International Journal of Control and Automation
Vol. 10, No. 10 (2017), pp.123-134
http//dx.doi.org/10.14257/ijca.2017.10.10.11

Design and Implementation of the RSIL to LLVM IR Translator for Verification of the Intermediate Code on IoT Virtual Machine

Yunsik Son[1], Seman Oh[1] and YangSun Lee[2]*

[1]*Dept. of Computer Engineering, Dongguk University*
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA
{sonbug, smoh}@dongguk.edu
[2]*Dept. of Computer Engineering, Seokyeong University*
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
**yslee@skuniv.ac.kr*

Abstract

Internet of Things (IoT) refers to the technology that connects sensors to the Internet by incorporating sensors and communication functions into various objects. Recently, such IoT devices are used in various fields such as defense, finance, and manufacturing, and virtual machines that are operated in IoT devices are being developed in order to develop IoT applications using a programming language that has been used in the past. In this paper, we proposed the RSIL (Reduced Smart Intermediate Language) to LLVM IR code translator, which can convert RSIL to LLVM IR for verification of RSIL, the executable code of LWVM (Light Weight Virtual Machine) on IoT environments. Using proposed translator, we verified the operational semantics of RSIL by comparison the same execution result as RSIL execution result.

***Keywords**: IoT, Virtual Machine, Intermediate Language, Code Translator, Code Verification, LLVM(Low-Level Virtual Machine), RSIL(Reduced Smart Intermediate Language), LWVM(Light Weight Virtual Machine)*

1. Introduction

The Internet of Things (IoT) refers to the technology that connects sensors to the Internet by incorporating sensors and communication function[1]s into various objects [1]. It is a technology that allows Internet connected objects to exchange data to provide users with appropriate information and services, or to provide various application services through big data analysis.

Recently, such IoT devices are used in various fields such as defense, finance, and manufacturing, and virtual machines that are operated in IoT devices are being developed in order to develop IoT applications using a programming language that has been used in the past [2, 3].

In this paper, we propose a RSIL [4, 5] to LLVM IR code [6, 7] translator to verify the IoT-Cloud VM (LWVM, Light-weighted Virtual Machine) [5] and RSIL, an intermediate code executed on the IoT VM, using the existing LLVM interpreter.

