.2. It is used to save data in extension registers `%xmm`, `%ymm`, or `%zmm` for later use; the set of instructions is typically followed by a `store` line. Below Figure 3.4 shows the general formatted structure of the *Bitwise-Idiom*.



(a) Before Bitwise Idiom Propagation (b) After Bitwise Idiom Propagation

Figure 3.4: Bitwise Idiom Propagation in LLIR

First `zext` instruction, in Figure 3.4, adds zeros in front of the v_i integer variable to extend the size of it. Suppose the size of the variable was m-bit, `zext` adds $(n - m)$ number of zeros to convert the value to a n-bit integer. We can use `colon` operator to re-write the statement as follow: $\%v_{k-1} = 0_{(n-m)} : \%v_i$.

Independent from the first `zext` statement, the second statement performs bitwise `and` operation in $\%v_j$ and -2^m. Any integer of -2^m is in 111...11000...0 format; `and` operator with -2^m zeros the last m-bit of variables.

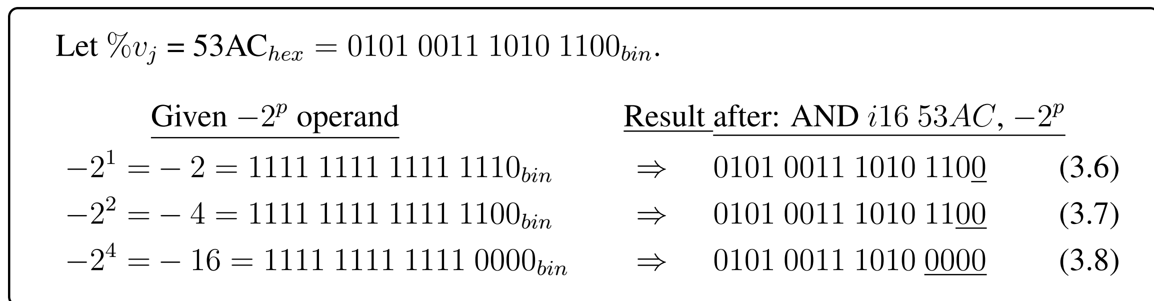


Figure 3.5: Example of AND i16 $\%v_j$, -2^M

In Figure 3.5, we compute 16-bit AND 0x53AC, -2^p, where p is set to be different values. For given $p = 1, 2, 4$, each result value consists of first (16-p)-bit value of 0x53AC and zeros for remaining lower p-bits. As a result, $\%v_j$.&. -2^m keeps the highest $(n-m)$-bits from $\%v_j$ and converts the low m-bits to 0s (denoted by $\%v_k = \%v_j : 0_m$).

The last instruction in the bitwise-idiom has an *or* operator, denoted by $v_{k-1}.|.v_k$. From previous statements, we know that the size of v_{k-1} and v_k are n-bit; we can say that $v_{k-1}.|.v_k \Rightarrow (0_{(n-m)} : v_i)$.|. $(v_j : 0_m)$ where the sizes of high $(n-m)$ and low m bits match one to another. Therefore, we can convert `or` instruction into `colon (:)` operator, denoted by $v_{k+1} = v_j : v_i$, using idiom propagation.

In contrast to the binary-idiom, BINARYOP, the optimization program alerts the bitwise-idiom when it finds an `and` operator statement with a negative integer operand which is power of 2. The `and` is not the first sequence of instructions in the idiom, but it has a noticeable property, -2^m operand; whereas, the first instruction `zext` does not have a special feature to indicate if it is a part of an idiom or not. So, the optimization program calls the BITWISEOP function when it sees the `and` operator with -2^m operand.

BTWISEOP function, described in Algorithm 2, determines whether the sequence of statements is a bitwise-idiom and converts the idiom to a single `colon` statement. Algorithm 2 takes three consecutive statements and the variable list (line 2) as the input.

In Algorithm 2, $cLine$ is an instruction statement with the `and` operator, and previous and next statement of `and` statement are denoted by $pLine$ and $nLine$. The BITWISEOP function extracts LHS variable of each statements and checks if previous and next statements have `zext` and `or` operator, respectively (line 8).

If the line sequence is not the [`zext`, `and`, `or`] idiom, it returns all the input statements and the unmodified list back to the caller. In contrast, if the statements fit the bitwise-idiom's pattern, the -2^m operand $op_{current}$ and an operand of `zext` in the previous statement op_{prev} are joined by the bitwise concatenation, as shown in line 10, Figure 3.4.

Algorithm 2 Bitwise operation with zeros

```
1:  caller: DETECTIDIOM
2:  Input:
        cLine := AND operator with op_current, (−2^m)
        pLine/nLine := previous / next line of cLine
        vList := List of defined variable (generated during the preprocess, Section 3.1)
3:  Output:
        [s1, s2, s3] := List of sequential statements (previous, current, and next lines)
        vList := either modified or unmodified variable list

4:  function BITWISEOP(pLine, cLine, nLine, vList)
5:      (pv, v, nv) ← LHS variables from pLine, cLine, and nLine
6:      if pv not NULL AND nv not NULL then
7:          (v_pre, v_current, v_next) ← pv, v, and nv variables data from the vList
8:          if ZEXT operator in pLine & OR in nLine then
9:              (op_current, op_pre) ← (operand in cLine, operand in pLine)
10:             state ← CONCAT(op_current, ":", op_pre)
11:             newLine ←CONCAT(nv, "=", state)

12:             set v_next's instruction to Empty-String and state to be state
13:             remove v_pre and v_current from the vList
14:             vList ← update v_next
15:             return ([ "", newLine, ""], vList)
16:         end if
17:     end if

18:     return ([pLine, cLine, nLine], vList)

19: end function
```

After creating the `colon` statement, the function writes a statement $newLine$ where the `colon` statement becomes a definition of the $nLine$'s LHS variable, nv. LHS variable of previous `zext` and current `and` statements are deleted from the variable-list since there is no more use of them; information on the nv is updated.

In the end, the function returns a list of three statements (an empty string, $newLine$ `colon` statement, and an empty string) and the updated variable-list; empty strings are added to keep the space for the list of three elements. The first two statements are meant for already-scanned statement, and the third statement is for next statement to scan; the illustration details are presented in Algorithm 3.

Idiom Detection

As illustrated in the previous sections, the optimization program detects idioms during the LLIR scanning process which is performed by the DETECTIDIOM function, as presented in Algorithm 3. The purpose of the function is to read the LLIR content, from Section 3.1, and call the corresponding sub-function, BINARYOP or BITWISEOP, for the pre-defined idiom conversion (Section 3.2).

Given a block of preprocessed IR-code and a variable list of the function, DETECTIDIOM reads the code line by line. Typical line fits one of the following format (memory address of an initial function or a basic block is denoted by $\#_{addr}$):

Define Function	`define void @fn_[`$\#_{addr}$`](%regset* ......)` {
Block Label	`entry_fn_`$\#_{addr}$`:` OR `exit_fn_`$\#_{addr}$`:` OR `bb_`$\#_{addr}$`:`
Statement	$var = [MachineInstrution]$ ... OR $[MachineInstruction]$ ... [3]
End of Function	}

In Algorithm 3, if a line declares a function, it parses `@fn_[`$\#_{addr}$`]` part from the line and sets it as a function name of current IR code (line 13 and 14). At line 16, we only care about lines with LHS variable since our idioms are associated with variable declaration and uses. Therefore, the function looks for line with LHS variable and an `add`, `sub`), or an `and` operator; then, it calls BINARYOP or BITWISEOP function. In the case of non-LHS, block labeling, or an end of function line, we do not do anything and read next line.

The DETECTIDIOM repeats the scanning and idiom-conversion process until there is no more line to read in the current function block's IR-code C_{ir}. Once the entire IR-code is read, it returns the parsed function name, idiom-converted IR-code, and the updated variable list.

[3]LLVM Instruction Manual: https://llvm.org/docs/LangRef.html

Algorithm 3 Detect idioms and Modify low-IR

```
 1: Input:
 2:     C_ir := IR file content of some function block (translated by dagger, in Section 3.1)
 3:     vList := List of defined variable                    (preprocessed, in Section 3.1)
 4: Output:
 5:     functionName := format: @fn_[initial address of the current function]
 6:     C_ir := Idiom-propagated / converted LLIR
 7:     vList := Modified variable list

 8: function DETECTIDIOM(C_ir, vList)
 9:     create variable functionName
10:     initialise n ← total line number of C_ir

11:     for i ← 0 ... n do
12:         line ← C_ir[i]
13:         if line is FunctionLine then
14:             functionName ← parse Function Name
15:         end if
16:         if line declares variable (LHS) then
17:             v ← LHS variable
18:             if line has ADD / SUB and Register operand (ptr) then
19:                 (ptr, idx) ← operands (op_1, op_2) from line
20:                 state ← Convert machine-like DEF(v) to a computational one
21:                 (C_ir[i], C_ir[i+1], vList) ← BINARYOP(v, state, C_ir[i+1], vList)

22:             end if
23:             if line has AND operator and −(2^bitsize) operand then
24:                 (tmp, vList) ← BITWISEOP(C_ir[i-1], line, C_ir[i+1], vList)
25:                 [C_ir[i-1], C_ir[i], C_ir[i+1]] ← tmp
26:             end if
27:         end if
28:     end for
29:     return (functionName, C_ir, vList)
30: end function
```

3.2.2 Variable

Variable Propagation is another propagation method to substitute a variable with another, especially to remove redundant information, such as repetitious memory accesses and type conversion.

In Figure 3.6(a), there is a variable x at the memory location %RSP-16. The variable x is used in division and summation; the data at the location %RSP-16 is loaded twice to two different variables (statement 3.9 and 3.13) since there is no concept of variables in Low-level Language. Thus, as marked in statement 3.20 in Figure 3.6(b), previous variable $\%v_n$ is able to be replaced with $\%1$ using propagation.

$$
\begin{aligned}
\%1 &= \text{load i64, i64* \%RSP-16} && (3.9)\\
\%2 &= \text{sext i32 2 to i64} && (3.10)\\
\%3 &= \text{sdiv i64 } \%1, \%2 && (3.11)\\
&\text{store i64 } \%3\text{, i64* \%RSP-16} && (3.12)\\
&\ldots\ldots\\
\%v_n &= \text{load i64, i64* \%RSP-16} && (3.13)\\
\%v_{n+1} &= \text{sext i32 3 to i64} && (3.14)\\
\%n_{n+2} &= \text{add i64 } \%v_n, \%v_{n+1} && (3.15)\\
&\text{store i64 } \%v_{n+2}\text{, i64* \%RSP-16} && (3.16)
\end{aligned}
$$

(a) Before Variable Propagation

$$
\begin{aligned}
&\ldots\ldots\\
\%2 &= \text{sext i32 } \mathbf{2} \text{ to i64} && (3.17)\\
\%3 &= \text{sdiv i64 } \%1, \underline{\mathbf{2}} && (3.18)\\
&\ldots\ldots\\
\%v_{n+1} &= \text{sext i32 } \mathbf{3} \text{ i64} && (3.19)\\
\%n_{n+2} &= \text{add i64 } \underline{\%\mathbf{1}}, \underline{\mathbf{3}} && (3.20)\\
&\text{store i64 } \%v_{n+2}\text{, i64* \%RSP-16} && (3.21)
\end{aligned}
$$

(b) After Varaible Propagation

Figure 3.6: Example of Variable Propagation

On the other hand, type declaration is essential in HLL to write a code; therefore, we need to keep the type of variables. However, most of type conversion instructions can be removed without changing the program behaviour, except for `bitcast` instruction between an integer and floating point number. In Figure 3.6(a), there is a 32-bit integer value 2 in statement 3.11. Converting it to 64-bit does not change the data value; we can ignore these type conversion statements. Variables $\%2$ and $\%v_{n+1}$ are substituted with a constant values 2 and 3, accordingly in statement 3.18 and 3.20 since they contain duplicated information.

In our optimization program, we design a PROPAGATION function to handle the variable propagation to reduce the duplicated information in LLIR-code. Algorithm 4 shows the propagation process in detail, and specifies the input and output of the algorithm.

