제22회 한국정보처리학회 추계학술발표대회 논문집 제11권 제2호 (2004. 11)

실행 파일 포맷 생성기의 설계 및 구현

손윤식*, 오세만
동국대학교 컴퓨터공학과
e-mail : {sonbug*, smoh}@dongguk.edu

Design and Implementation of Executable File Format Generator

Yunsik Son*, Seman Oh
Dept. of Computer Engineering, Dongguk University

요 약

EVM(Embedded Virtual Machine)은 임베디드 시스템을 위한 가상기계로서 플랫폼에 독립적이며, 모바일 디바이스와 디지털 TV 등에 탑재할 수 있는 핵심 기술로 다운로드 솔루션에서는 꼭 필요한 기술이다. SIL(Standard Intermediate Language)은 임베디드 시스템을 위한 가상기계의 표준 중간 언어로 객체지향 프로그래밍 언어와 순차 적인 프로그래밍언어를 모두 수용할 수 있다. SIL 로 기술된 프로그램이 EVM 에서 실행되기 위해서는 EFF(Executable File Format)형태로 변환되어야 한다. 임베디드 시스템을 위한 실행 파일 포맷인 EFF 는 구조가 간결하며 확장이 용이한 특징을 지닌다. 또한 메타데이터와 표준 중간 언어가 서로 독립적으로 구성되어 분석이 쉽고 타입 체크가 편리한 구조이다.

본 논문에서는 가상기계를 위한 표준 중간 언어인 SIL 을 EVM 에서 실행 가능한 형태로 변환시켜주는 실행 파일 포맷 생성기(EFF Generator)를 설계하고 구현한다. 먼저, SIL 구조를 위한 SIL 문법을 설계하고 파서 생성기(PGS)를 사용하여 SIL 프로그램을 위한 어휘분석기와 구문분석기를 구현한다. 그리고 AST 를 생성한 후, 포맷 생성기를 통하여 AST 를 운행하며 EFF 를 생성한다.

1. 서론

가상 기계란 프로세서나 운영체제 등이 바뀌더라도 응용프로그램이나 컨텐츠를 변경하지 않고 사용할 수 있는 기술로 특히, 임베디드 시스템을 위한 가상기계 기술은 모바일 디바이스와 디지털 TV 등에 탑재할 수 있는 핵심 기술로 다운로드 솔루션에서는 꼭 필요한 기술이다. 이러한 가상 기계의 예로는 PASCAL P-기계, JVM, GVM, EVM 등을 들 수 있으며, 각각 가상기계에 최적화된 실행 파일 포맷이 존재한다.

EVM 모델은 임베디드 시스템을 위한 가상기계 솔루션으로서 C#과 자바 바이트코드를 위한 번역기와 가상기계를 위한 표준 중간 언어 및 실행 파일 포맷 생성기(EFF Generator), 임베디드 시스템을 위한 가상기계 등 크게 3 부분으로 구성된다. C#, 자바 등 객체지향 언어뿐만 아니라 C 언어와 같이 순차적인 언어로 작성된 프로그램을 어셈블리 언어 형태인 *. sil 을 거쳐서 EVM 파일 포맷으로 변환하여 임베디드 시스템에 탑재된 가상기계에서 실행할 수 있도록 한다[8].

본 논문에서는 EVM 의 어셈블리 언어인 표준 중간 언어(SIL)을 EVM 에서 실행 가능한 실행 파일 포맷(EFF)으로 변환하는 실행 파일 포맷 생성기를 설계하고 구현한다.

본 연구는 한국과학재단 목적기초연구(R01-2002-000-00041-0)지원으로 수행되었음.

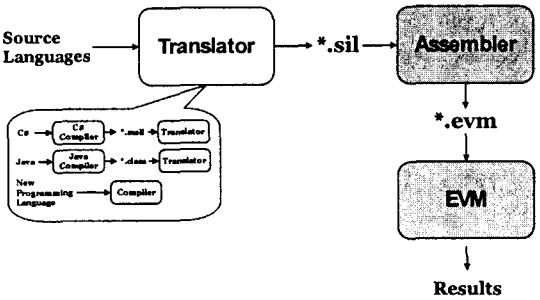
2. 배경 연구

2.1 EVM

EVM 은 모바일 장치, 셋톱 박스, 디지털-TV 등에 탑재되어 동적 응용프로그램을 다운로드 하여 실행할 수 있는 가상기계 솔루션이다. 또한 컨텐츠 개발을 쉽게 하기 위해서 다양한 언어를 지원하고 언어들 간의

통합이 가능하다.