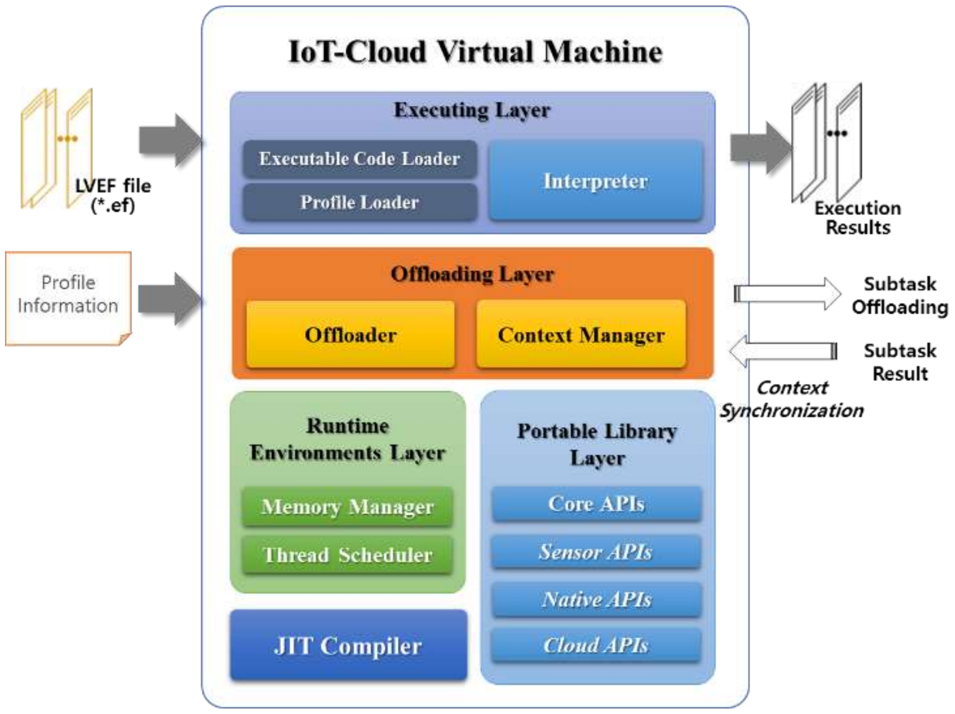
* Corresponding Author

2. Related Works

2.1. LWVM (Light-Weighted Virtual Machine)

The IoT-Cloud VM is a stack-based virtual machine designed to run on IoT devices under low computational performance. Figure 1 shows the system configuration of the IoT-Cloud VM [5].

Figure 1. System Model of Smart Cross Platform



The Light-Weighted IoT Virtual Machine is a stack-based virtual machine that was designed to execute on low computing powered IoT devices by the cloud-based offloading method. Moreover, it was designed for small computing devices that have restricted resources. The system configuration of the proposed Light-Weighted IoT Virtual Machine is comprised of the execution layer, offloading layer, runtime environment layer, portable library layer, and JIT (Just-In-Time) compiler.

2.2. IoT-Cloud Fusion Virtual Machine System

The IoT-Cloud fusion virtual machine system is designed to support downloading and executing application programs without platform dependency on various IoT devices [5].

The system consists of four main parts; compiler, assembler, profiler and IoT-Cloud virtual machine. It is designed in a hierarchal structure to minimize the burden of the retargeting process. Figure 2 shows a system configuration of the IoT-Cloud fusion virtual machine system.

The IoT-Cloud Virtual Machine is a stack based virtual machine solution with cloud offloading technology, loaded on IoT devices, which allows dynamic application programs to be downloaded and run platform independently with high computing performance. The VM is designed to use an intermediary language, RSIL (Reduced Smart Intermediate Language), which is capable of accommodating both procedural and objectoriented languages. It has the advantage of accommodating languages such as C/C++, Java, and Objective-C used in the iOS, which are currently used by a majority of developers.

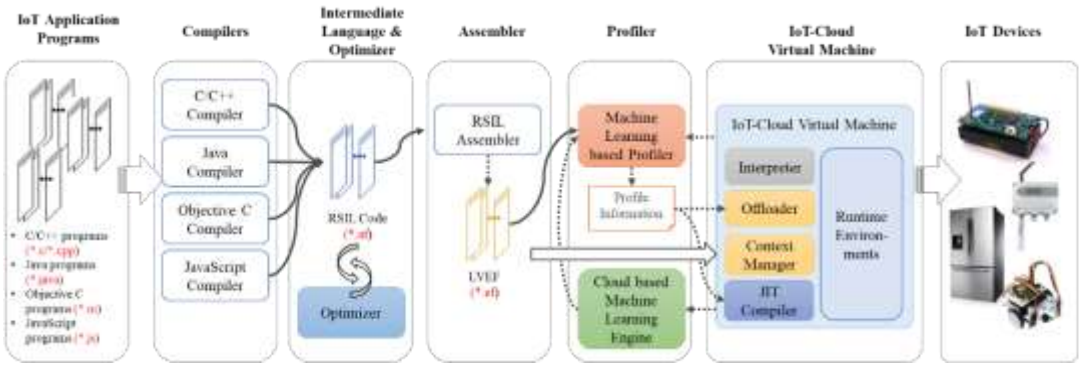


Figure 2. System Configuration of the IoT-Cloud Light-weighted Virtual Machine(LWVM)

The LWVM system consists of a compiler that compiles the application and generates an LVAF (Light-weighted Virtual machine Assembly Format) file composed of RSIL code, an assembler that converts the LVAF file to LVEF (Light-weighted Virtual machine Executable Format), and the IoT-Cloud VM that accepts and executes the LVEF file.

2.3. RSIL (Reduced Smart Intermediate Language)

We have defined the RSIL by decreasing the number of SIL instruction sets to 188 using upper criteria, and reducing the size of the virtual machine instruction set for IoT devices by the 1byte level. Also, optimization codes that specially designed for IoT devices could be added on the RSIL's unused remind instruction space. Table 1 shows the virtual machine code category information of RSIL. Table 2 shows the major pseudo codes of RSIL.