Algorithm 4 Propagation

```
1:  Input:
2:      C_ir := IR-code of a function after idiom-propagation        (output of Algorithm 3)
3:      vList := List of defined variables in the function            (output of Algorithm 3)
4:      pList := List of defined registers in the function      (preprocessing, Section 3.1)
5:  Output:
6:      C_ir := variable propagated function IR-code
7:      vList, pList := Propagated and modified list of general and register variables

8:  function PROPAGATION(C_ir, vList, pList)
9:      initialise n ← total line number of C_ir
10:     for i = 0 ... n do
11:         line ← C_ir[i]
12:         if line has LHS variable then
13:             x ← LHS variable from the line
14:             if x is a register variable then
15:                 v_x, p_x ← x variable data from the vList and pList
16:                 if line has colon(:) then (eg. x = high : low)
17:                     rstate of p_x ← state of v_x
18:                     pList ← Update p_x info
19:                 end if
20:                 C_ir[i] ← CONCAT(x, =, p_x rstate)
21:                 replace variable from x with p_x rstate in C_ir[i+1]...[n]
22:                 update vList for any LHS variable change in C_ir[i+1]...[n]

23:             else(x is General Variable)
24:                 switch (line instruction type)
25:                     case conversion ⇒ x_new ← operand of line statement
26:                     case colon(:) ⇒ x_new ← low operand of line
27:                     default (other instruction types) ⇒ x_new ← x

28:                 replace variable from x with x_new in C_ir[i+1]...[n]
29:                 update vList for any LHS variable change in C_ir[i+1]...[n]
30:             end if
31:         end if
32:     end for
33:     return (C_ir, vList, pList)
34: end function
```

The main idea of the PROPAGATION function is to deal with the propagation of register variables (eg. %BP) and general variables separately as we have register list, $pList$, in addition to the variable list, $vList$, from the idiom propagation.

Again, propagation is associated with variables, the function only cares for the line statement with LHS variable; it reads the next line if the statement does not define a variable. However, for every line with LHS variable, the program checks if the LHS variable x is a register or a general variable (line 14 in Algorithm 4).

Assume x is a register, we get variable information from the $vList$ and the $pList$. If the definition of x in $vList$ is modified during the idiom propagation (Section 3.2.1) and contains `colon (:)`, the function edits the *rstate* information of x in $pList$ (List 3.3, Section 3.1); such that, it matches to the x information from $vList$ in line 16 to line 18.

Once x in $pList$ is updated, we change the current statement (line 20) which is built on the base register and index. Then, we substitute x with p_x $rstate$ in IR code; for instance, new line statement is %RSP_1 = %RSP-12, and all %RSP_1 is replaced with %RSP-12. In the end, all of the code lines use the base variable of the current x instead of SSA formatted registers.

Otherwise, if the LHS is a general variable, it checks if the statement has conversion instruction or colon. For type conversion operator, it takes its operand as a new variable v_{new} in line 25; whereas, it takes low-bit operand as a v_{new} in colon statement (line 26). Next, it replaces all LHS to new variable v_{new} in the IR and updates the variable list as needed.

The PROPAGATION function may change the IR content and read edited lines during the process; it runs until there is no more line statement to read. In the end, the function results in edited C_{ir} content and updated $vList$ and $pList$ which are more suitable for *elimination* state (Section 3.3), as there are defined variables that are no longer in use.

3.3 Elimination

Elimination technique is applied at the end of the *middle-end* phases of decompilation; it cleans up code and converts LLIR code to HLIR code. The propagated LLIR code has statements with useless variables (called *dead*) or variables with no declaration; we delete these *dead* variables and statements with undefined variables from the IR code. For instance, left side of Figure 3.7 is the output of the variable propagation from Section 3.2.

Suppose variables $\%2$, $\%v_n$, and $\%v_{n+1}$ are not used after the propagation, (see Figure 3.6). We cross out statements 3.30, 3.32, and 3.33, which define dead variables, $\%2$, $\%v_n$, and $\%v_{n+1}$, as they are longer needed.

$$\%1 = \text{load i64, i64* \%RSP-16} \tag{3.22}$$
$$\%2 = \text{sext i32 2 to i64} \tag{3.23}$$
$$\%3 = \text{sdiv i64 } \%1\text{, 2} \tag{3.24}$$
$$\dots\dots$$
$$\%v_n = \text{load i64, i64* \%RSP-16} \tag{3.25}$$
$$\%v_{n+1} = \text{sext i32 3 i64} \tag{3.26}$$
$$\%n_{n+2} = \text{add i64 } \%1\text{, 3} \tag{3.27}$$
$$\text{store i64 } \%v_{n+2}\text{, i64* \%RSP-16} \tag{3.28}$$

(a) Before Elimination

$$\%1 = \text{load i64, i64* \%RSP-16} \tag{3.29}$$
$$\cancel{\%2 = \text{sext i32 2 to i64}} \tag{3.30}$$
$$\%3 = \text{sdiv i64 } \%1\text{, 2} \tag{3.31}$$
$$\dots\dots$$
$$\cancel{\%v_n = \text{load i64, i64* \%RSP-16}} \tag{3.32}$$
$$\cancel{\%v_{n+1} = \text{sext i32 3 i64}} \tag{3.33}$$
$$\%n_{n+2} = \text{add i64 } \%1\text{, 3} \tag{3.34}$$
$$\text{store i64 } \%v_{n+2}\text{, i64* \%RSP-16} \tag{3.35}$$

(b) After Elimination

Figure 3.7: Example of Elimination

Algorithm 5 illustrates the elimination process. In Algorithm 5, we have a function called ELMINATION which takes similar arguments as PROPAGATION of Algorithm 4, and a Boolean typed variable, called $change$, to indicate if there are any changes made in the content during the scanning process.

In contrast to the propagation techniques, elimination handles both LHS and non-LHS lines; it removes lines with dead variable v (where $\|\text{USE}(v)\| = 0$, line 9). Also, it deletes lines with undefined variables u which have no DEF(u) but USE(u), at line 16 in

Algorithm 5; this way, we do not need to read the code several time to remove USE(x)-statement while we delete dead variable x, but only read the code line once.

Algorithm 5 Elimination

```
 1: pre-condition:
        C_ir := IR file content of some function block            (result of PROPAGATION)
        vList, pList := List of defined variables and registers   (result of PROPAGATION)

 2: function ELIMINATION(C_ir, vList, pList)

 3:     change ← FALSE                                                          (Boolean)
 4:     n ← Total Number of Lines in C_ir

 5:     for i =  0... n do
 6:         line ← C_ir[i]
 7:         if line is NOT a define-function, block-label, or end-of-function line then  text
 8:             op ←operand(s) of line statement
 9:             if line has LHS variable then
10:                 x ← LHS variable
11:                 if DEAD(x) then                                           USE(x) = 0
12:                     Find and Delete x from vList
13:                     change ← TRUE
14:                 end if
15:             end if
16:             if DEF(op) is NULL then                                     (if op ∉ vList)
17:                 change ← TRUE
18:             end if
19:         end if
20:     end for

21:     if change is FALSE then
22:         return (C_ir, vList)                             End the Elimination Phases
23:     else
24:         ELIMINATION(C_ir, vList, pList)
25:     end if
26: end function
```

Every time lines get deleted, *change* value is set to Boolean type $TRUE$ to indicate that there is change in the current reading process. Hence, the function stops when no change is made during the reading process till the end of contents (line 21) and outputs modified HLIR contents and the variable list.

Chapter 4

Evaluation

Our decompilation framework, focusing on the optimization techniques, propagation and elimination, is tested on *Ubuntu 18.04.1 (LTS) 64-bit* system and uses the following software packages: *LLVM 6.0.0*, *Clang 6.0.0*, *Haskell runghc/ghci 8.0.2*, and *Dagger LLVM 5.0.0 svn*. Based on LLVM IR and *MC Layer* API of LLVM open project, *dagger* is used to translate binary code to IR code. Our optimization techniques, propagation and elimination, are implemented in a Functional Programming Langauge, Haskell; it optimizes the LLVM format code and generates the High-level Intermediate Representation code.

We use a *sample* binary code, originally written in C language, to evaluate the functionality of propagation and elimination techniques, illustrated in in Chapter 3; the code is a single basic block without arguments, pointers, or functional call/return.

Following the processing order of decompilation, the output of binary translation and preprocessed code is displayed in Section 4.1. Section 4.2 presents the IR code after applying idiom detection (as illustrated in Algorithm 3) and the propagation techniques (as illustrated in Algorithm 4) to demonstrate the propagation results.

Then in Section 4.3, we show the result of elimination method from Section 3.3; we compare the output code from applying the elimination technique (as illustrated in Algorithm 5) to an Intermediate Representation code generated from the sample source code.

4.1 Binary Translation and Preprocessing

The front-end of decompilation framework takes binary code as input and translate it into LLVM-IR code, using *dagger*. As introduced in Section 3.1, the output Low-level Intermediate Representation (LLIR) code of *dagger* contains the source-relevant function and `@fn_400480`, and attribute functions (eg. `@main`, `@main_init_regset`, `...`), which will not be used in this optimization framework, see Result 4.1. In addition, there are lines which defines or uses 8 or 16-bits register (eg. line 11-13 and line 24).

```
1
2  define void @fn_400480(%regset* noalias nocapture) {
3    entry_fn_400480:
4    %RIP_ptr = getelementptr inbounds %regset, %regset* %0, i32 0, i32 14
5    %RIP_init = load i64, i64* %RIP_ptr
6    %RIP = alloca i64
7    store i64 %RIP_init, i64* %RIP
8    %EIP_init = trunc i64 %RIP_init to i32
9    %EIP = alloca i32
10   store i32 %EIP_init, i32* %EIP
11   %IP_init = trunc i64 %RIP_init to i16
12   %IP = alloca i16
13   store i16 %IP_init, i16* %IP
14   ...
15   br label %bb_400480
16
17   exit_fn_400480:                                   ; preds = %bb_400480
18   ...
19   ret void
20
21   bb_400480:                                        ; preds = %entry_fn_400480
22   %RIP_1 = add i64 4195456, 1
23   %EIP_0 = trunc i64 %RIP_1 to i32
24   %IP_0 = trunc i64 %RIP_1 to i16
25   ...
26   br label %exit_fn_400480
27 }
28
29 define i32 @main(i32, i8**) {......}
30
31 define void @main_init_regset(%regset*, i8*, i32, i32, i8**) {......}
32
33 define i32 @main_fini_regset(%regset*) {......}
34
35 define void @fn_4003D0(%regset* noalias nocapture) {
36   entry_fn_4003D0:
37
38   exit_fn_4003D0:          ; preds = %bb_4003EB, %bb_4003F8
39
40   bb_4003D0:               ; preds = %entry_fn_4003D0
41   ...
42   bb_4003F8:               ; preds = %bb_4003E1, %bb_4003D0
43 }
44 ...
```

Result 4.1: Result of *dagger* binary translated, LLVM Low-Level IR Code

Based on the framework's scope and assumption, we want to eliminate attribute functions and lower-bit register statements. Thus, we apply our preprocessing method on the initial LLIR, Result 4.1, and get following LLIR as shown in Result 4.2.

Our *sample* code does a simple calculation and does not call library or other external function(s); we expect our result to contain a single function with three basic blocks, function's *entry*, *exit*, and an *initial* block, see Description 3.2.

```
1
2   define void @fn_400480(%regset* noalias nocapture) {
3     entry_fn_400480:
4     %RIP_ptr = getelementptr inbounds %regset, %regset* %0, i32 0, i32 14
5     %RIP_init = load i64, i64* %RIP_ptr
6     %RIP = alloca i64
7     store i64 %RIP_init, i64* %RIP
8     %EIP_init = trunc i64 %RIP_init to i32
9     %EIP = alloca i32
10    store i32 %EIP_init, i32* %EIP
11    %IP_init = trunc i64 %RIP_init to i16
12    %IP = alloca i16
13    store i16 %IP_init, i16* %IP
14    ...
15    br label %bb_400480
16
17    exit_fn_400480:                               ; preds = %bb_400480
18    ...
19    ret void
20
21    bb_400480:                                    ; preds = %entry_fn_400480
22    %RIP_1 = add i64 4195456, 1
23    %EIP_0 = trunc i64 %RIP_1 to i32
24    %IP_0 = trunc i64 %RIP_1 to i16
25    ...
26    br label %exit_fn_400480
27  }
```

Result 4.2: Result of PreProcessing initial LLIR

Keeping source-related functions and 32/64-bits related statements, our result LLIR contains a single function, named `@fn_400480`, a possible source-code *main* function. Also, the `@fn_400480` has three basic blocks, *@entry_fn_400480*, *@exit_fn_400480:*, and *@bb_400480* (at line 3, 17, and 21), as we hoped.