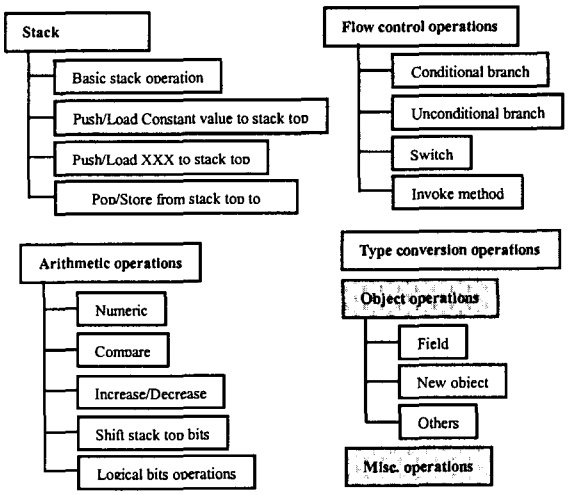
EVM 은 크게 세 부분으로 나뉘어진다. 첫 번째 부분은 C# 또는 자바 등의 고급 프로그래밍 언어로 작성된 프로그램을 가상 기계를 위한 SIL 로 번역하는 부분이다. 두 번째 부분은 SIL 코드들을 입력으로 하여 가상기계에서 실행 가능 한 형태인 EFF 로 변환하는 어셈블러 부분이다. 마지막으로 세 번째 부분은 실제 하드웨어에 탑재되어 EFF 를 실행하는 가상기계 부분이다. EVM 의 가상 기계 부분은 계층적인 구조로 설계되어 retargeting 과정의 부담을 최소화 한다. 이와 같은 EVM 의 시스템 구성도는 [그림 1]과 같다.



[그림 1] EVM 시스템 구성도

2.2 SIL

EVM의 어셈블리 언어인 SIL은 기존의 가상기계 어셈블리 언어들의 표준화 모델로 설계하였다. SIL은 클래스 선언 등 특정 작업의 수행을 나타내는 의사코드와 실제 명령어에 대응되는 연산코드로 이루어져 있다. SIL의 연산 코드는 다음과 같이 6개의 연산 카테고리로 나누어진다[6].



[그림 2] SIL 연산코드 카테고리

2.3 EFF

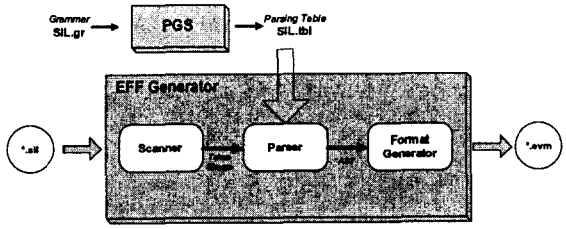
EVM 을 위한 실행 파일 형식인 EFF 는 가상 기계를 위한 *.sil 을 입력으로 받는 EFF generator 의 출력이다. 이는 다시 EVM 에 입력이 되어 실행 결과를 만들어 낸다. EFF 는 언어간에 통합을 기반으로 설계 되었으며 구조가 간결하고 확장이 용이하다. 또한 메타데이터와 SIL 이 분리되어 있어 파일 분석이 쉽고 타입 체크가 편리한 구조이다. EFF 는 header, metadata table 그리고 SIL 정보의 세 부분으로 이루어져 있다. Header 부분은 EFF 를 나타내는 magic number, 파일 포맷의 버전을 나타내는 version 정보, 그리고 소스파일을 나타내는 module name 으로 구성되어 있다. Metadata Table 부분은 tag 와 각 tag 의 속성들로 이루어져 있고 tag 로는 class, filed, method 그리고 string 이 있다. SIL 정보 부분은 메소드 table index 와 코드의 길이를 나타내는 length, 그리고 프로그램에 실행 정보를 나타내는 SIL code 들로 구성되어 있다[9].