The major pseudo code of RSIL in Table 2 means that RSIL is segmented by functions represented in the code section. Each function has its own function name, parameter information and opcode list denoted by mnemonics. The mnemonic of pseudo codes improve readability by using the keyword at the source code level, and defining indicators as functions.

Table 1. Virtual Machine Code Category Information of RSIL

Operator Category	Count
Stack manipulation	49
Arithmetic operator	86
Flow control	14
Type conversion	22
Optimization	17
Total	188

2.4. LLVM (Low-Level Virtual Machine)

LLVM [6] is a compiler infrastructure that allows a program to easily implement optimization regardless of the programming language in compile time, link time, and runtime. With LLVM, it is possible to compile the code statically, and it is also possible to compile it into the LLVM IR format, which is an intermediate code that is compiled once more into the machine language during execution using the LLVM interpreter.

Table 2. Major Pseudo Codes of RSIL

Pseudo Code	Description
%%CodeSectionStart	Section flag for starting code area.
%FunctionStart	Section flag for starting function area.
.func_name	Describe the function name.
.param_count	Describe the number of parameters for target function.
.opcode_start	Declares opcode list for corresponding function.
.opcode_end	Declares end of opcode list for the corresponding function.
%FunctionEnd	Section flag for finishing function area.
%%CodeSectionEnd	Section flag for finishing code area.

3. RSIL to LLVM IR Translator

RSIL to LLVM IR Translator reads the RSIL code information of the LVAF through the LVAF Loader. Next, the table manager configures the data table with information about the program variables, constants, and functions. Finally, all of the information is translated to LLVM IR code by the code writer. Figure 3 shows the overall code translate flow for the proposed RSIL to LLVM IR translator.

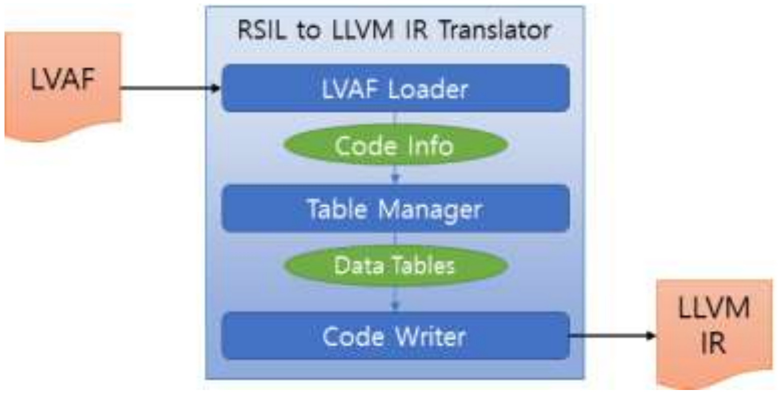


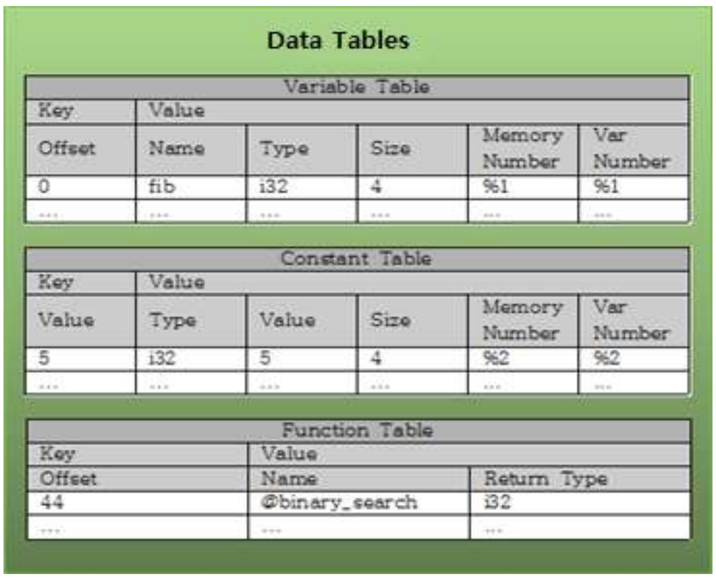
Figure 3. RSIL to LLVM IR Translator Model

