가상기계 코드 실행을 위한 역컴파일러

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

Decompiler for Executing Virtual Machine Code

Duk-ki Ahn*, Se-man Oh
Dept. of Computer Engineering, Dongguk University

요 약

최근 가상기계 시스템은 임베디드 유비쿼터스 컴퓨팅의 필수적인 기술로서, 그 중요성이 더욱 강조되고 있으며, 컴파일러, 어셈블러 그리고 가상기계의 구현으로 구축된다. 이러한 시스템의 구축 과정에서 각 컴포넌트의 신뢰성을 위하여 정확한 검증 방법이 요구되며, 검증의 효율성을 위해서 순차적으로 진행되어야 한다.

본 논문에서는 가상기계 시스템의 컴파일러를 검증하기 위해서, 컴파일 된 가상기계 코드를 역컴파일하여 실행하는 기법을 제안하고, 그러한 기법에 따라 EVM(Embedded Virtual Machine) SIL(Standard Intermediate Language) 역컴파일러를 구현하였다. 구현된 역컴파일러는 EVM이 개발되기 전에 효율적인 실행 시스템으로 이용됨은 물론 EVM ANSI C 컴파일러의 검증 도구로서 이용될 수 있으며, EVM 시스템을 체계적으로 개발할 수 있도록 할 것이다.

1. 서 론

최근 임베디드 유비쿼터스 컴퓨팅을 위한 연구가 강조되고 있으며, 이를 위하여 가상기계 시스템 구축 기술은 필수적이다.

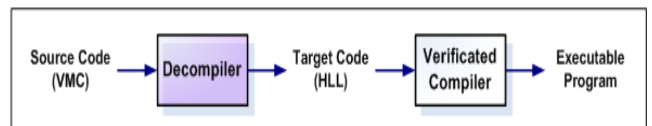
가상기계 시스템은 컴파일러, 어셈블러 그리고 가상기계로 구성되며, 컴파일러에 의해서 어셈블리 형식의 가상기계 코드가 생성되고 어셈블러에 의해 가상기계에서 실행 가능한 구조로 변경되어 실행된다. 이러한 시스템의 구축은 일반적으로 병렬적으로 수행되며, 구축 과정에서 각 컴포넌트 단위의 신뢰성을 위하여 정확한 검증 방법이 필요하다. 또한 검증의 효율성을 위하여 순차적으로 진행되어야 할 것이다.

컴파일러의 검증 여부는 생성된 가상기계 코드를 실행함으로써 확인할 수 있으며, 이러한 관점에서 **<표 1>**과 같이 3가지 방법으로 구분할 수 있다.

<표 1> 가상기계 코드 실행 방법

VMC 실행 방법	실행 환경
컴파일링 기법	컴파일러의 후단부
인터프리팅 기법	가상기계의 인터프리터
역컴파일링 기법	역컴파일러

역컴파일링 기법은 **(그림 1)**과 같은 과정으로 가상기계 코드를 변환하여 실행한다. 이 기법은 검증 데이터로서 가상기계 코드만이 요구되며, 검증된 컴파일러를 이용하여 실행함으로써 컴파일링 기법보다 신뢰적이고, 인터프리팅 기법과 비교하여 구현 시간과 노력이 적게 요구된다.



(그림 1) 역컴파일러에 의한 VMC 실행 과정

본 논문에서는 가상기계 시스템의 컴파일러를 검증하기 위해서 스택 기반으로 연산되는 가상기계 코드를 역컴파일하여 실행하는 기법을 제안하였고, EVM의 가상기계 코드인 SIL[9, 10] 코드를 기준으로 구현하였다. 구현한 역컴파일러 시스템은 해석적 기법에 기초한 C 코드 생성 부분과 MS Visual C 컴파일러를 이용한 실행 부분을 단계적 또는 일괄적으로 수행하도록 함으로써, 컴파일러 검증은 물론 가상기계 코드 실행에 보다 능률적으로 이용될 수 있다.

2. 관련 연구

2.1 가상기계 코드

가상기계 코드란 가상기계에서 실행될 수 있는 코드(Java bytecode, .NET PE)와 대응되는 명령어의 집합을 말하며, 연산방식에 따라 스택 기반 코드와 레지스터 기반 코드로 구분할 수 있다. 대부분의 가상기계 코드는 Java Oolong[6], .NET IL[7] 그리고 EVM SIL과 같이 스택 기반 코드로 설계되며, 필요에 따라서 PASM(Parrot ASseMbly language)[3]과 같이 레지스터 기반 코드로 설계되기도 한다.

가상기계 코드는 일반적으로 데이터부(data section)의 의사 코드와 코드부(code section)의 연산 코드로 구성된다.

2.2 역컴파일러

역컴파일러란 컴파일 된 기계어나 어셈블리 코드로부터 컴파일 전의 본래 소스 코드를 추출해 내는 프로그램을 의미하며, 일반적으로 본래 소스 코드와 동일한 소스 코드를 추출할 수 없으므로 의미적으로 동등한 고급언어가 생성되도록 구현된다.

역컴파일러는 컴파일 된 파일의 구조에 따라서 <표 2>와 같이 구분할 수 있으며, 바이너리 코드의 역컴파일 과정은 역어셈블 단계가 포함되므로 어셈블리 코드와 비교하여 더 복잡하다.