The purpose of the preprocessing the LLIR is to reduce unnecessary low-level properties and the size of the code by eliminating irrelevant lines. In Result 4.3, initially translated *Dagger-IR* is a 124869 byte sized file with 3768 lines of code, which is much greater than the *objdump* disassembled code. Then, the preprocessing shrinks the size of the IR code

almost by 1/12. Based on our assumption, we only need source code-related functions and 32/64-bit relevant statements; our front-end meets the expectation.

```
"Assembly: 29040 bytes (665)"
"Dagger-IR: 124869 bytes (3768)"
"PREPROCESS: 9679 bytes (314)"
```

Result 4.3: IR Code output Summary after *dagger* and PreProcessing

In addition, preprocessing step creates lists of variables while we eliminate the 8/16-bit register statements. Result 4.4 and 4.5 are part of actual $pList$ and $vList$ after applying preprocessing on *sample* code.

```
-----------------------------------------
RegisterName (Base, Index): base+idx/RHS
-----------------------------------------
%RIP_ptr (%RIP_ptr, 0): getelementptr ...↩

%RIP_init (%RIP_ptr, 0): %RIP_ptr
%RIP (%RIP, 0): alloca i64
%EIP_init (%RIP_ptr, 0): %RIP_ptr
%EIP (%EIP, 0): alloca i32
  ...
%RIP_1 (, 4195457): 4195457
%EIP_0 (, 4195457): 4195457
  ...
%RSP_0 (%RSP, 0): %RSP
%RSP_1 (%RSP, -8): %RSP-8
%RIP_2 (, 4195460): 4195460
%RIP_3 (, 4195462): 4195462
%RIP_4 (, 4195470): 4195470
  ...
%ZMM0_1 (%ZMM0_1, 0) : or i512 %34, %35
%RIP_5 (, 4195478): 4195478
%EIP_4 (, 4195478): 4195478
%ZMM1_0 (%ZMM1, 0): %ZMM1
%XMM1_0 (%ZMM1, 0): %ZMM1
%XMM1_1 (%XMM1_1, 0): or i128 %41, %42
%YMM1_0 (%ZMM1, 0): %ZMM1
%YMM1_1 (%YMM1_1, 0): or i256 %44, %45
%ZMM1_1 (%ZMM1_1, 0): or i512 %47, %48
  ...
```

Result 4.4: Example *sample* Register List

```
-----------------------------------------
Variablename (type): base+idx / RHS
-----------------------------------------
%RIP_ptr (%RIP_ptr, 0): getelementptr ...
%RIP_init (%RIP_ptr, 0): %RIP_ptr
%RIP (%RIP, 0): alloca i64
  ...
%RIP_5 (i64): add i64 %RIP_4, 8
%EIP_4 (i32): trunc i64 %RIP_5 to i32
%36 (i64): add i64 %RIP_5, 202
%37 (double*):inttoptr i64 %36 to double*
%38 (double): load double, double* %37
%39 (i64): bitcast double %38 to i64
%ZMM1_0 (<16 x float>): load
          <16 x float>, <16 x float>* %ZMM1
%40 (i512): bitcast <16 x float> %ZMM1_0
                                    to i512
%XMM1_0 (i128): trunc i512 %40 to i128
%41 (i128): zext i64 %39 to i128
%42 (i128): and i128 %XMM1_0,
                      -18446744073709551616
%XMM1_1 (i128): or i128 %41, %42
  ...
%53 (i64): trunc i128 %XMM1_1 to i64
%54 (double): bitcast i64 %53 to double
%55 (i64): add i64 %RSP_1, -16
%56 (double*):inttoptr i64 %55 to double*

  ...
```

Result 4.5: Example *sample* Variable List

The registers are defined as a variable; $vList$ contains the information on defined registers, as well as general variables, even though $pList$ already has registers information. Few registers' information might be same and duplicated, but most of the information

does not overlap because $vList$ initially holds the Right Hand Side (RHS) of variables without backtracking and calculating the base register, R_{base}. Thus, $pList$ holds simplified information on registers, and we only need to modify and remove variables from the $vList$ in the later analysis.

The front-end of our decompilation framework takes binary code as an input and translates it into LLVM-IR code, using *dagger*. Then, it modifies the IR code, such that it has no duplicated or irrelevant information to obtain High-level Intermediate Representation (HLIR). Also, it creates two variable lists, $pList$ for defined registers and $vList$ for any defined variables in the IR, during the preprocess phase to help IR analysis, as illustrated in Section 3.2 and 3.3.

4.2 Propagation

The first optimization technique in our middle-end of decompilation process is propagating variables (Section 2.4) in the LLIR code, generated from the Section 4.1. Propagation technique substitutes a variable with another to decrement the redundancy of code. We present the reulst of Idiom detection (illustrated in Algorithm 3) in the Subsection 4.2.1. Then, ew present the result of Variable propagation in Subsection 4.2.2.

4.2.1 Idiom Detection

Idiom detection is used to further reduce the IR code by converting a consecutive LL-like statements into HLL-like statements, typically shorter and more understandable than the one before the conversion. It takes preprocessed LLIR and $vList$ as an input and make changes in them.

Result 4.6 is a part of the preprocessed LLIR contents from the previous Section 4.1; it has following idioms:

Binary Idiom : [`add(sub), inttoptr, load(store)`] instructions at line 17-19

Bitwise Idiom : [`zext, and, or`] instructions at line 24-26, 29-31, and 33-35

```
1   define void @fn_400480(%regset* noalias nocapture) {
2     entry_fn_400480:
3     ...
4     exit_fn_400480: ; preds = %bb_400480
5     ...
6     bb_400480:
7     %RIP_1 = add i64 4195456, 1
8     ...
9     %RSP_0 = load i64, i64* %RSP
10    %RSP_1 = sub i64 %RSP_0, 8
11    ...
12    %RIP_2 = add i64 %RIP_1, 3
13    %RIP_3 = add i64 %RIP_2, 2
14    %RIP_4 = add i64 %RIP_3, 8
15    %RIP_5 = add i64 %RIP_4, 8
16    ...
17    %36 = add i64 %RIP_5, 202
18    %37 = inttoptr i64 %36 to double*
19    %38 = load double, double* %37, align 1
20    %39 = bitcast double %38 to i64
21    %ZMM1_0 = load <16 x float>, <16 x float>* %ZMM1
22    %40 = bitcast <16 x float> %ZMM1_0 to i512
23    %XMM1_0 = trunc i512 %40 to i128
24    %41 = zext i64 %39 to i128
25    %42 = and i128 %XMM1_0, -18446744073709551616
26    %XMM1_1 = or i128 %41, %42
27    %43 = bitcast <16 x float> %ZMM1_0 to i512
28    %YMM1_0 = trunc i512 %43 to i256
29    %44 = zext i128 %XMM1_1 to i256
30    %45 = and i256 %YMM1_0, -340282366920938463463374607431768211456
31    %YMM1_1 = or i256 %44, %45
32    %46 = bitcast <16 x float> %ZMM1_0 to i512
33    %47 = zext i128 %XMM1_1 to i512
34    %48 = and i512 %46, -340282366920938463463374607431768211456
35    %ZMM1_1 = or i512 %47, %48
36    ...
37    %53 = trunc i128 %XMM1_1 to i64
38    %54 = bitcast i64 %53 to double
39    %55 = add i64 %RSP_1, -16
40    %56 = inttoptr i64 %55 to double*
41    store double %54, double* %56, align 1
42    ...
43  }
```

Result 4.6: LLVM Low-Level IR code before the Idiom Phase

After applying idiom detection, Algorithm 3, we obtain the IR code, shown in Result 4.7. Suppose the variable %36 and %37 are used only once to load a data to %38, at line 19; we expect the line 18, to be rewritten in HLLC format.

- Form %37 = inttoptr i64 %36 to double* to %37 = %RIP_5 + 202

Also, the desired results of bitwise idioms are conversion of `zext, and, or` instruction sequences into a single `colon (:)` instruction (see Section 3.2.1). For in-

stance, Algorithm 3 should convert the first bitwise idiom, at line 24-26, to a new statement %XMM1_1=%XMM1_0:%39.

```
1  define void @fn_400480(%regset* noalias nocapture) {
2    entry_fn_400480:
3      ...
4    exit_fn_400480: ; preds = %bb_400480
5      ...
6    bb_400480:
7      %RIP_1 = add i64 4195456, 1
8
9      %RSP_0 = load i64, i64* %RSP
10     %RSP_1 = sub i64 %RSP_0, 8
11     ...
12     %RIP_2 = add i64 %RIP_1, 3
13     %RIP_3 = add i64 %RIP_2, 2
14     %RIP_4 = add i64 %RIP_3, 8
15     %RIP_5 = add i64 %RIP_4, 8
16     ...
17     %37 = %RIP_5+202
18     %38 = load double, double* %37, align 1
19     %39 = bitcast double %38 to i64
20     %ZMM1_0 = %ZMM1
21     %40 = bitcast <16 x float> %ZMM1_0 to i512
22     %XMM1_0 = %ZMM1
23     %XMM1_1 = %XMM1_0 : %39
24     %43 = bitcast <16 x float> %ZMM1_0 to i512
25     %YMM1_0 = %ZMM1
26     %YMM1_1 = %YMM1_0 : %XMM1_1
27     %46 = bitcast <16 x float> %ZMM1_0 to i512
28     %ZMM1_1 = %46 : %XMM1_1
29     ...
30     %53 = trunc i128 %XMM1_1 to i64
31     %54 = bitcast i64 %53 to double
32     %55 = add i64 %RSP_1, -16
33     %56 = inttoptr i64 %55 to double*
34     store double %54, double* %56, align 1
35   ...
```

Result 4.7: IR Code after Idiom Conversion

Line 17 and 18 are the output of the idiom detection (Algorithm 3). By calling the BINARYOP function, from Algorithm 1; variable %37 is redefined in HLL format. IDIOM DETECTION function, from Algorithm 3, calls BITWISEOP function (Algorithm 2) when it detects a bitwise idiom and redefined SSA formatted %XMM, %YMM, and %ZMM register variables, as in line 23, 26, and 28.

In addition, Algorithm 3 also modifies the $vList$, generated during the preprocessing phase. As the algorithm redefines variables in IR, it updates the variables' RHS information to be match with the code, see Result 4.8.

```
------------------------------------------------------------------------------------
Variablename (type) <- base+idx / RHS   ; previous Variable's information
------------------------------------------------------------------------------------
...
%37 (double*) <- %RIP_5+202           ; %37 (double*) <- inttoptr i64 %36 to double*
...
%XMM1_1 (i128) <- %XMM1_0 : %39        ; %XMM1_1 (i128) <- or i128 %41, %42
%YMM1_1 (i256) <- %YMM1_0 : %XMM1_1    ; %YMM1_1 (i256) <- or i256 %44, %45
%ZMM1_1 (i512) <- %46 : %XMM1_1        ; %ZMM1_1 (i512) <- or i512 %47, %48
...
```

Result 4.8: Updated *sample* Variable List after the Idiom Detection

note: previous RHS value is shown on the right side of the new RHS value as a comment (;)

After the idiom detection phase, we obtain the IR without the idioms and with modified $vList$. Also, the decompilation framework prints out its working summary, see Result 4.9. The numbers `12` and `15` indicates the number of corresponding idioms the framework found. So we know there are twelves binary idioms and fifteen bitwise idioms in the code; these information will be used in later Section 4.4 to judge the program's completeness and correctness of the *sample* program.

```
----------------------
@fn_400480
----------------------

Detected Idioms
- Idiom 1 (binaryOP): 12
- Idiom 2 (bitwiseOP): 15

"IDIOM: 7826 bytes (272)"
```

Result 4.9: IR Code output Summary after Idiom Detection

4.2.2 Variable Propagation

Variable propagation optimization technique substitutes certain variables or information in the IR, after all the idioms are removed. It replaces registers' definition from SSA format with a R_{base}+`index` format or an indicating integer, using $pList$ which is generated during the preprocessing phase. Also, it redefines a non-register variable's RHS, as shown in line 24 in Algorithm 4. Then, the variable propagation algorithm replaces the variable (either a non-register or a register) with its updated information.