3. 실행 파일 포맷 생성기의 설계

실행 파일 포맷 생성기는 SIL을 입력으로 받아서 EVM에서 실행 가능한 파일 포맷인 EFF를 생성한다. 이 장에서는 EFF의 전체적인 구조와 함께 SIL 문법의 소개 및 각 모듈과 자료구조에 대한 설명을 한다.

3.1 실행 파일 포맷 생성기의 구조

실행 파일 포맷 생성기는 크게 스캐너(Scanner), 파서(Parser), 포맷 생성기(Format Generator)로 구성되어 있으며, SIL 문법을 설계한 후 PGS 를 통하여 얻어진 정보를 이용하여 스캐너와 파서를 구현하였다. [그림 3]은 이와 같은 과정을 보여준다.



[그림 3] EFF Generator 구성도

SIL.gr 은 SIL 의 명령어를 context-free 문법으로 작성되었으며, 실행 파일 포맷 생성기에 필요한 어휘 정보와 파싱 테이블을 PGS 를 통행 생성하게 된다.

*.sil 을 입력으로 받은 실행 파일 포맷 생성기는 스캐너를 통하여 일련의 토큰으로 분할하여 파서에 전달한다. 파서는 스캐너에서 전달받은 토큰을 가지고 파싱 테이블을 참조하여 구문분석을 하면서

SDT(Syntax-Directed Translation) 방식으로 AST 를 생성한다. 포맷 생성기는 파서가 생성한 AST 를 탐색하면서 EVM 에서 실행 가능한 실행 포맷인 EFF 를 생성한다.

3.2 문법 설계

SIL 프로그램의 구조를 그대로 반영하고 있는 SIL 문법은 SIL 프로그램을 클래스의 집합으로 표현하고 있다. SIL 의 명령어의 수정에 따른 문법과 어휘정보 및 파싱 테이블의 변경을 피하기 위해 SIL 명령어의 개별적인 기술을 피하고, 가상 토큰 기법을 사용하여 스캐너에서 별도로 처리하였다.

SIL 의 명령어를 효율적으로 분석하기 위하여 context-free 문법으로 작성된 SIL 문법은 [표 1]과 같다.

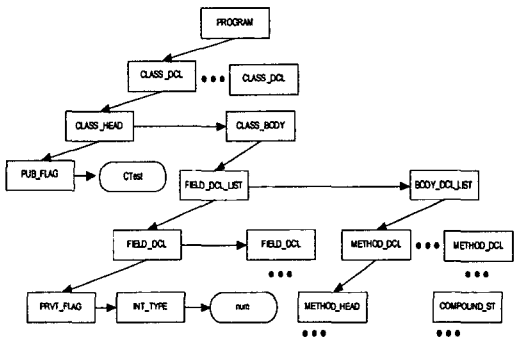
[표 1] SIL 문법

```
SYNTAX SIL

<SIL_program>     ::= { <class_dcl> }
<class_dcl>       ::= '.class' <access_flags> <class_name>[':' <super_class_name>]
                      '.bgn'
                      <class_body>
                      '.end'

<class_body>      ::= { <field_dcl> } { ( <method_dcl> | <class_dcl> ) }
<field_dcl>       ::= '.field' <access_flags> <type_specifier> <filed_name>
<method_dcl>      ::= '.method' <access_flags> <type_specifier> <method_name>
                      '(' [<formal_param>] ')'
                          '.bgn'
                          <method_body>
                          '.end'
<formal_param>    ::= <type_specifier> <formal_param_name>
                      {',' <type_specifier> <formal_param_name>}

<method_body>     ::= { <dcl_statement> } { <instr_statement> }
<dcl_statement>   ::= <stack_dcl_st> | <local_var_dcl_st> | <exception_dcl_st>
                      | <exception_proc_st>
<stack_dcl_st>    ::= '.maxstack' <number>
<local_var_dcl_st> ::= '.locals' '(' [<local_var_list>] ')'
<local_var_list>  ::= <local_var> { <local_var> }
<local_var>       ::= <type_specifier> <local_var_name> { ',' <local_var_name> }
<exception_dcl_st> ::= '.throws' <exception_type_name>
<exception_proc_st> ::= '.catch' <exception_type_name> 'from' <start_lable_name>
                        'to' <end_label_name> 'using' <except_label_name>

<instr_statement> ::= [ <label_name> ':' ]
                      ( <stack_op> | <arithmetic_op> | <flow_control_op>
                      | <object_op> | <typ_conversion_op> | <except_op> )

<access_flags>    ::= 'public' | 'private' | 'static' | 'terminal'
                      | 'guarded' | 'interface' | 'concept'
<type_specifier>  ::= 'byte' | 'integer' | 'long' | 'float' | 'double' | 'short'
                      | 'char' | 'reference' | 'boolean'

<*_name>          ::= '%ident'
<*_number>        ::= '%number'
<*_op>            ::= <opcode_name> [ <param_name> ]
```

