IoT 가상기계의 중간코드 검증을 위한 RSIL to LLVM IR 변환기의 설계 및 구현

조재현*, 최찬휘*, 손윤식**, 이양선*
*서경대학교 컴퓨터공학과
**동국대학교 컴퓨터공학과
e-mail:cjh7163@skuniv.ac.kr

A Study on RSIL to LLVM IR Translator For Verification of Intermediate Code on IoT Virtual Machine

Jaehyeon Cho*, Chanwhi Choi*, Yunsik Son**, YangSun Lee*
*Dept of Computer Engineering, SeoKyeong University
**Dept of Computer Engineering, Dongguk University

요 약

본 논문은 IoT 장치에서 동작하는 가상기계의 중간코드를 LLVM IR 코드로 변환하는 코드 변환기를 설계 및 구현하였다. LLVM 인터프리터를 통해 RSIL 코드로부터 변환된 LLVM IR 코드를 실행하고, 그 결과를 비교하여 IoT 가상기계의 중간코드를 검증하였다.

1. 서론

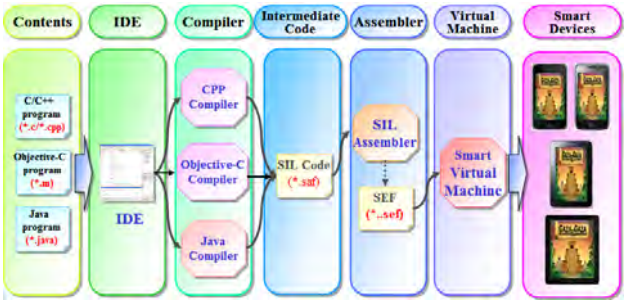
IoT(Internet of Things, 사물인터넷) [1]는 각종 사물에 센서와 통신 기능을 내장하여 인터넷에 연결하는 기술을 의미한다. 인터넷으로 연결된 사물들이 데이터를 주고받아 스스로 분석하고 학습한 정보를 사용자에게 제공하거나 사용자가 이를 원격 조정할 수 있는 인공지능 기술이다. 최근에 이러한 IoT 장치들이 국방, 금융, 제조 등 여러 분야에 걸쳐 이용되고 있으며, 기존에 사용하던 프로그래밍 언어를 이용하여 IoT 애플리케이션을 개발 할 수 있도록 하기 위해 IoT 장치에서 동작하는 가상기계도 개발되고 있다.

본 논문은 IoT 가상기계인 LWVM과 그것을 통해 실행되는 중간코드인 RSIL을 기존의 LLVM 인터프리터를 이용하여 검증하기 위해 RSIL to LLVM IR 코드 변환기를 구현하였다.

2. 관련연구

2.1 스마트 크로스 플랫폼

스마트 크로스 플랫폼(Smart Cross Platform)은 본 연구팀이 이전에 개발한 스마트 기기를 위한 가상기계 기반의 플랫폼이다 [2]. 스마트 크로스 플랫폼은 중간 언어를 사용하는 가상기계 기반의 플랫폼으로 여러 스마트기기에 탑재되어 플랫폼 독립적으로 수행이 가능하며, 중간언어인 SAF(Smart Assembly Format)는 순차적 언어인 C와 객체지향 언어인 C++과 Java를 모두 수용한다. 그림 1은 스마트 크로스 플랫폼의 모델을 나타낸 것이다.



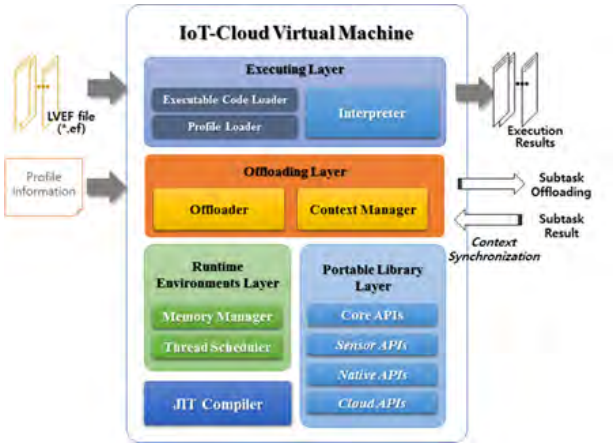
(그림 1) 스마트 크로스 플랫폼의 모델

스마트 크로스 플랫폼은 크게 응용프로그램을 컴파일하여 SIL(Smart Intermediate Language) 코드로 구성된 SAF 형식의 파일을 생성하는 컴파일러, SAF 파일을 실행 포맷인 SEF(Smart Executable Format)로 변환하는 어셈블러, SEF 형식의 파일을 입력 받아 실행하는 스마트 가상기계(SVM, Smart Virtual Machine)의 세 부분으로 구성된다.