Result 4.10 shows the IR code after the varaible propagation, in which all the insturction pointers `%RIP` are reassigned with an integer, from $pList$. The SSA variables are redefined by R_{base} (line 7 and 8), and the register variables are replaced by their R_{base} status (eg. line 22, Algorithm 4).

```
1  define void @fn_400480(%regset* noalias nocapture) {
2    entry_fn_400480: ...
3    exit_fn_400480: ...; preds = %bb_400480
4    bb_400480:
5      %RIP_1 = 4195457  ; := add i64 4195456, 1
6      ...
7      %RSP_0 = %RSP     ; := load i64, i64* %RSP
8      %RSP_1 = %RSP-8   ; := sub i64 %RSP_0, 8
9      ...
10     %RIP_2 = 4195460  ; := add i64 %RIP_1, 3
11     %RIP_3 = 4195462  ; := add i64 %RIP_2, 2
12     %RIP_4 = 4195470  ; := add i64 %RIP_3, 8
13     %RIP_5 = 4195478  ; := add i64 %RIP_4, 8
14     ...
15     %37 = 4195680     ; := %RIP_5+202
16     %38 = load double, double* 4195680, align 1 ; %37 := 4195680
17     %39 = %38         ; := bitcast double %38 to i64
18     %ZMM1_0 = %ZMM1
19     %40 = %ZMM1       ; := bitcast <16 x float> %ZMM1_0 to i512
20     %XMM1_0 = %ZMM1
21     %XMM1_1 = %38     ; := %XMM1_0 : %39
22     %43 = %ZMM1       ; := bitcast <16 x float> %ZMM1_0 to i512
23     %YMM1_0 = %ZMM1
24     %YMM1_1 = %38     ; := %YMM1_0 : %XMM1_1
25     %46 = %ZMM1       ; := bitcast <16 x float> %ZMM1_0 to i512
26     %ZMM1_1 = %38     ; := %46 : %XMM1_1
27     ...
28     %53 = %38         ; := trunc i128 %XMM1_1 to i64
29     %54 = %38         ; := bitcast i64 %53 to double
30     %56 = %RSP-24     ; := %RSP_1-16
31     store double %38, double* %RSP-24, align 1
32                                              ; store double %54, double* %56, align 1
33   ...
```

Result 4.10: IR Code after Propagation Phase

note: previous statement is shown on the right side of the statement as a comment

In addition, general variables are also substituted with either an integer pointed by `%RIP` or a value based on `%RSP`. In line 15 of Result 4.10, variable `%37` originally holds `%RIP_5+202`, but redefined with an integer value `4195680` as `%RIP_5` is reassigned. Then, in line 16, `%37` is replaced with its new integer value, `4195680`. Also, the `%56` is substituted with `%RSP-24`; so we can truncated statements (eg. line 28 and 30) with elimination technique in later Section 4.3).

```
----------------------
@fn_400480
----------------------

Detected Idioms
- Idiom 1 (binaryOP): 12
- Idiom 2 (bitwiseOP): 15

"PROPAGATION: 6650 bytes (272)"
```

Result 4.11: IR Code output Summary after the Variable Propagation

After applying the variable propagation (as illustrated in Algorithm 4), the decompilation framework prints out the previous information from idiom detection and the size of the result IR in Result 4.11. In summary, after variable propagatoin, we have the 6650 bytes sized IR written in LL and HLL format and the modified lists.

4.2.3 Double Precision and Naming Variable

While propagating variables, we notice that variable of floating point is referred with %RIP which is substituted by the integer. A typical compiler uses *floating-point* format for a rational number, such as *double*. Also, these non-integer numbers are stored in a *read only* data block, denoted by *.rodata* in assembly code.

Result 4.12 shows the*objdump* disassembled code of the *sample* input. The disassembled code stores initialised data at a memory location relevant to *rbp*, 64-bit base pointer (line 9-12). First two data are constants, but the others are *xmm* registers defined at line 5

and 7. These registers hold data from location `0x400558` and `0x400560`, which are relevant to *rip*, pointing to *.rodata* section in disassembled file, see Result 4.13.

```
1  0000000000400480 <main>:
2  400480: 55                     push    rbp
3  400481: 48 89 e5               mov     rbp,rsp
4  400484: 31 c0                  xor     eax,eax
5  400486: f2 0f 10 05 ca 00 00   movsd   xmm0,QWORD PTR [rip+0xca] # 400558
6  40048d: 00
7  40048e: f2 0f 10 0d ca 00 00   movsd   xmm1,QWORD PTR [rip+0xca] # 400560
8  400495: 00
9  400496: c7 45 fc 00 00 00 00   mov     DWORD PTR [rbp-0x4],0x0
10 40049d: c7 45 f8 0a 00 00 00   mov     DWORD PTR [rbp-0x8],0xa
11 4004a4: f2 0f 11 4d f0         movsd   QWORD PTR [rbp-0x10],xmm1
12 4004a9: f2 0f 11 45 e8         movsd   QWORD PTR [rbp-0x18],xmm0
13 4004ae: 8b 4d f8               mov     ecx,DWORD PTR [rbp-0x8]
14 4004b1: f2 0f 2a c1            cvtsi2sd xmm0,ecx
15 4004b5: f2 0f 59 45 e8         mulsd   xmm0,QWORD PTR [rbp-0x18]
16 4004ba: f2 0f 58 45 f0         addsd   xmm0,QWORD PTR [rbp-0x10]
17 4004bf: f2 0f 11 45 e0         movsd   QWORD PTR [rbp-0x20],xmm0
18 4004c4: 5d                     pop     rbp
19 4004c5: c3                     ret
20 4004c6: 66 2e 0f 1f 84 00 00   nop     WORD PTR cs:[rax+rax*1+0x0]
```

Result 4.12: objdump Disassembled Code

```
1  Contents of section .rodata:
2  400550 01000200 00000000 00000000 00002440  ..............$@
3  400560 a4703d0a d7937340                   .p=...s@
```

Result 4.13: objdump Disassembled .rodata section

The contents at the *.rodata* are ambiguous and non-human readable, but we need the data to convert the LLIR to a HLIR. The decompilation program calls *objdump* command to extract the *.rodata* contents (as shwon in Result 4.13). With the assumption that the data is stored in little-endianness, we can convert the binary data, indicated by `%RIP` register, to a decimal form based on *double-precision floating-point format*.

Moreover, all of the general variables are referred with `%RSP`, instead of a variable name like a string. We create a list of sub-sequences of alphabets and use it to give a human-readable name to the pointer values. Since every variable needs to be declared before use, we add a statement which defines new variable, as shown in line 3, Result 4.14. LLVM IR code below is an example of both floating-point conversion and naming variables, see line 13, Result 4.14.

```
1  define void @fn_400480(%regset* noalias nocapture) {
2    ...
3    %d = alloca double, align 8
4    entry_fn_400480:
5      ...
6    exit_fn_400480: ; preds = %bb_400480
7      ...
8    bb_400480:
9      ...
10     ; %38 = load double, double* 4195680, align 1 ()
11     ; store double %38, double* %RSP-24, align 1
12     ; 4195680 = 0x400560
13     store double 313.24, double* %d, align 1
14   ...
```

Result 4.14: IR Code after Naming Varaibles and Reading Double Precision

note: previous statement is shown on the right side of the statement as a comment

Even though floating-point precision conversion and naming variable are not a part of our propagation or elimination techniques (Algorithm 3 and 4), we consider them as a part of the middle-end phase of decompiler because they transform Low-level Language (LL) properties into High-level Langauge (HLL). After the changes, the decompilation program lists new names of variables and indicated address, pointed by `%RSP` register (see Result 4.15).

```
----------------------
@fn_400480
----------------------

Detected Idioms
- Idiom 1 (binaryOP): 12
- Idiom 2 (bitwiseOP): 15

Detected Variables: 6
- %b <- %RSP-12
- %c <- %RSP-16
- %d <- %RSP-24
- %e <- %RSP-32
- %f <- %RSP-40
- %a <- %RSP-8

"VARIABLE_NAME: 6556 bytes (278)"
"PRECISION: 6538 bytes (278)"
```

Result 4.15: IR Code output Summary after the Variable and Percision Conversion

4.3 Elimination

Elimination method is our last step of middle-end analysis; it cleans up by removing lines with dead variable(s). Since it is the last algorithm before the high-level language conversion (back-end), we expect our result to be shorter than the propagated IR, but also clear to read with minimum low-level formats.

Result 4.16 and 4.17 are the results of Algorithm 5, ELIMINATION function. They both are smaller than the previous IRs and more understandable; the difference is whether the ELIMINATION algorithm keeps the registers (Result 4.16) or forces to remove registers (Result 4.17), defined in the *entry* block, `entry_fn_400480`.

```
1   define void @fn_400480(%regset* noalias nocapture) {
2     %f = alloca double, align 8
3     %e = alloca double, align 8
4     %d = alloca double, align 8
5     %c = alloca i32, align 4
6     %b = alloca i32, align 4
7     %a = alloca i64, align 8
8
9   entry_fn_400480:
10    %RIP = alloca i64
11    %EIP = alloca i32
12    %RBP = alloca i64
13    %RSP = alloca i64
14    %EBP = alloca i32
15    %RAX = alloca i64
16    %EAX = alloca i32
17    %ZMM0 = alloca <16 x float>
18    %XMM0 = alloca <4 x float>
19    %YMM0 = alloca <8 x float>
20    %ZMM1 = alloca <16 x float>
21    %XMM1 = alloca <4 x float>
22    %YMM1 = alloca <8 x float>
23    %RCX = alloca i64
24    %ECX = alloca i32
25    %CtlSysEFLAGS = alloca i32
26    br label %bb_400480
27
28  exit_fn_400480: ; preds = %bb_400480
29    ret void
30
31  bb_400480: ; preds = %entry_fn_400480
32    store i64 %RBP, i64* %a, align 1
33    %EAX_0 = %RAX
34    %EAX_1 = xor i32 %EAX_0, %EAX_0
35    store i32 0, i32* %b, align 1
36    store i32 10, i32* %c, align 1
37    store double 313.24, double* %d, align 1
38    store double 10.0, double* %e, align 1
39    %76 = load double, double* %e, align 1
40    %77 = fmul double %c, %76
41    %89 = load double, double* %d, align 1
42    %90 = fadd double %77, %89
```

```
43        store double %90, double* %f, align 1
44        %CtlSysEFLAGS_0 = load i32, i32* %CtlSysEFLAGS
45        store i32 %CtlSysEFLAGS_0, i32* %CtlSysEFLAGS
46        store i32 %EAX_1, i32* %EAX
47        store i32 %a, i32* %EBP
48        store i32 %c, i32* %ECX
49        store i32 %RSP, i32* %EIP
50        store i64 %EAX_1, i64* %RAX
51        store i64 %a, i64* %RBP
52        store i64 %c, i64* %RCX
53        store i64 %RSP, i64* %RIP
54        store <4 x float> %90, <4 x float>* %XMM0
55        store <4 x float> 313.24, <4 x float>* %XMM1
56        store <8 x float> %90, <8 x float>* %YMM0
57        store <8 x float> 313.24, <8 x float>* %YMM1
58        store <16 x float> %90, <16 x float>* %ZMM0
59        store <16 x float> 313.24, <16 x float>* %ZMM1
60        br label %exit_fn_400480
61      }
```

Result 4.16: IR Code after Elimination Phase before removing registers' statement

Every defined variable in Result 4.16 IR is a live variable with at least one use; the elimination algorithm correctly removes unnecessary lines. Comparing the two result IRs, we are able to delete defined registers in the *entry* block, without changing or removing information about the program's behaviour. Thus, we get Result 4.17, after adding registers as irrelevant variables and applying elimination algorithm, Algorithm 5.

```
1       define void @fn_400480(%regset* noalias nocapture) {
2         %f = alloca double, align 8
3         %e = alloca double, align 8
4         %d = alloca double, align 8
5         %c = alloca i32, align 4
6         %b = alloca i32, align 4
7
8         entry_fn_400480:
9         br label %bb_400480
10
11        exit_fn_400480: ; preds = %bb_400480
12        ret void
13
14        bb_400480: ; preds = %entry_fn_400480
15        store i32 0, i32* %b, align 1
16        store i32 10, i32* %c, align 1
17        store double 313.24, double* %d, align 1
18        store double 10.0, double* %e, align 1
19        %76 = load double, double* %e, align 1
20        %77 = fmul double %c, %76
21        %89 = load double, double* %d, align 1
22        %90 = fadd double %77, %89
23        store double %90, double* %f, align 1
24        br label %exit_fn_400480
25      }
```

Result 4.17: IR Code after Elimination Phase without registers' DEF and USE statements

4.4 HLL Conversion and Program's Completeness

The intention of this thesis is to recreate a source-like code which is more understandable and still has same program behaviour. In this section, we list our expectation on the final product, c-like high-level language code, and evaluate if the output matches the list of expectations. Then, we compare our result code to the original source code of *sample* to see how similar and different they are.