The code writer as a core module of the proposed translator has four processing steps: header generation, code generation, function declaration, and function attribute declaration. First, in the header generation, the data of the literal table created by the table manager is read and converted into a literal form used in the LLVM IR. Second, in the code generation phase, we refer to three data tables and convert the RSIL instruction to the LLVM IR instruction on a function-by-function basis. Third, in the function declaration phase, the program creates declarations for functions that are called but not defined internally. Finally, in the function attribute declaration step, the attributes of the function are identified through the commands used in the function, and then the attribute group is created.

3.1. LVAF Loader & Table Manager

The LVAF Loader is a component that loads the entire code information such as the number of functions, the function name, the number of instructions in the function, and the label name into the memory by reading the LVAF file divided into the header area, the code area and the data area.

The table manager configures and manages data tables in the form of hash maps with the information necessary to convert the RSIL code into the LLVM IR code. The data tables are divided into three types: variable tables, constant tables, and function tables. Figure 4 shows an example of the data tables, a constant table, and a function table managed by the table manager.



Data Tables

Variable Table					
Key	Value				
Offset	Name	Type	Size	Memory Number	Var Number
0	fib	i32	4	%1	%1
...	...	...	...	...	...

Constant Table					
Key	Value				
Value	Type	Value	Size	Memory Number	Var Number
5	i32	5	4	%2	%2
...	...	...	...	...	...

Function Table		
Key	Value	
Offset	Name	Return Type
44	@binary_search	i32
...	...	...

Figure 4. Example of the Data Tables while RSIL to LLVM IR Translation

The table manager records the name of the function in the function table when the function is started in the translation process, and records the return type of the function when the function terminates. Next, using the RSIL instructions read from the LVAF file, the type and size of variables or constants with corresponding offsets are inferred and recorded in a variable table or a constant table.

Table 3 below compares the types used in RSIL and LLVM IR. It can be confirmed that seven commonly used types are matched in each intermediate language, and intermediate code conversion is performed using this information.

Table 3. Type Comparison Table for RSIL with LLVM

Type (signed/unsigned)	RSIL	LLVM IR
Char	c / uc	i8
Short	s / us	i16
Int	i / ui	i32

Long	l / ul	i64
Float	f	float
Double	d	double
Pointer	p	{Type}*

3.2. Code Generator

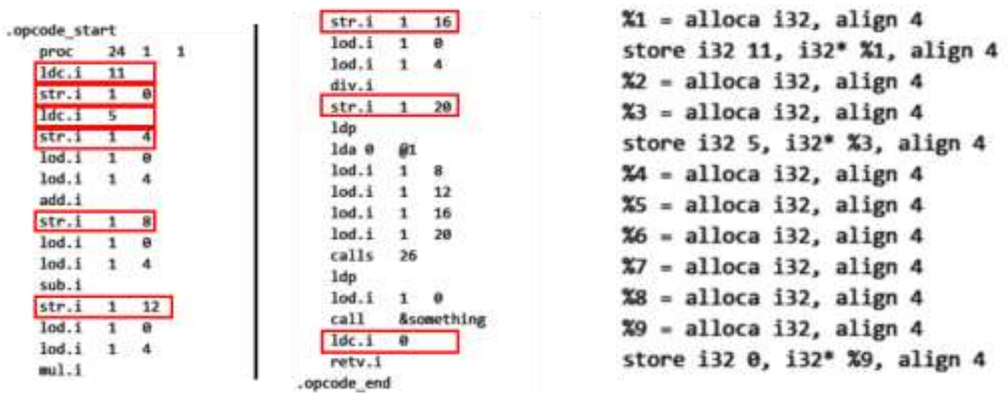
The translation phase of the code generator consists of four steps: header, code, function declaration, and function attribute declaration processing.