2.2 LWVM(Light-Weighted Virtual Machine)

LWVM(Light-Weighted Virtual Machine, 경량 가상기계) [3]은 낮은 계산 성능을 갖는 IoT 장치에서 실행하기 위해 설계된 스택 기반의 가상머신이다. 그림2는 경량 가상기계의 시스템 구성도를 나타낸 것이다.

"이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No.2016R1A2B4008392)"



(그림 2) 경량 가상기계의 시스템 구성도

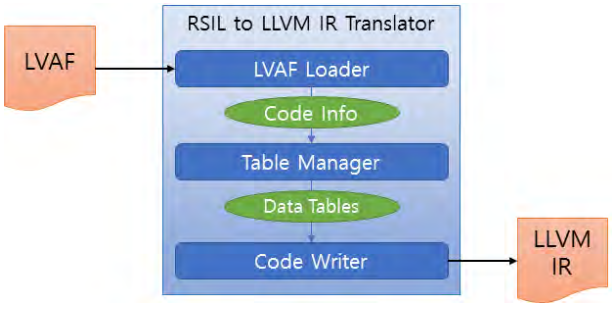
경량 가상기계 시스템은 응용프로그램을 컴파일하여 RSIL(Reduced Smart Intermediate Language) 코드로 구성된 LVAF 형식의 파일을 생성하는 컴파일러, LVAF 파일을 LVEF로 변환하는 어셈블러, LVEF 형식의 파일을 입력 받아 실행하는 경량 가상기계로 구성되어 있다.

2.3 LLVM(Low-Level Virtual Machine)

LLVM [4]은 프로그램을 컴파일 타임, 링크 타임, 런타임 상황에서 프로그램의 작성 언어에 상관없이 최적화를 쉽게 구현할 수 있도록 구성된 컴파일러의 기반구조이다. LLVM을 이용하면 코드를 정적으로 컴파일 하는 것이 가능하며, LLVM 인터프리터를 이용하여 실행 도중 기계어로 한 번 더 컴파일되는 중간코드인 LLVM IR 형식으로 컴파일 하는 것 또한 가능하다.

3. RSIL to LLVM IR Translator

RSIL to LLVM IR Translator는 LVAF Loader를 통해 LVAF의 RSIL(Reduced Smart Intermidate Language) 코드 정보를 읽어오고 이를 이용해 테이블 매니저를 통해서 프로그램의 변수, 상수, 함수에 대한 정보들로 데이터 테이블을 구성한 후 최종적으로 LLVM IR 코드를 생성한다. 그림3은 LLVM 코드 변환기의 전체적인 코드 변환 흐름을 보인 것이다.



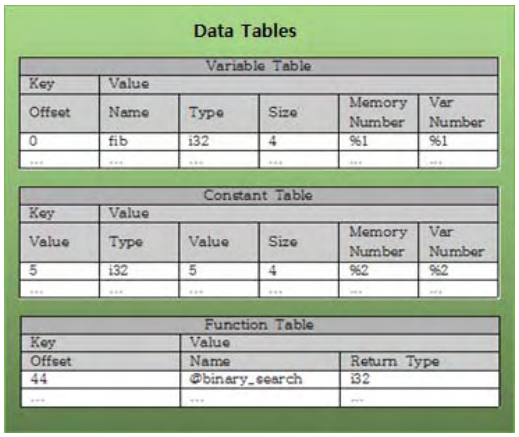
(그림 3) 전체적인 코드 변환 흐름

3.1 LVAF Loader

LVAF Loader는 헤더 영역, 코드 영역, 데이터 영역으로 나누어 작성된 LVAF 파일을 읽어 들여서 함수 개수, 함수이름, 함수 내의 명령어 개수, 레이블 이름 등의 전체적인 코드 정보를 메모리에 로드하는 컴포넌트이다.

3.2 테이블 매니저

테이블 매니저는 RSIL 코드를 LLVM IR 코드로 변환하는데 필요한 정보를 갖는 해시 맵 형태의 테이블들을 구성, 관리한다. 테이블은 변수 테이블, 상수 테이블, 함수 테이블 세 가지로 나뉜다. 그림4는 테이블 매니저가 관리하는 변수 테이블, 상수 테이블, 함수 테이블의 예를 보인 것이다.



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
...	...	...

(그림 4) 데이터 테이블 예

테이블 매니저는 변환 과정에서 함수가 시작될 때 함수 테이블에 함수의 이름을 기록하며, 함수가 종료될 때 함수의 반환 타입을 기록한다. 다음으로 LVAF 파일로부터 읽어 들인 명령어들을 이용하여 해당 오프셋을 갖는 변수 혹은 상수의 타입 및 크기를 추론하여 변수 테이블 혹은 상수 테이블에 기록한다.