Result 4.18 shows the program's summary from the previous steps, from the pre-processing phase to the elimination method. As mentioned in Section 4.2.1, we handled twelves binary-typed idioms, fifteen bitwise-typed idioms, and six variables in *sample* code. Binary idiom is used to `load` a data into a variable before the use or to `store` a data into the memory location, and bitwise idiom arises when a variable is stored in memory. Thus, we expect our c-like high-level code will have following properties and match to the original source code:

- The code has at most six variables since elimination algorithm might have removed some.
- Totally, five variables are used since a variable is stored in three different registers `%xmm`, `%ymm`, and `%zmm`, and each of them is counted as an individual idiom - bitwise idiom.
- Variables are defined (store) and used (load) twelve times totally- binary idiom

```
----------------------
@fn_400480
----------------------

Detected Idioms
- Idiom 1 (binaryOP): 12
- Idiom 2 (bitwiseOP): 15

Detected Variables: 6
- %b <- %RSP-12
- %c <- %RSP-16
- %d <- %RSP-24
- %e <- %RSP-32
- %f <- %RSP-40
- %a <- %RSP-8
```

```
"Assembly: 29040 bytes (665)"
"Dagger-IR: 124869 bytes (3768)"

"PREPROCESS: 9679 bytes (314)"
"IDIOM: 7826 bytes (272)"
"PROPAGATION: 6650 bytes (272)"

"VARIABLE_NAME: 6556 bytes (278)"
"PRECISION: 6538 bytes (278)"
"ELIMINATION: 1665 bytes (59)"
```

Result 4.18: Overall Program Summary

Result 4.19 is the final output of the decompiler framework. To replicate a HLL format, specifically a C language, block labels are removed from the previous code, shown in Result 4.17, as well as the `load`, `store`, or terminator instruction statements. Hence, the result code has five defined variables, `%f`, `%e`, `%d`, `%c` and `%b`. Four of them are initialised before the calculation; `%e`, `%d`, and `%c` are used to redefine the variable `%f`.

Oppose to our program summary, there are eight DEF and USE in the result code, including USE(`%c`), USE(`%e`), and USE(`%d`); we are missing four binary idioms and two sets of bitwise idioms (total six bitwise idiom). However, this is due to elimination of variable `%a`, and two variables, `%d` and `%e`, which are initialised in double type. Named variable `%a` does not appear in the code because it is a place holder for a 64-bit base pointer, `%RBP`, which is deleted during the elimination phase. `%a` is defined as `%RBP`, in the beginning of the function, and used to restore the register before the function's exit.

Also, `%d` and `%e` require extra work to initialise the variables. As explained in Section 4.2.3, non-integer rational numbers are stored in separate memory locations pointed by *%RIP* register (or *rbp* in disassembled code). Values at each memory location are required to be loaded into variables before the initialisation; this raises flags on binary idiom and bitwise idiom. So, considering all the propagated, transformed, or deleted information, the number of detected idioms and variables from the binary and IR code matches to our final HLLC.

From the above assessment, we know that all the algorithms and analysis phases work as we intended. To match to the source code of the *sample*, we have named our function as `main`; we expect our HLL code to be similar to the source code, shown in Result 4.20.

```
void main ( %regset* noalias nocapture ) {

    double %f
    double %e
    double %d
    int %c
    int %b
    %b = 0
    %c = 10
    %d = 313.24
    %e = 10.0
    %f = %c * %e + %d

    return void
}
```

Result 4.19: HLL Transformation Result

```
int main(){

    int a = 10;
    double b = 313.24, x = 10;

    double y = a * x + b;

    return 0;
}
```

Result 4.20: source code of *sample*

Even though the *sample* does not contain arguments to keep it simple, HLL code has the argument variable `%regset*` and two defined attributes *noalias* and *nocapture*[1]. Also, variables are still denoted by `%`, and the function returns `void`; these are not C language syntax and grammars.

Comparing the HLLC to the source code of *sample*, the return type of the function, arguments, names of variables, and minor syntax are different. However, both code perform multiplication and summation with same values, with different naming; they compute variables, either `%f` or `y`, with a double type rational number, $10 \times 10.0 + 313.24$.

In other words, the optimization technique algorithms, presented in Chapter 3, work as we expected; also, the original source code and the code generated by our decompiler framework perform the same computation, and have the same semantics.

[1] see *Parameter Attributes* section from the LLVM Reference website [21]

Chapter 5

Case Study

To evaluate the generality of our decompiler framework, we construct two different cases, using pre-compiled code, written in C language. First Subsection 5.1 runs different code with a single basic block without functional call/return. Then, we have basic test cases with `if-else` and `switch` conditional statements, in Subsection 5.2.

5.1 Single Basic Blocks

In this section, we use several code samples with different types of variables and assignments (calculation operations) to evaluate our decompiler framework. Similar to the *sample* code from Chapter 4, all the code samples have a single basic block without calling libraries (and functions). Below figures are example code from each category.

Figure 5.1 is one of the example test code with a statement, using a compound assignment (eg. +=, -=, *=, /=). Without complex ISA, the compound assignment can be rewritten with a simple calculation symbol (eg. `A += B` $\rightarrowtail$ `A = A + B`); decompiler generated HLLC 5.1 represents the same program behaviour as Source 5.2. Also, additional testing cases with compound assignment statements gives us expected results.

```
void main ( %regset* noalias nocapture ) {
  long %d
  int %c
  short %b
  %b = 5
  %c = 123
  %d = 43210
  %d = %d - %c * %b
  return void
}
```

HLLC 5.1: HLLC of Source 5.2

```
int main(){

  unsigned short a = 5;
  unsigned int b = 123;
  long c = 43210;

  c -= (a * b);

  return 0;
}
```

Source 5.2: Original Code of HLLC 5.1

Figure 5.1: Single Basic Block, Compound Assignment case

During the previous test (see Figure 5.1), we learn that our decompiler framework does not replicate variables type, except `short`, `int` and `long` for integers, and `fp128`, `ppc_fp128`, `double`, `float`, and `half` for decimal point numbers. The rest of variable remains in iN LLVM type format; since the focus of this thesis is middle-end analysis, our HLL generator is incapable of differentiating `signed` and `unsigned` values.

In addition to a compound assignment, we want to see if the decompiler framework generates a HLLC with an increment and a decrement values correctly. Similar to how compound assignment is decompiled in HLLC 5.1, our framework rewrites the increment (++) and decrement (–) sign to (α + 1). The rewritten statement does not change the program's functionality; HLLC 5.3 and Source 5.4 execute the same program behaviour.

```
void main ( %regset* noalias nocapture ) {
  double %c
  float %b
  %b = 11.0
  %c = 99.0
  %c = %c + 1.0
  %b = %b - 1.0
  return void
}
```

HLLC 5.3: HLLC of Source 5.4

```
int main(){

  float a = 11;
  double b = 99;

  b++;
  a--;

  return 0;
}
```

Source 5.4: Original Code of HLLC 5.3

Figure 5.2: Increment and Decrement

In addition, using bracket is not a problem in HLLC 5.1; however, most of the generated C-like code do not meet our expectation when the code has a computation with bracket(s), see Figure 5.3. First of all, the front-end *dagger* interchanges the order of the

computation during the binary transformation (eg. `(b - a)` instead of `(a - b)`); then, our HLL generator does not add bracket during the IR transformation even though the order of the computation in IR is correct, as shown in Result_IR 5.7.

```
void main ( %regset* noalias nocapture ) {
  int %e
  int %d
  int %c
  short %b
  %b = 64
  %c = 128
  %d = %c * %b - %c + %b
  %e = %c + %b | %c + %b >> 31 << 32 / %c - %b
  return void
}
```

HLLC 5.5: HLLC of Source 5.6

```
int main(){

  short a = 64;
  int b = 128;

  int x = (int) (a + b) - (a * b);
  int y = (int) (a + b) / (a - b);

  return 0;
}
```

Source 5.6: Original Code of HLLC 5.5

Figure 5.3: Single Basic Block, Using Bracket case

```
1   %RSP_1 = %RSP-8                          ; %RSP_1 = %RSP-8
2   %16 = %RSP_1-14                          ; a
3   store i16 64, i16* %16, align 1          ; a (= %RSP-22 = %RSP_1-14) ← 5
4   %18 = %RSP_1-12
5   store i32 128, i32* %18, align 1         ; b (= %RSP-20 = %RSP_1-12) ← 123
6   %EDX_0 = %RSP-22                         ; %EDX_0 = a
7   %EAX_0 = %RSP-20                         ; %EAX_0 = b
8   %EDX_1 = %EAX_0 + %EDX_0                 ; %EDX_1 ← b + a
9   %EAX_1 = %RSP-22                         ; %EAX_1 = a
10  %32 = %RSP_1-12                          ; %32 = b
11  %33 = load i32, i32* %32, align 1
12  %EAX_2 = %33 * %EAX_1                    ; %EAX_2 ← (b * a)
13  %EDX_2 = %EAX_2 - %EDX_1                 ; %EDX_2 ← (b * a) - (b + a)$
14  %38 = %RSP_1-8                           ; %38 = x
15  store i32 %EDX_2, i32* %38, align 1      ; x ← EDX_2
16  %RDX_4 = %RSP-22                         ; %RDX_4 = a
17  %RAX_5 = %RSP-20                         ; %RAX_5 = b
18  %46 = %RAX_5 + %RDX_4                    ; %46 ← b + a
19  %ECX_0 = %RAX_5 + %RDX_4                 ; %ECX_0 ← b + a
20  %EAX_4 = %RSP-22                         ; %EAX_4 = a
21  %53 = %RSP_1-12                          ; %53 = b
22  %54 = load i32, i32* %53, align 1
23  %EAX_5 = %54 - %EAX_4                    ; %EAX_5 ← b - a
24  %EDX_4 = ashr i32 %ECX_0, 31             ; %EDX_4 ← %EXC_0 >> 31
25  %59 = zext i32 %EDX_4 to i64
26  %60 = zext i32 %ECX_0 to i64
27  %61 = shl i64 %59, 32                    ; %61 ← (%EXC_0 >> 31) << 32
28  %62 = or i64 %61, %60                  ; %61 ← ((%EXC_0 >> 31) << 32) || (b + a)
29  %63 = sext i32 %EAX_5 to i64             ; %63 = %EAX_5
30  %64 = sdiv i64 %62, %63    ; %64 ← [((%EXC_0 >> 31) << 32) || (b + a)] / (b - a)
31  %EAX_6 = %64                             ; %EAX_6 = %64
32  %69 = %RSP_1-4                           ; y = %69
33  store i32 %EAX_6, i32* %69, align 1      ; y ← %EAX_6
```

Result_IR 5.7: IR code of Figure 5.3 after Idiom Detection and Conversion

dagger generates `ashr` (arithmetic right shift), `shl` (logical left shift), and `or` operations because of the type conversion `(int)` in Source 5.6 ()see line 24, 27, and 28 in Result_IR 5.7). However, `%61` becomes zero, and `%62` becomes `(0 || %60)` which is `(b + a)` since our integer is 32-bits, and we have `(%EXC_0 >> 31) << 32`(at line 27). Consequently, `%EAX_6` becomes `(b + a) / (b - a)` which still results different program behaviour but holds similar format.