First, at the header stage, the data of the literal table generated by the table manager is read and converted into a literal form used in the LLVM IR. Second, in the code phase, we refer to three data tables and convert the RSIL instruction to the LLVM IR instruction on a function-by-function basis. Third, at the function declaration stage, the program creates declarations for functions that are called but not defined internally. Finally, in the function attribute declaration step, the attribute of the function is identified through the commands used in the function, and then the attribute group is created.

Figures 5 and 6 below show the converted result of the literal section and the converted result of the code section, respectively. In Figure 5, the upper picture is a literal table in RSIL, and the lower one is the converted header information of LLVM IR. In Figure 6, the left side is the code information in the function of RSIL, and the right side is the converted code information of LLVM IR.

```
%LiteralTableStart
    .literal_start  @0  0   9
        0x61,0x20,0x3d,0x20,0x25,0x64,0x5c,0x6e,0x00
    .literal_end
    .literal_start  @1  0   14
        0x25,0x64,0x20,0x25,0x64,0x20,0x25,0x64,0x20,0x25,0x64,0x5c,0x6e,0x00
    .literal_end
%LiteralTableEnd
```

Figure 5. Translation Example for Literal Section



```
.opcode_start
    proc    24  1   1
    ldc.i   11
    str.i   1   0
    ldc.i   5
    str.i   1   4
    lod.i   1   0
    lod.i   1   4
    add.i
    str.i   1   8
    lod.i   1   0
    lod.i   1   4
    sub.i
    str.i   1   12
    lod.i   1   0
    lod.i   1   4
    mul.i
```

```
    str.i   1   16
    lod.i   1   0
    lod.i   1   4
    div.i
    str.i   1   20
    ldp
    lda 0   @1
    lod.i   1   8
    lod.i   1   12
    lod.i   1   16
    lod.i   1   20
    calls   26
    ldp
    lod.i   1   0
    call    &something
    ldc.i   0
    retv.i
.opcode_end
```

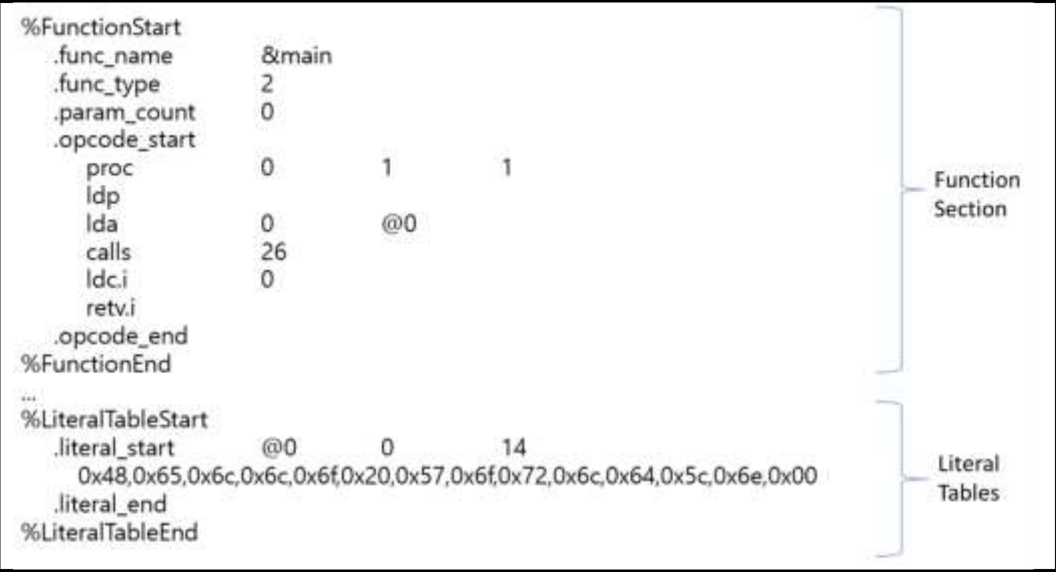
```
%1 = alloca i32, align 4
store i32 11, i32* %1, align 4
%2 = alloca i32, align 4
%3 = alloca i32, align 4
store i32 5, i32* %3, align 4
%4 = alloca i32, align 4
%5 = alloca i32, align 4
%6 = alloca i32, align 4
%7 = alloca i32, align 4
%8 = alloca i32, align 4
%9 = alloca i32, align 4
store i32 0, i32* %9, align 4
```

Figure 6. Translation Example for Code Section

Table 4 shows the program structure of RSIL and the program structure of LLVM IR generated by using code generator.