3.3 코드 생성기

코드 생성기의 변환 단계는 헤더, 코드, 함수 선언, 함수 애트리뷰트 선언으로 총 4개의 단계로 구성되어 있다. 첫 번째로 헤더 단계에서는 테이블 매니저가 생성한 리터럴 테이블의 데이터들을 읽어 LLVM IR에서 쓰이는 리터럴의 형태로 변환한다. 두 번째로 코드 단계에서는 세 가지 데이터 테이블을 참조하여 함수 단위로 RSIL 명령어를 LLVM IR 명령어로 변환한다. 세 번째로 함수 선언 단계에서는 프로그램에서는 호출은 되었으나 내부에 정의가 되지 않은 함수의 선언을 작성한다. 마지막으로 함수 애트리뷰트 선언 단계는 함수 내에서 사용되는 명령어들을 통하여 함수의 특성을 파악한 후 애트리뷰트 그룹을 작성한다. 표1은 RSIL 코드와 LLVM 코드 구조를 보인 것이다.

<표 1> RSIL 코드와 LLVM IR 코드 구조

RSIL 코드 구조

```
%FunctionStart                                          ┐
   .func_name        &main                              │
   .func_type        2                                  │
   .param_count      0                                  │
   .opcode_start                                        │
      proc           0        1        1               │ 함수
      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 │ 리터럴
   .literal_end                                         │ 테이블
%LiteralTableEnd                                        ┘
```

LLVM IR 코드 구조

```
@0 = private unnamed_addr constant [13 x i8]            ┐ 헤더
       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 │ 코드
       ([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*, ...)                           ┐ 함수선언

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

4. 실험결과

본 논문에서 구현한 LLVM 코드 변환기를 이용하여 RSIL 코드를 LLVM IR 코드로 변환한다. 변환된 LLVM IR 코드는 Linux 운영체제 상에서 LLVM 인터프리터를 이용해 실행할 수 있다. 표2는 C언어로 작성한 소수를 구하는 프로그램의 핵심 알고리즘의 코드와 RSIL 컴파일러에 의해 생성된 RSIL 코드, 그리고 본 논문에서 구현한 RSIL to LLVM IR 변환기에 의해 변환된 LLVM IR 코드와 실행결과를 보인 것이다.

<표 2> C Source / RSIL / LLVM IR 코드 및 실행 결과

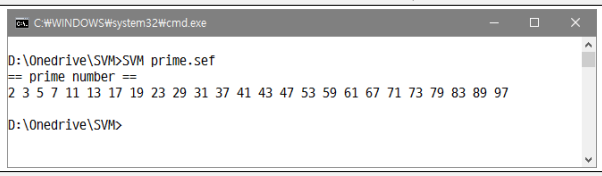
C 소스 코드

```
int isPrime(int number) {
    int i, j;
    i = 2;
    j = number / 2;
    while (i <= j) {
        if (number % i == 0)
            return 0;
        i++;
    }
    return 1;
}
```

C 소스코드로부터 변환된 RSIL 코드

```
%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
            ... 중간 생략 ...
                  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
```

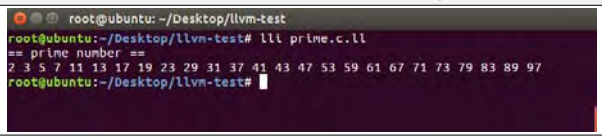
RSIL 코드 실행 결과



RSIL 코드로부터 변환된 LLVM IR 코드

```
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
    ... 중간 생략 ...
  %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
}
```

LLVM IR 코드 실행 결과



5. 결론 및 향후연구

본 연구팀은 RSIL to LLVM IR Translator를 이용하여 RSIL 코드를 LLVM IR 코드로 변환하였고, RSIL 코드와 LLVM IR 코드를 각각 LWVM과 LLVM 인터프리터를 통해 실행하여 결과를 비교함으로써 중간 코드를 검증하였다. 향후 RSIL to LLVM IR Translator와 LLVM 프로젝트 [5]에서 제공하는 정적 분석 도구 및 시각화 도구를 결합하여 RSIL 코드를 분석하는 연구를 진행할 예정이다.

참고 문헌

[1] Chui Michael, Löffler Markus, Roberts Roger, “The Internet of Things,” McKinsey Quarterly, 2014.

[2] 한성민, 손윤식, 이양선, “스마트 크로스 플랫폼을 위한 스마트 가상기계의 설계 및 구현,” 한국멀티미디어학회 논문지, Vol. 16, No. 2, pp. 190-197, 2013.

[3] Yunsik Son, YangSun 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, pp. 19-24, 2016

[4] The LLVM Compiler Infrastructure, http://llvm.org

[5] LLVM Projects, http://llvm.org/ProjectsWithLLVM

[6] Qiu, Lili, Programming Language Translation, Cornell University, technical report, 1999.