Our third single basic block case is the numeric values type conversion; as shown in Figure 5.3 and 5.4, *dagger* translates type conversion, among `short`, `int`, and `long`, using additional `ashr` and `shl`. But the HLLC 5.8 still holds same program behaviour as the Source 5.9.

```
void main ( %regset* noalias nocapture ) {
  long %d
  long %c
  short %b
  %b = 5
  %c = 123
  %d = %c >> 63 << 64 | %c / %b
  return void
}
```

HLLC 5.8: HLLC of Source 5.9

```
int main(){

  short a = 5;
  long b = 123;

  long x = b / (long) a;

  return 0;
}
```

Source 5.9: Original Code of HLLC 5.8

Figure 5.4: Numeric Variables Type Conversion from Short to Long

In the contrast to the previous test case with integer numeric values, the additional operators are not used when we try to convert `float` type to a `double` type. As shown in Result_IR 5.12, there is no shifting operations but several type conversion operations of LLVM IR, such as `fpext`, at line 4. Also, HLLC 5.10 and Source 5.11 have same program behaviour; our decompiler framework works as we expected and generated HLLC with floating-point conversion without additional operations.

```
void main ( %regset* noalias nocapture ) {
  double %d
  double %c
  float %b
  %b = -11.109999656677246
  %c = 12345.678
  %d = %c / %c - %b
  return void
}
```

HLLC 5.10: HLLC of Source 5.11

```
int main(){

  float a = -11.11;
  double b = 12345.678;

  double x = b / (double) b - (double) a;

  return 0;
}
```

Source 5.11: Original Code of HLLC 5.10

Figure 5.5: Numeric Variables Type Conversion from Float to Double

```
1     ...
2     %91 = %RSP_1-20                          ; variable 'a', from Source 5.11
3     %92 = load float, float* %91, align 1    ; float a
4     %93 = fpext float %92 to double          ; double a, using 'fpext' LLVM ↩
          Instruction
5     %94 = bitcast double %93 to i64          ; i64 a
6     %XMM1_3 = %XMM1_2 : %94                  ; i128 a, see Subsection 3.2.1 in Chapter 3
7     ...
8     %103 = trunc i128 %XMM1_3 to i64         ; i64 a
9     %104 = bitcast i64 %103 to double        ; double a
10    ...
```

Result_IR 5.12: Change of variable type a in Figure 5.5's Idiom IR code

Furthermore, we run test cases with a char typed variable. Following Figure 5.6, is an example test case where we only defines char variables. Suppose 'a' is 97 in decimal integer, variable 'a' and ('a' + 4) are treated as integers value (see Result_IR 5.15). As a result, our decompiler framework correctly replicates Source 5.14 and results in HLL 5.13, in Figure 5.6.

```
void main ( %regset* noalias nocapture ) {
  i8 %c
  i8 %b
  %b = 97
  %c = 101
  return void
}
```

HLLC 5.13: HLLC of Source 5.14

```
int main(){

  char a = 'a';
  char x = 'a' + 4;

  return 0;
}
```

Source 5.14: Original Code of HLLC 5.13

Figure 5.6: Char Type Variable 1

```
1   ...
2   %c = alloca i8, align 1
3   %b = alloca i8, align 1
4   ...
5   store i8 97, i8* %b, align 1
6   store i8 101, i8* %c, align 1
7   ...
```

Result_IR 5.15: IR code after the Elimination Phase for Figure 5.6

On the other hand, our framework does not generate any statement which uses `char` typed variable(s) because of our early assumption. In Section 3.1, Chapter 3, we made assumption where lower-bit registers (eg. %AL, %AH, and %AX) will not be useful because our focus is on 32-bit and 64-bit architectures and deleted all the statements which use lower-bit registers, during the preprocessing phase.

Although most machines use 32/64-bit architecture, `char` type is defined as 8-bit and uses 8-bit registers for `load` and `store` instruction. Defining the variable initially would not cause a problem, but using a variable or reassigning value requires to use lower-bit registers. Thus, generated HLLC does not match to its original source code, see Figure 5.7 and Figure 5.8.

```
void main ( %regset* noalias nocapture ) {
  i8 %b
  %b = 97
  return void
}
```

HLLC 5.16: HLLC of Source 5.17

```
int main(){
  char a = 'a';
  char x = a;
  return x;
}
```

Source 5.17: Original Code of HLLC 5.16

Figure 5.7: Char Type Variable 2

```
void main ( %regset* noalias nocapture ) {
  i8 %c
  i8 %b
  %b = 97
  %c = 65
  return void
}
```

HLLC 5.18: HLLC of Source 5.19

```
int main(){

  char a = 'a', b = 'A';
  char x = a + b - 16;

  return 0;
}
```

Source 5.19: Original Code of HLLC 5.18

Figure 5.8: Char Type Variable 3

In conclusion, sometimes, an order of variables in a computation statement can be incorrect, but our decompiler framework generates correct types and compound assignment calculation. However, due to our assumption on lower-bit registers, the framework is incapable of generating correct HLLC when it comes to using `char` typed variables.

5.2 Conditional Statements

In this section, we use code with two different type of conditional statements, `if-else` and `switch`, to evaluate the decompiler framework. No code has a statement which uses `char` type variable since we know that our framework does not handle 8-bit (and 16-bit) registers. Individual example code represents different cases of similar patterned code.

Figure 5.9 is the first example of conditional statement test cases. As shown in Source 5.21, we want a single `if` statement with a `(x == y)` condition in our HLLC; however, *dagger* generates an additional `if` statement which checks if the two variables, in the condition, are numbers (numeric values), see HLLC 5.20. Thus, the framework generates the expected comparison statement as the `else` statement; generated HLLC and its original source code have the same program behaviour.

```
void main ( %regset* noalias nocapture ) {
  int %d
  double %c
  double %b
  %b = 12.3            ; x
  %c = 12.3            ; y
  %d = 0               ; a
  ; IF x or y is NOT a Number
  if (%b == NaN || NaN == %c) {
  } else {
    if (%c == %b) { ; IF (y == x)
      %d = 1         ;   THEN a = 1
    }
  }
  return void
}
```

HLLC 5.20: HLLC of Source 5.21

```
int main(){

  double x = 12.3, y = 12.3;
  int a = 0;

  if (x == y){
    a = 1;
  }

  return 0;
}
```

Source 5.21: Original Code of HLLC 5.20

Figure 5.9: Conditional Statement, If Statement

In addition to `if` statements, we run test code with `else` statements; Figure 5.10 and Figure 5.12 are decompiled HLLC and their original source code. First figure (see Figure 5.10) has an `equal` (==) comparison condition, where as Figure 5.12 has `less than equal` ($\leq$) one, see HLLC 5.20; we expect to see the `if` statement with the correct comparison and the `else` statement.

Similar to the previous test case, Figure 5.9, generated HLLC 5.22 has the additional NaN (Not a number) comparison if statement and correctly regenerated conditional statements, as we expected. However, it has additional statements which are not from the Source 5.23. These incorrect additional statements are generated while the decompiler transforms an IR to a HLL, using a CFG.

```
void main ( %regset* noalias nocapture ) {
  int %d
  double %c
  double %b
  %b = -12.3                    ; x
  %c = %b * 4.0 + 7.2           ; y
  %d = 0                        ; a
  ; IF x or y is NOT a Number
  if (%b == NaN || NaN == %c) {
  } else {
    if (%b == %c) {             ; IF (x == y)
      %d = %b - %c              ;    a = x - y
    } else {                    ; ELSE (x /= y)
      %d = %b * %c              ;    a = x * y
    }
  }
  %b * %c = %b * %c
  %d = %b * %c
  %b - %c = %b - %c
  %d = %b - %c
  return void
}
```

HLLC 5.22: HLLC of Source 5.23

```
int main(){

  double x = -12.3, y = (4 * x) + 7.2;
  int a = 0;

  if (x == y){
    a = x - y;
  }else{
    a = x * y;
  }

  return 0;
}
```

Source 5.23: Original Code of HLLC 5.22

Figure 5.10: Conditional Statement, If and Else Statement with Equal condition

During the HLL generation, we draw a parenthesis around the list of sorted vertices to indicate the beginning and the end of conditional statements. Beginning of the conditional jump is marked when the last line of a block has a branch (br) instruction, and the end of the conditional jump is marked when the false block is called.

Figure 5.11 is a CFG of Figure 5.10. When 2:NaN is true, it should go to block 6:return. However, *dagger* translates binary code such that 2 calls the block 6 when 2 condition is true. Thus, as shown in the figure, the order of blocks is 1 2 3 5 4 6, and marking a conditional statement results in the order 1 (2 (3) 5 4) 6. The order of the block is supposed to be 1 (2 (3 5) 4) 6, but we draw a first close parenthesis after 3 because the block is called when the 2 condition is false.

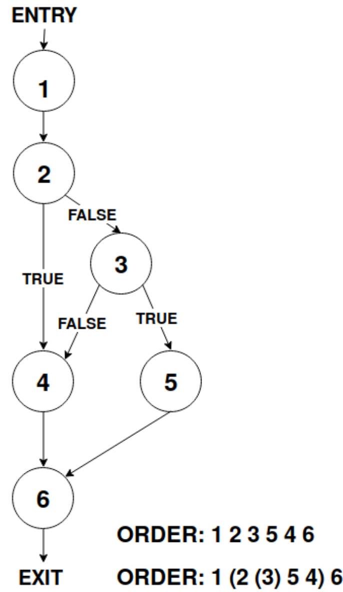


Figure 5.11: Control Flow Graph of Figure 5.10 and its Block Order

Considering `(3)`, the HLL generator does not see the rest of blocks `4 5` until it exits current conditional statement; it forces to call block `4` and `5` and displays their block content. Once it exits, it sees the order `(2 (3) 5 4)` and points to `5`, then `4`, which is already displayed. Thus, result HLLC has duplicated and mixed statements.

In the contrast to HLLC 5.22, our second `if-else` code does not include the duplicated `if` and `else` block statements. Even though the inequality comparison is different ($y \leq x$) than what we want to see ($x < y$), the functionality of both generated HLLC 5.22 and the original Source 5.23 are same.

```
void main ( %regset* noalias nocapture ) {
  int %d
  double %c
  double %b
  %b = 12.3                      ; x
  %c = %b * 2.5                  ; y
  %d = -103                      ; a

  if (%c < %b | %c == %b) {  ; IF (y <= x)
    %d = %c - %d                 ;    a = y - a
  } else {                       ; ELSE (y > x)
    %d = %b * %b                 ;    a = x * x
  }
  return void
}
```

HLLC 5.24: HLLC of Source 5.25

```
int main(){

  double x = 12.3, y = (x * 2.5);
  int a = -103;

  if (x < y){
    a = x * x;
  }else{
    a = y - a;
  }

  return a;
}
```

Source 5.25: Original Code of HLLC 5.24

Figure 5.12: Conditional Statement, If and Else Statement with None Equal condition

From the test case above, we learn that these incorrect additional statements only happens when the condition is `equal` (==), in `if-else` conditional statement; our decompiler runs as we expected during the `middle-end` analysis phase and faces some difficulty during the IR translation. Thus, not all code with `if-else` statement is translated from binary code to a HLLC accurately.

Once we face some limitations of our decompiler framework, we want to see if this duplication error would happen in conditional statements with an `else if` condition. We use a code with an equality in `if` condition and an inequality in `else if` condition, see Figure 5.13, to evaluate our framework.

```
void main ( %regset* noalias nocapture ) {
  int %e
  int %d
  double %c
  double %b
  %b = -12.3                          ; x
  %c = 32.1                           ; y
  %d = 32                             ; a
  %e = 97                             ; c
  ; IF x or y is NOT a Number
  if (%b == NaN || NaN == %c) {
  } else {
    if (%c == %b) {                   ; IF (y == x)
      %e = %e - %d                    ;    THEN (c = c - a)
    } else {
      if (%c < %b | %c == %b) {       ; ELSE IF (y <= x)
        %e = -1                       ;    THEN c = -1
      } else {                        ; ELSE (y > x)
          %e = %e + 1                 ;    THEN c = c + 1
      }
    }
  }
  return void
}
```

HLLC 5.26: HLLC of Source 5.27

```
int main(){

  double x = -12.3, y = 32.1;
  int a = 32;
  int c = 97;

  if (x == y){
    c = c - a;
  }else if (x < y){
    c++;
  }else{
    c = -1;
  }

  return 0;
}
```

Source 5.27: code of HLLC 5.26

Figure 5.13: Conditional Statement, If, Else-If Statement

Similar to the HLLC 5.24, comparison, in HLLC 5.26, it is not exactly the same as the one in Source 5.27, but the functionality of the conditional statements in HLLC is same as the one in the source code. Also, generated HLLC 5.24 does not contain inaccurate additional statements which happened in Figure 5.10. The decompiler works as we expected and translates a binary `else if` conditional statements into a HLL `else if` statement.