3.3 스캐너, 파서 모듈

스캐너와 파서는 실행 파일 포맷 생성기에서 각각 어휘 분석 과정과 구문 분석 과정을 담당한다. 스캐너에서는 PGS 에서 생성된 정보를 기반으로 *.sil 을 입력으로 읽어 들여 토큰번호와 값으로 구성된 토큰으로 분류하게 된다. 또한 sil.gr 에서 명시하지 않은 SIL 의 명령어에 대한 분석이 이루어진다.

파서는 LR 방법으로 수행되며 PGS 에서 얻어진 파싱 테이블과 스캐너를 이용하여 생성된 토큰 정보를 가지고 shift, reduce, accept, error 의 4 가지 구동루틴을 통하여 입력된 *.sil 을 구문 분석하게 된다. 구문 분석 과정에서 SDT 방식으로 SIL 프로그램의 구조를 AST 로 변환하며, 심볼 테이블을 생성하여 EFF 생성에 필요한 정보를 가지게 된다. [그림 4]는 SIL 프로그램의 기본적인 구조를 나타낸 AST 형태이다.



[그림 4] AST 의 기본 구조도

Class 단위로 구성된 SIL 프로그램의 구조를 표현하기 위하여 PROGRAM 노드의 자식 노드가 CLASS_DCL 로 표현되어 있는 것을 확인 할 수 있다.

3.4 포맷 생성기

포맷 생성기는 스캐너와 파서를 통해 생성된 AST 와 심볼 테이블의 정보를 이용하여 EFF 를 생성하게 된다.

EFF 의 생성 순서는 header, metadata table, SIL 정보 순으로 이루어진다.

먼저, SIL 프로그램의 기본정보와 실행 파일 포맷 생성기의 환경설정을 이용하여 header 를 작성한다. 그리고 AST 를 운행하면서 수집한 정보와 SIL 지시어를 를 이용하여 EFF 의 metadata table 을 작성한다. AST 의 설계가 SIL code 와 이를 위한 정보를 분리하기 용이하게 구조적으로 이루어졌으므로 metadata 를 생성하기 위하여 필요한 AST 의 노드 방문 횟수가 줄어들게 된다. 마지막으로, SIL 정보 부분은 AST 의 명령어 부분을 순행하여 얻은 정보와 미리 정의된 명령어 테이블을 매칭하여 해당 명령어를 코드화하여 기록하게 된다.

4. 실행 파일 포맷 생성기의 구현

4.1 구현 환경

다음은 실행 파일 포맷 생성기를 개발하기 위한 기본환경 구성으로, 실행 파일 포맷 생성기의 입력인 SIL 을 얻기 위해 사용한 SIL 변역기(SIL Translator)의 사용 환경과 실행 파일 포맷 생성기의 구현 및 사용 환경으로 크게 두 가지로 나누어 기술하였다.