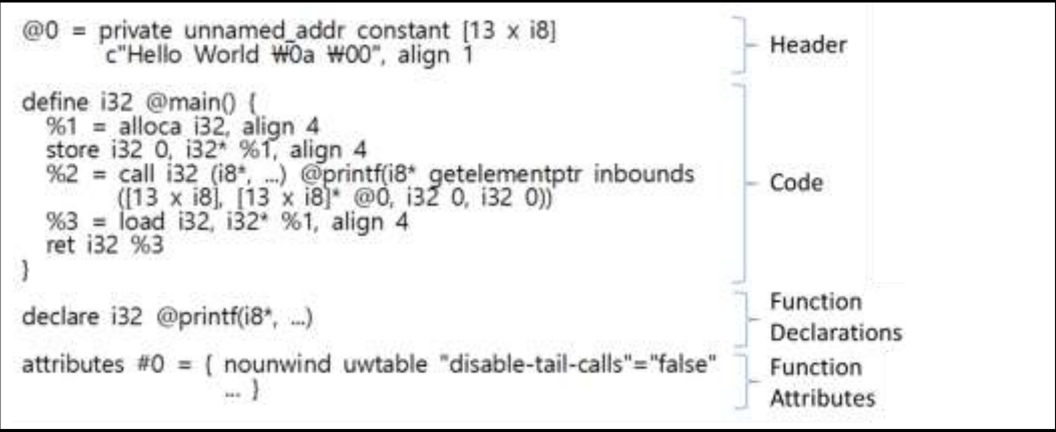
Table 4. A Mapping Example for Program Structures of the RSIL and the LLVM IR

Structure of the RSIL Code



```
%FunctionStart
    .func_name          &main
    .func_type          2
    .param_count        0
    .opcode_start
        proc            0           1           1
        ldp
        lda             0           @0
        calls           26
        ldc.i           0
        retv.i
    .opcode_end
%FunctionEnd
...
%LiteralTableStart
    .literal_start      @0          0           14
      0x48,0x65,0x6c,0x6c,0x6f,0x20,0x57,0x6f,0x72,0x6c,0x64,0x5c,0x6e,0x00
    .literal_end
%LiteralTableEnd
```

Structure of the LLVM IR Code



```
@0 = private unnamed_addr constant [13 x i8]
        c"Hello World ₩0a ₩00", align 1

define i32 @main() {
   %1 = alloca i32, align 4
   store i32 0, i32* %1, align 4
   %2 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
           ([13 x i8], [13 x i8]* @0, i32 0, i32 0))
   %3 = load i32, i32* %1, align 4
   ret i32 %3
}

declare i32 @printf(i8*, ...)

attributes #0 = { nounwind uwtable "disable-tail-calls"="false"
                   ... }
```

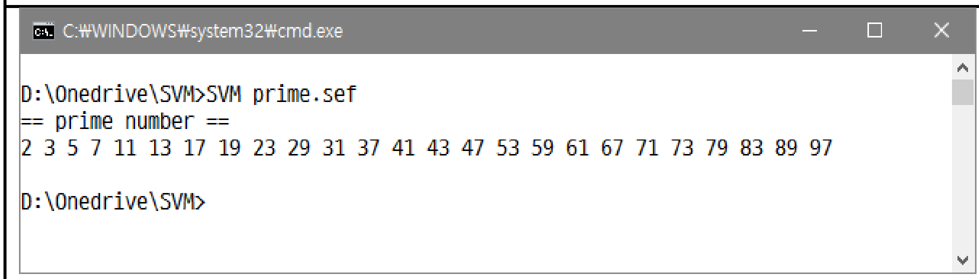
4. Experimental Results

The RSL code is converted into the LLVM IR code using the LLVM code converter implemented in this paper. The converted LLVM IR code can be executed using the LLVM interpreter on the Linux operating system. Table 5 shows the core algorithm code of the program that obtains decimal numbers written in C language, the RSIL code generated by the RSIL compiler, the LLVM IR code converted by the RSIL to LLVM IR converter implemented in this paper, and the execution result.