Besides the `if-else` conditional statement, C language has a `switch` statement;

Figure 5.14 is an example of test code with switch statement and we expect either a generated switch statement or a nested if-else statements to replicate the switch. As result in Section 5.1, the decompiler cannot translate statements which usechar type variables, but it works with statements which define char variables. So, as shown in Figure 5.14, result of decompiling Source 5.29 generates HLLC 5.28 that has a same functionality as its source code; the switch is rewritten in a nested if-else format in the HLLC.

```
void main ( %regset* noalias nocapture ) {
  i8 %c
  i8 %b
  %b = 99
  if (%b == 65) {
    %c = 65
  } else {
    if (%b == 97) {
      %c = 97
    } else {
      if (%b != 48) {
        %c = 45
      } else {
        %c = 48
      }
    }
  }
  return void
}
```

HLLC 5.28: HLLC of Source 5.29

```
int main(){

  char c = 'c', d; // ascii decimal 99

  switch(c){
    case '0':
      d = '0';    // ascii decimal 65
      break;

    case 'A':
      d = 'A';    // ascii decimal 97
      break;

    case 'a':
      d = 'a';    // ascii decimal 48
      break;

    default:
      d = '-';    // ascii decimal 45
  }
  return 0;
}
```

Source 5.29: Original Code of HLLC 5.28

Figure 5.14: Conditional Statement, Switch Statement

Finally, we have a last test case where code has both *if-else* and *switch* statements, see Figure 5.15. See Source 5.31, the if-else statement comes first and assigns an integer to the value a. Then, switch takes a, as a condition, and defines b. Since both statements are related, we expect the HLLC to either have an *if-else* statement and a nested if-else statement separately.

Unfortunately, the decompiler does not translate the nested statements from switch, as shown in HLLC 5.30. Assigning statement of case from switch is printed out under the if statement; then, the rest of else if and else statements are displayed.

```
void main ( %regset* noalias nocapture ) {
  int %e
  int %d
  double %c
  double %b
  %b = -12.3            ; x
  %c = 32.1             ; y
  ; IF x or y is NOT a Number
  if (%b == NaN || NaN == %c) {
  } else {
    if (%b == %c) {    ; IF (x == y)
      %d = 0           ;   a = 0
      %e = -1          ;   b = -1
      %e = 1           ;   b = 1
      %e = 0           ;   b = 0
    } else {
                       ; ELSE IF (y <= x)
      if (%c < %b | %c == %b) {
        %d = 1         ;   a = 1
      } else {         ; ELSE (y > x)
        %d = -1        ;   a = -1
      }
    }
  }
  return void
}
```

HLLC 5.30: HLLC of Source 5.31

```
  double x = -12.3, y = 32.1;
  int a, b;

  if(x == y){
    a = 0;
  }else if(x < y){
    a = -1;
  }else{
    a = 1;
  }

  switch(a){
    case -1:
      b = -1;
      break;
    case 1:
      b = 1;
      break;
    default:
      b = 0;
  }

  return b;
}
```

Source 5.31: Original Code of HLLC 5.30

Figure 5.15: Conditional Statement, If Else and Switch Statement

As shown in Figure 5.16, when the `switch` comes before the `if-else`, `switch` statement is generated in nested `if-else` statement format, see HLLC 5.32; the `if-else` statement from the source is displayed under the first `if` statement of the `switch`.

Our decompiler also fails additional test code, with *if-else* and *switch* statements; it prints out block contents under the first nested `if` statement but never generates the second conditional statement correctly.

In conclusion, the decompiler framework generates a C-like high-level language code that has similar functionality as the original source code, except for `use(char v)` statements. Although an order of two variables are switched during binary translation, our middle-end framework does not recognize the incorrect statement. So it leads the decompiler to generate inaccurate results. Also, the results are inaccurate when a `if-else` condition, but not a `else if`, uses `equality`, and two separate conditional statements are related to each other. However, these problems happen during the IR translation and HLL generation; the decompiler's middle-end analysis phase works as we expected.

```
void main ( %regset* noalias nocapture ) {
  int %f
  int %e
  double %d
  double %c
  double %b
  %b = 100.0
  %c = -10.0
  if (%b - %c == 90) {     ; SWITCH CASE 90:
    %e = -1                ;    b = -1
  } else {
    if (%b - %c == 110) { ; SWITCH CASE
        110:
      %e = 1               ;    b = 1
    } else {               ; SWITCH DEFAULT:
      %e = 0               ;    b = 0
      %f = 0
      %f = 1
      %f = -1
    }
  }
  return void
}
```

HLLC 5.32: HLLC of Source 5.33

```
int main(){

  double x = 100.0, y = -10.0;
  int a, b;

  switch((int) (x - y)){
    case 90:
      b = -1;
      break;
    case 110:
      b = 1;
      break;
    default:
      b = 0;
  }

  if(b < 0){
    a = -1;
  }else if(a > 0){
    a = 1;
  }else{
    a = 0;
  }

  return 0;
}
```

Source 5.33: Code of HLLC 5.32

Figure 5.16: Conditional Statement, If Else and Switch Statement

Chapter 6

Conclusion

A decompiler is a program translator, transforming an executable program to a High-level Language Code (HLLC). Like a debugger and dissembler, it is used for malware analysis, vulnerability detection, safety verification, and supports static analysis.

Previously, open-source decompilers produce a source-like code with low-level properties like `GOTO` operators and non-eliminated registers; also, some are not being currently maintained. During our research, RetDec, from Avast Software [1], had an online decompiler which covered 32-bit architectures but provided library for IDA, from Hex-Rays [14], which is a closed-source program.

In this thesis, we attempted to implement code optimization techniques, propagation and elimination, to reduce the low-level language properties during the decompilation process. Our decompiler uses *dagger* to translate binary code to *LLVM-Intermediate Representation (IR)*; it performs compiler optimization techniques, propagation and elimination. Based on the program's argument, it prints out either the *IR* code after a certain analysis phase or a decompiled high-level language code. However, the program is limited to a straight lined binary code or one with conditional jumps, which are not loops, without a `char` type.

Although our attempt of designing a decompiler is based on several assumptions,

which leads to the failure in decompiling a binary with `char` types and generating loop conditions in the final product, it supports certain program analysis, especially a simple module. Our decompiler still supports some static analysis, in limited circumstances. Also, program analysts are able to use IR codes and see the data-flow and changes made in each phase. Written in Haskell, decompiler is able to take a mathematical proof code, more feasibly, for analysis and verification.

Our decompiler needs to support low bit registers for *char* typed variables and improve conditional-case analysis since typical executable programs have `String` typed variables and multiple, even nested, loops. Additional improvements include pointer analysis and keeping attribute functions for program analysis.

Our Current version decompiler backtracks the IR code to remove the SSA formatted registers and keep the original ones, such that the decompiler can easily detect variables and re-write the code to re-use the same variables. Since our test case has a single `main` function with a straight line of code and does not have pointers except for registers, we do not track the changes of pointers (registers). However, pointer analysis is essential to handle `String` and any kind of `list`, and to decompile functional calls and returns, including attribute functions. With the additional features, the decompiler will be more practical for program analysis, including vulnerability detection and malware analysis.

Abbreviations

BB	Basic Block
CFA	Control Flow Analysis
CFG	Control Flow Graph
DFA	Data Flow Analysis
DU	Define-Use chain
ELF	Executable and Linkable Format
HLIR	High-level Intermediate Representation
HLL	High-level Langauge
HLLC	High-level Language Code
IR	Intermediate Representation
ISA	Instruction Set Architecture
LHS	Left Hand Side
LL	Low-level Language
LLC	Low-level Language Code
LLIR	Low-level Intermediate Representation
MC	Machine Code
RHS	Right Hand Side
SSA	Static Single Assignment Form
UD	Use-Define chain

Bibliography

[1] Ava [n.d.], *Retargetable Decompiler*. Accessed on 2017-03-25; Last Accessed on 2019-04-19.

URL: https://retdec.com/

[2] *Boomerang* [n.d.].

URL: http://boomerang.sourceforge.net/

[3] Brumley, D., Jager, I., Avgerinos, T. and Schwartz, E. J. [2011], Bap: A binary analysis platform, *in* G. Gopalakrishnan and S. Qadeer, eds, 'Computer Aided Verification', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 463–469.

URL: http://users.ece.cmu.edu/~aavgerin/papers/bap-cav-11.pdf

[4] Caprino, G. [2015], 'Rec studio 4 - reverse engineering compiler'.

URL: www.backerstreet.com/rec/rec.htm

[5] Caprino, G. [n.d.], 'Decompilers'. Accessed on 2018-05-03.

URL: http://www.backerstreet.com/decompiler/decompilers.htm

[6] Chow, F. [2013], 'Intermediate representation', *Queue* **11**(10), 30:30–30:37.

URL: http://doi.acm.org/10.1145/2542661.2544374

[7] Cifuentes, C. [1994*a*], Interprocedural data flow decompilation, Technical report, Australia.
URL: http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai%3Ancstrlh%3Aqut_fit%3Ancstrl.qut_fit%2F%2FFIT-TR-1994-04

[8] Cifuentes, C. [1994*b*], Reverse Compilation Techniques, PhD thesis, Queensland University Of Technology.
URL: http://taz.newffr.com/TAZ/Win32/win32inc/reversing/decompilation_thesis.pdf

[9] Cifuentes, C. and Gough, K. J. [1995], 'Decompilation of binary programs', *Softw. Pract. Exper.* **25**(7), 811–829.
URL: https://pdfs.semanticscholar.org/a5f9/2c049dc8062501dd36f149d746e8140f09e9.pdf

[10] Cifuentes, C., Simon, D. and Fraboulet, A. [1998], Assembly to high-level language translation, *in* 'Proceedings of the International Conference on Software Maintenance', ICSM '98, IEEE Computer Society, Washington, DC, USA, pp. 228–.
URL: http://dl.acm.org/citation.cfm?id=850947.853321

[11] Derevenets, Y. [n.d.], 'Snowman'.
URL: https://derevenets.com/index.html

[12] Dogtiev, A. [2018], 'App download and usage statistic 2017'. Accessed on 2018-05-02.
URL: www.businessofapps.com/data/app-statistics/

[13] *GNU Compiler Collection (GCC) Internals* [n.d.].
URL: https://gcc.gnu.org/onlinedocs/gccint/#SEC_Contents

[14] *Hex-Rays* [n.d.].

URL: https://www.hex-rays.com/index.shtml

[15] Horgan, P. [2010], 'Linux x86 program start up'. Accessed on 2018-10-24.

URL: http://dbp-consulting.com/tutorials/debugging/linuxProgramStartup.html

[16] *Intermediate Representation* [n.d.].

URL: https://docs.angr.io/advanced-topics/ir

[17] Křoustek, J. [2014], Retargetable Analysis of Machine Code, PhD thesis, Faculty of Information Technology, Brno University of Technology, CZ.

URL: http://www.fit.vutbr.cz/study/DP/PD.php?id=482&file=t

[18] Lattner, C. [2012], Llvm, *in* A. Brown and G. Wilson, eds, 'The Architecture of Open Source Applications', Creative Commons, chapter 11.

URL: http://www.aosabook.org/en/llvm.html

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

URL: https://llvm.org/pubs/2004-01-30-CGO-LLVM.html

[20] LLVM [2018*a*], 'The llvm compiler infrastructure'.

URL: https://llvm.org

[21] LLVM [2018*b*], 'Llvm language reference manual'.

URL: https://llvm.org/docs/LangRef.html

[22] Necula, G. C., McPeak, S., Rahul, S. P. and Weimer, W. [2002], *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, Springer Berlin

Heidelberg, Berlin, Heidelberg.

URL: https://people.eecs.berkeley.edu/~necula/Papers/cil_cc02.pdf

[23] Ora [2010], *Initialization and Termination Sections*. Accessed on 2018-10-24.

URL: https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-48195/index.html

[24] Repzret [2017], 'Repzret/dagger.'.

URL: https://github.com/repzret/dagger

[25] Shawn Hickey and Michael Satran [2018], 'Windows data types'.

URL: https://docs.microsoft.com/en-us/windows/desktop/winprog/windows-data-types

[26] Sou [2006], 'Source code definition.'. Accessed on 2018-05-03.

URL: http://www.linfo.org/source_code.html

[27] Sou [2018], 'What is free software?'. Accessed on 2018-05-03.