[표 2] 구현 환경

SIL Translator - SIL 1.12 - tools : JDK 1.4.2, Visual studio .NET (ver7.1) - O/S : Windows XP professional service pack 1 EFF Generator - EFF 1.0 - tools : Visual studio .NET (ver7.1) - O/S : Windows XP professional service pack 1

4.2 실험 결과

다음은 SIL 번역기를 이용하여 변환된 대상 SIL 프로그램을 포맷 생성기의 입력으로 사용하고, 그 결과로 얻어진 evm 파일이다. 거쳐 [표 3]에서의 실험 대상은 소수를 구하는 프로그램이다.

[표 3] 실험 결과

```
.class      public CPrime
.bgn
            .field      public static integer max
            .method     public static integer _initExtVarFunc( )
            .bgn
                        ldc.i       100
                        strsfld     CPrime::max
                        ret
            .end
            .method     public static integer Main( )
            .bgn
                        calls       void CPrime::_initExtVarFunc( )
                        .locals ( integer i, j, k, rem, prime)
                        ldc.i       2
...
```

```
...
00000050h: 04 12 05 12 05 12 03 0B 30 12 03 54 64 FF 4D 05
00000060h: 0E 01 00 10 00 03 01 01 00 01 00 00 00 00 00 00
00000070h: 00 10 02 02 00 02 00 00 00 00 00 02 00 03 00 00
00000080h: 00 00 00 30 02 4D 04 04 00 74 65 73 74 05 00 43
00000090h: 54 65 73 74 03 00 6E 75 6D 02 00 63 68 03 02 42
...
```

[표 4]는 실행 파일 포맷의 정보 분석을 위하여 만들어진 dump 프로그램의 결과이다. 실행 파일 포맷에서 메타데이터의 정보와 sil 프로그램에 대한 정보가 표현되어있는 것을 확인 할 수 있다.

[표 4] dump 결과

```
=== EFF information ===
Moudle : Prime
Language : java
Entry point : Main
...
=== Metadata information ===
- Metadata String Information
00 00 : Cpime                00 01 : max
00 02 : _initExtVarFunc      00 03 : Main
...
=== SIL information ===
// CPrime: _initExtVarFunc()
00 00 : code number [0X06] opcode [ldc.i]  operand [100]
...
```

5. 결론 및 향후 연구

EVM 솔루션은 모바일 디바이스와 디지털 TV에 적용 가능한 가상기계 기술로서 스택 머신을 기반으로 한 SIL은 EVM의 중간언어이다.

본 논문에서는 컴파일러 기술을 이용하여 SIL문법을 설계하고 포맷 생성기의 각 부분을 모듈화 하여 실행 파일 포맷 생성기를 구현하였으며, 이를 이용하여 SIL을 EVM에서 실행 가능한 파일 포맷인 EFF로 변환하였다. EFF는 EVM에서 실행하기 위한 실행파일 포맷이지만 가상기계의 특성상 EVM에서 효율적으로 실행 되기 위해서는 대상 가상기계에 최적화된 형태로 변환이 필요하다.

향후 과제로는 실행 파일 포맷 생성기의 보완 및 EFF의 최적화에 대한 연구가 필요하겠다.

참고문헌

[1] David Platt, Introducing Microsoft .NET 2nd Edition, Microsoft Press, 2002.

[2] ECMA, Common Language Infrastructure (CLI), Microsoft, 2002.

[3] Kevin Burton, .NET Common Language Runtime, SAMS, 2002.

[4] MSIL Instruction Set Specification, Microsoft Corporation, Nov. 20. 2000.

[5] Serge Lindin, INSIDE MICROSOFT .NET IL ASSEMBLER, Microsoft Corporation, Feb. 6. 2002.

[6] 남동근, 윤성림, 오세만, "가상기계의 어셈블리 언어", 정보처리학회 춘계학술발표논문집(중), 제 10 권 제 1 호, pp.783-786, 2003.

[7] 오세만, 컴파일러 입문 개정판, 정익사, 2004.

[8] 오세만, 이양선, 고광만, 임베디드 시스템을 위한 가상기계의 설계 및 구현, 한국과학재단 특정 기초 연구, 2 차 중간보고서, 2004.6.

[9] 정한종, 윤성림, 오세만, "가상기계를 위한 실행 파일 포맷", 정보처리학회 추계학술발표논문집(중), 제 10 권 제 2 호, pp.647-650, 2003.