Table 5. Experimental Results for Example Code Translation

C Source Code
```
…
int isPrime(int number) {
    int i, j;
    i = 2;
    j = number / 2;
    while (i <= j) {
        if (number % i == 0)
            return 0;
        i++;
    }
    return 1;
}
…
```
Translated RSIL Code from C Source Code
```
…
%FunctionStart
            .func_name      &isPrime
            .func_type      2
            .param_count    1
            .opcode_start
                        proc     12      1       1
                        str.i    1       0
                        ldc.i    2
                        str.i    1       4
                        lod.i    1       0
                        ldc.i    2
                        div.i
                        str.i    1       8
                       ... omitted ...
                        retv.i
            %Label ##2
                        lod.i    1       4
                        ldc.i    1
                        add.i
                        str.i    1       4
                        ujp      ##0
            %Label ##1
                        ldc.i    1
                        retv.i
            .opcode_end
%FunctionEnd
…
```

Execution Result of the Translated RSIL Code



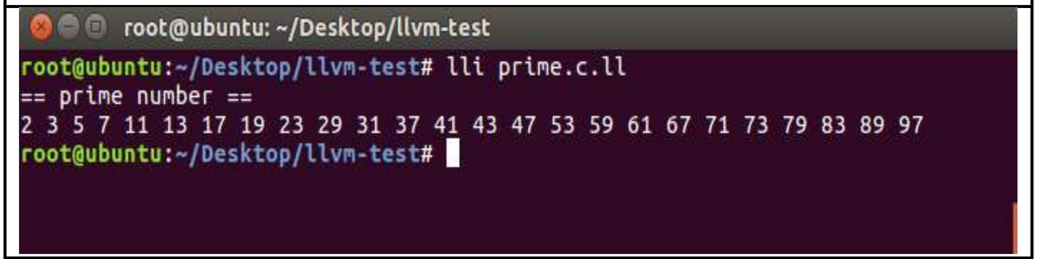
```
D:\Onedrive\SVM>SVM prime.sef
== prime number ==
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

D:\Onedrive\SVM>
```

Generated LLVM IR Code from RSIL Code

```
…
define i32 @isPrime (i32 %v1)  {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 2, i32* %3, align 4
  ... omitted ...
  %20 = load i32, i32* %6, align 4
  %21 = icmp eq i32 %19, %20
  br i1 %21, label %22, label %25
  %23 = load i32, i32* %6, align 4
  ret i32 %23
  br label %25
  %26 = load i32, i32* %4, align 4
  %27 = load i32, i32* %7, align 4
  %28 = add i32 %26, %27
  store i32 %28, i32* %4, align 4
  br label %12
  br label %30
  %31 = load i32, i32* %7, align 4
  ret i32 %31
}
…
```

Execution Result of the Generated LLVM IR Code



```
root@ubuntu:~/Desktop/llvm-test# lli prime.c.ll
== prime number ==
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
root@ubuntu:~/Desktop/llvm-test#
```

The table mainly shows the conversion process of the core code, and it is confirmed that the function of the original C source program is converted into RSIL and executed, and the result of RSIL is converted into LLVM IR again. This allows verification of the RSIL code generator.

5. Conclusions and Further Researches

The Internet of things is a technology that connect the internet with sensors and communication functions on various objects. Our research team is studying the common development / execution environment that can be used in various internet devices by applying virtual machine technology to the internet.

RSIL is an intermediate language for virtual machines used in these objects on the Internet. Verifying the validity of RSIL is a very important task in the constructing virtual machine based IoT execution system.

In this paper, we propose RSIL to LLVM IR translator, convert RSIL code to LLVM IR code, and verify intermediate code by comparing RSIL and LLVM IR code through LWVM and LLVM interpreter respectively. As a further study, we will analyze the RSIL code by combining the RSIL to LLVM IR Translator with the static analysis and visualization tools provided by the LLVM project.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (No. 2016R1A2B4008392).