URL: https://www..org/philosophy/free-sw.en.html

[28] Sugeerth [2017], 'Power of emerging economies on the global internet businesses'. Accessed on 2018-05-02.

URL: https://medium.com/@sugeerth/power-of-emerging-economies-on-th

[29] Van Emmerik, M. [2007], Static Single Assignment for Decompilation, PhD thesis, University of Queensland.

URL: https://yurichev.com/mirrors/vanEmmerik_ssa.pdf

■Ďurfina et al.

[30] Ďurfina, L., Křoustek, J., Zemek, P., Kolář, D., Hruška, T., Masařík, K. and Meduna, A. [2011], 'Design of a retargetable decompiler for a static platform-independent mal-

ware analysis', *International Journal of Security and Its Applications* **5**(4), 91–106.
URL: http://www.sersc.org/journals/IJSIA/vol5_no4_2011/8.pdf

[31] Wikipedia contributors [2018], 'Use-define chain — Wikipedia, the free encyclopedia'.
URL: https://en.wikipedia.org/w/index.php?title=Use-define_chain&oldid=867797121

Appendix A

Source Code of Algorithms

A.1 Haskell Source code for Algorithm 1: BINARYOP

```
1 binaryOP v new_state nextline vList
2   | ((not.isNothing) nv &&(instrType nvar =="conversion")&&(op nvar =="inttoptr")) = do
3     let newLine = concat[(fromJust nv), " = ", new_state]
4         n = fromJust $ lookupList (fromJust nv) vList
5         n' = n{ instruction="binaryOP", state=new_state }
6         newList = updateList n' (rmVariable c vList []) []
7     ([newLine], "", newList)
8   | otherwise = do
9     let newLine = concat[v, " = ", new_state]
10        c' = c{ instruction="binaryOP", state=new_state }
11        newList = updateList c' vList []
12    ([newLine], nextline, newList)
13    where (nv, (nvar, nreg)) = statement nextline
14          c = fromJust $ lookupList v vList
```

Haskell Code A.1: Source Code of Binary Idiom

A.2 Haskell Source code for Algorithm 2: BITWISEOP

```
1  bitwiseOP pre curr next content vList
2    | ((not.isNothing) pv && (not.isNothing) nv) = do
3      let [p, c, n] = map fromJust $ map (`lookupList` vList) $ map fromJust [pv, cp, nv]
4          useP = length (filter (not.null) $ findUse (fromJust pv) content)
5          useC = length (filter (not.null) $ findUse (fromJust cv) content)
6          useN = length (filter (not.null) $ findUse (fromJust nv) content)
7          isUse_one = (useP == 1) && (useC == 1)
8          isIdiomInstr = (or$map (getInstr p ==) ["zext", "trunc"]) && (getInstr n =="or")
9          isIdiom = (elem (fromJust pv) nreg) && (elem (fromJust cv) nreg)
10     if (isUse_one && isIdiomInstr && isIdiom)
11       then do
12         let newState = (head $ values cvar) ++ " : " ++ (value pvar)
13             newV = n{ instruction="bitwiseOP", state=newState }
14             newLine = concat [variable n, " = " , newState]
15             vList2 = updateList newV (rmVariable c (rmVariable p vList []) []) []
16         ("", [newLine], "", vList2)
17       else (pre, [curr], next, vList)
18   | otherwise = (pre, [curr], next, vList)
19     where [(pv, (pvar, preg)), (nv, (nvar, nreg))] = map statement [pre, next]
20           (cv, (cvar, creg)) = statement curr
```

Haskell Code A.2: Source Code of Handle Bitwise Idiom

A.3 Haskell Source code for Algorithm 3: IDIOM

```haskell
detectIdiom :: [String] -> [String] -> Function -> Integer -> Integer -> Function
detectIdiom [] txt f bin bit = (f {code = txt})
detectIdiom (line: content) oldTxt f binaryN bitwiseN
  | (isLHS line (fname f)) = do
    let (v, (xState, reg)) = statement (strip line)

    case (instrType xState) of
      "binary" -> do
        let instr = op xState
            vList = variables f

        if (or $ map isNum reg)
          then do
            let ptr = bool (last reg) (head reg) (isNum $ last reg)
                idx = bool (read (head reg) :: Integer) (read (last reg) :: Integer) (isNum $ last reg)

            if (instr == "add" || instr == "sub" && hasRegPointer (last $ splitOn " = " line))
              then do
                let sym = bool (bool "+" "-" (idx < 0)) (bool "-" "+" (idx < 0)) (instr /= "add")
                    new_state = concat [ptr, sym, show (abs idx)]
                    (cline, nline, newList) = binaryOP (fromJust v) new_state (head content) vList
                    newContent = filter (not.null) $ nline:(tail content)
                (f{variables = newList}) (binaryN + 1) bitwiseN

              else (detectIdiom content (oldTxt ++ [line]) f binaryN bitwiseN)

          else if (instr == "add" || instr == "sub" && hasRegPointer (last $ splitOn " = " line))
            then do
              let sym = bool " + " " - " (instr /= "add")
                  new_state = concat [head reg, sym, last reg]
                  (cline, nline, newList) = binaryOP (fromJust v) new_state (head content) vList
                  newContent = filter (not.null) $ nline:(tail content)
              detectIdiom newContent (oldTxt ++ cline) (f {variables = newList}) (binaryN + 1) bitwiseN
            else (detectIdiom content (oldTxt ++ [line]) f binaryN bitwiseN)

      "bitwise" -> do
        let instr = op xState
            [a, b] = reg
            vList = variables f

        if ((instr == "and") && (isNum b || isNum' b) && (isInt $ strToFloat b) && (is0s $ strToInt b))
          then do
            let (pline, cline, nline, newList) = bitwiseOP (last oldTxt) (line) (head content) (line : content) vList
                newContent = filter (not.null) $ nline:(tail content)
                newPre = filter (not.null) $ init oldTxt ++ [pline] ++ cline
            detectIdiom newContent newPre (f {variables = newList}) binaryN (bitwiseN + 1)

          else (detectIdiom content (oldTxt ++ [line]) f binaryN bitwiseN)

      _ -> detectIdiom content (oldTxt ++ [line]) f binaryN bitwiseN

  | otherwise = detectIdiom content (oldTxt ++ [line]) f binaryN bitwiseN
```

Haskell Code A.3: Source Code of Detecting Idiom

A.4 Haskell Source code for Algorithm 4: PROPAGATION

```
1  propagation :: Function -> [String] -> [String] -> Function
2  propagation f [] newCode = f{code = newCode}
3  propagation f (line: nextCont) preCont
4    | (isFunction line || isFunctionEnd line || isBlockLabel line || isEntryExit line) = ↩
       propagation f nextCont (preCont ++ [line])
5    | (isLHS line (fname f)) = do
6      let (x, (xvar, xreg)) = statement (strip line)
7          (lhs:rhs:_) = splitOn " = " line
8          vList = variables f
9          v = fromJust $ lookupList lhs vList
10
11     if (isRegPointer line)
12       then do -- REGISTER --
13         if (isInfixOf "_init" line || isInfixOf "_ptr" line || elem lhs reg_base)
14           then propagation f nextCont (preCont ++ [line]) -- REGISTER: initial or ptr --
15           else do
16             let pList = registers f
17                 p = fromJust $ lookupList_p lhs pList
18             if (instrType xvar == "colon")
19               then do -- %REG = a : b
20                 let p' = p{ rstate=(low xvar), permit=True }
21                     newLine = concat[lhs, " = ", getRstate p']
22                     newList = updateList_p p' pList []
23                     (new_nextCont, fnew) = propagateRegister (f {registers = newList}) ↩
                         nextCont lhs p' []
24                 propagation fnew new_nextCont (preCont ++ [newLine])
25               else do
26                 let newLine = concat[lhs, " = ", getRstate p]
27                     (new_nextCont, fnew) = bool (nextCont, f) (propagateRegister f ↩
                         nextCont lhs p []) (getPermit p)
28                 propagation fnew new_nextCont (preCont ++ [newLine])
29
30       else do -- VARIABLE --
31         case (instrType xvar) of
32           "conversion" -> do
33             let (oldType, oldVar) = (ty xvar, lhs)
34                 (newType, newVar) = (ty1 xvar, head xreg)
35                 (nextCont2, newList) = propagateVariable nextCont oldVar newVar vList []
36             propagation (f{variables = newList}) nextCont2 (preCont ++ [lhs ++ " = " ++ ↩
                 head xreg])
37
38           "colon" -> do
39             let (oldVar, newVar) = (lhs, low xvar)
40                 (nextCont2, newList) = propagateVariable nextCont oldVar newVar vList []
41             propagation (f{variables = newList}) nextCont2 (preCont ++ [lhs ++ " = " ++ ↩
                 newVar])
42
43           "binaryOP" ->
44             propagation f (replace' lhs (getState v) nextCont) (preCont ++ [line])
45
46           _-> case (op xvar) of
47             "load" -> do
48               let newState = replace lhs (head xreg) (snd $  strSplit' "=" line)
49                   v = fromJust $ lookupList lhs vList
50                   newList = updateList (setState newState v) vList []
51               propagation (f { variables = newList}) nextCont (preCont ++ [lhs ++ " = " ↩
                   ++ newState])
52
53             "equal" -> propagation f (replace' lhs rhs nextCont) (preCont ++ [line])
54     _ -> propagation f nextCont (preCont ++ [line])
55
56   | otherwise = propagation f nextCont (preCont ++ [line])
```

Haskell Code A.4: Source Code of Propagation

A.5 Haskell Source code for Algorithm 5: ELIMINATION

```
1  variableElim :: Bool -> Function -> [String] -> [String] -> Function-- [LeftVar] -> [RP]↩
       -> [String] -> ([String], [LeftVar])
2  variableElim False f [] newCode = f {code = newCode }
3  variableElim True f [] preCont = variableElim False f preCont []
4  variableElim change f (line : nextCont) preCont
5    | (isFunction line || isBlockLabel line || isBasicBlock line || isEntryExit line) = ↩
        variableElim change f nextCont (preCont ++ [line])
6    | (isLHS line (fname f)) = do -- LHS
7      let (x, (var, ops)) = statement (strip line)
8          v = fromJust x
9          use = filter (not.null) (findUse v nextCont)
10         operands = filter (not.isInfixOf "fn") (filter (not.isInfixOf "bb") $ filter (↩
               isPrefixOf str_var) (words $ unwords ops))
11         vList = variables f
12
13     case (isInfixOf "_init" line || isInfixOf "_ptr" line || elem v reg_base) of
14       True -> do
15         let v' =  fromJust $ lookupList v vList
16             newList = rmVariable v' vList []
17         variableElim True (f{ variables = newList }) nextCont preCont
18
19       _ -> do
20         if (null use)
21           then do -- DEAD Variable
22             if (isNothing $ lookupList v vList)
23               then (variableElim True f nextCont preCont)
24               else do
25                 let v' = fromJust $ lookupList v vList
26                     newList = rmVariable v' vList []
27                 variableElim True (f{ variables = newList }) nextCont preCont
28
29           else if (hasNoDef operands vList)
30             then do-- LIVE Variable but NoDef(ops)
31               let v' = fromJust $ lookupList v vList
32                   newList = rmVariable v' vList []
33               variableElim True (f{ variables = newList }) nextCont preCont
34             else -- LIVE Variable and Def(ops)
35               variableElim change f nextCont (preCont ++ [line])
36
37   | otherwise= do
38     -- No LHS
39     let (na, (var, reg)) = statement (strip line)
40         operands = filter (not.isInfixOf "fn") (filter (not.isInfixOf "bb") $ filter (↩
               isPrefixOf str_var) (words $ unwords reg))
41         vList = variables f
42
43     if (op var == "store")
44       then do
45         if (isInfixOf "_init" line || isInfixOf "_ptr" line)
46           then variableElim True f nextCont preCont
47
48           else if (hasNoDef operands vList)
49             then variableElim True f nextCont preCont -- NoDef(reg)
50             else variableElim change f nextCont (preCont ++ [line]) -- Def(reg)
51
52         else if (hasNoDef operands vList)
53           then variableElim True f nextCont preCont -- NoDef(reg)
54           else variableElim change f nextCont (preCont ++ [line]) -- Def(reg)
```

Haskell Code A.5: Source Code of Elimination