References

[1] C. Michael, L. Markus and R. Roger, “The Internet of Things”, McKinsey Reports Quarterly, **(2014)**.
[2] Y.S. Lee, S.M. Oh and Y.S. Son, “Design and Implementation of HTML5 based SVM for Integrating Runtime of Smart Devices and Web Environments”, International Journal of Smart Home, SERSC, vol.8, no.3, **(2014)**, pp. 223-234.
[3] Y.S. Lee and Y.S. Son, “A Study on the Smart Virtual Machine for Smart Devices”, Information -an International Interdisciplinary Journal, International Information Institute, vol. 16, no. 2, **(2013)**, pp. 1465-1472.
[4] Y. S. Lee, J. Jeong and Y. Son, “Design and implementation of the secure compiler and virtual machine for developing secure IoT services”, Future Generation Computer Systems, no. 76, **(2016)**, pp. 350-357.
[5] Y. Son and Y. S. Lee, “A Study on the Interpreter for the Light-Weighted Virtual Machine on IoT Environments,” International Journal of Web Science and Engineering for Smart Device, vol. 3, no. 2, **(2016)**, pp. 19-24.
[6] LLVM Projects, http://llvm.org/ProjectsWithLLVM.
[7] LLVM Compiler Infrastructure, http://llvm.org.
[8] Y. Son, Y. S. Lee, “Offloading Method for Efficient Use of Local Computational Resources in Mobile Location-Based Services Using Clouds,” Mobile Information Systems, vol. 2017, **(2017)**, pp. 1-10.
[9] P. G. Vijayrajan, “Analysis of Performance in the Virtual Machines Environment”, International Journal of Software Engineering and Its Applications, vol.32, **(2011)**, pp. 53-64.
[10] J. E. Smith, R. Nair, Virtual Machines, Morgan Kaufmann, **(2005)**.
[11] Y. S. Lee and Y. Son, “A Study on the Smart Virtual Machine for Executing Virtual Machine Codes on Smart Platforms”, International Journal of Smart Home, vol. 6, no. 4, **(2012)**, pp. 93–105.
[12] D. Werth, A. Emrich and A. Chapko, “An ecosystem for user-generated mobile”, Journal of Coverage, vol. 3, no. 4, **(2012)**, pp.

Authors

Yunsik Son, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 2004, and M.S. and Ph.D. degrees from the Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 2006 and 2009, respectively. Currently, he is a Researcher of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, secure software, programming languages, compiler construction, and mobile/embedded systems.

Seman Oh, he received the B.S. degree from the Seoul National University, Seoul, Korea, in 1977, and M.S. and Ph.D. degrees from the Dept. of Computer Science, Korea Advanced Institute of Science and Technology, Seoul, Korea in 1979 and 1985, respectively. He was a Dean of the Dept. of Computer Science and Engineering, Graduate School, Dongguk University from 1993-1999, a Director of SIGPL in Korea Institute of Information Scientists and Engineers from 2001-2003, a Director of SIGGAME in Korea Information Processing Society from 2004-2005. Currently, he is a Professor of the Dept. of Computer Science and Engineering, Dongguk University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.

Yangsun Lee, he received the B.S. degree from the Dept. of Computer Science, Dongguk University, Seoul, Korea, in 1985, and M.S. and Ph.D. degrees from Dept. of Computer Engineering, Dongguk University, Seoul, Korea in 1987 and 2003, respectively. He was a Manager of the Computer Center, Seokyeong University from 1996-2000, a Director of Korea Multimedia Society from 2004-2005, a General Director of Korea Multimedia Society from 2005-2006 and a Vice President of Korea Multimedia Society in 2009. Also, he was a Director of Korea Information Processing Society from 2006-2010 and a President of a Society for the Study of Game at Korea Information Processing Society from 2006-2010. And, he was a Director of Smart Developer Association from 2011-2012. Currently, he is a Professor of Dept. of Computer Engineering, Seokyeong University, Seoul, Korea. His research areas include smart system solutions, programming languages, and embedded systems.