<표 2> 프로그램 수준에 따른 역컴파일러의 종류

프로그램 수준	역컴파일러	소스파일
바이너리 코드	dcc[2], ExeToC REC Uncc Decomp Decompiler JODE, Jasmine spices.decompiler, Dis# ...	80286 DOS binaries Elf, COFF, AOUT, … X86 binary files Vax BSD 4.2 (a.out format) Java bytecode .NET PE ...
어셈블리 코드	RelipmoC XACT Asm21ToC ...	i386 i386, ULTRIX, and CDC469 ADSP-21xx ...

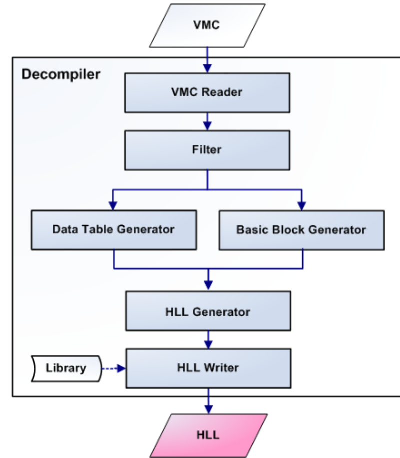
가상기계 코드는 어셈블리 코드 수준에 속하며 역컴파일 과정이 비교적 간단하다. 최근에는 언어에 독립적인 가상기계 코드가 쟁점이 되고 있으며, 이러한 가상기계 코드로부터 여러 종류의 고급언어를 추출할 수 있는 크로스 역컴파일러의 구현은 쉽지 않다. spices.decompiler는 대표적인 크로스 역컴파일러로서, .NET IL에서 5가지 언어(C#, VB.Net, Delphi.Net, J# and managed C++)를 추출할 수 있다.

2.3 해석적 코드 생성

해석적 코드 생성 기법은 컴파일러 후단부의 코드 생성 자동화(automatic code generation)를 위해서 이용되는 방법이며, 중간 언어의 형태가 추상기계(abstract machine) 명령어로 구성된 중간 표현에 대해서 주로 이용되는 기법이다. 현재 이 기법은 목적기계의 특징을 묘사한 부분과 코드 생성 알고리즘을 혼합 또는 분리하여 구현할 수 있으며[11], 코드 생성의 재목적성(retargetability)을 위한 경우 분리하여 구현된다.

3. 역컴파일러의 구조

역컴파일러는 (그림 2)와 같이 구성되며, 데이터 테이블 생성기, 기본 블록 생성기 그리고 고급언어 생성기가 역컴파일 과정에서 핵심 역할을 한다.



(그림 2) 역컴파일러의 구조

3.1 코드 리더와 필터

가상기계 코드를 읽어서 역컴파일 과정 중 불필요한 코드를 제거한다. 즉, 가상기계 코드의 주석문과 라이브러리 수준에서 추가된 코드가 제거된다.

3.2 데이터 테이블 및 기본 블록 생성기

필터링 된 가상기계 코드의 데이터부와 코드부를 참조하여, 데이터 테이블과 기본 블록을 생성한다.

데이터 테이블은 구조형, 함수, 변수 그리고 문자열 상수 영역과 같은 정보로 구성되며, 기본 블록은 고급언어의 기본문을 의미하는 가상기계 명령어를 기준으로 구성된다. <표 3>은 EVM SIL, Java Oolong 그리고 .NET IL에서 기본 블록을 구성하기 위한 가상기계 명령어를 나타낸 것이다.

<표 3> 기본 블록 구성 기준 VM 명령어

HLL 기본문		VMI (VM Instruction)		
		EVM SIL	Java Oolong	.NET IL
배 정 문		str.[Dtypes]	[Dtypes]store_[0-3]	st[sfld,fld,loc,arg].[0-3]
제어문 반복문	무조건 분기	ujp	goto	br, br.s
	조건 분기	tjp, fjp	ifeq,ifne,ifgt,iflt,...	beq,bne,bgt,blt
호 출 문		call, calli, calls	invokespecial,...	call, calli
반 환 문		ret, retv.[Dtypes]	Dtypes.return	ret

※ Dtypes : 자료형 기호, [ ] : 괄호내의 요소 중 택일.

3.3 고급언어 생성기

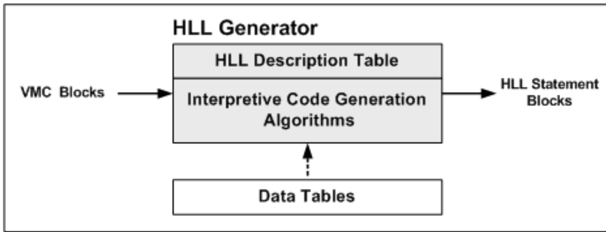
가상기계 연산 코드로 구성된 각 블록을 고급언어문(HLL statement)으로 변환한다. 이 과정에서 구조형 정의, 함수의 헤더 그리고 변수 선언문이 해당 데이터 테이블을 참조하여 구성되며, 고급언어의 제어문을 조건문과 분기문으로 대치함으로써 고급언어 생성기의 구현에 대한 노력을 줄일 수 있다.

3.4 고급언어 기록기

라이브러리 바인딩 코드를 추가하며, 기본 블록을 참조하여 고급언어 프로그램을 생성한다.

4. 고급언어 생성 기법

각 블록내의 가상기계 연산 코드는 **(그림 3)**과 같이 고급언어 생성기에 의해서 고급언어문으로 변환된다.



(그림 3) HLL 생성기

고급언어 생성기는 자료형 및 연산자를 고급언어 묘사 테이블과 데이터 테이블을 참조하여, 각 블록의 형에 따라 해석적 기법으로 고급언어문을 생성한다.

배정형(assignment type), 조건 분기형(conditional jump type), 호출형(procedure call type) 그리고 반환형(procedure return type) 블록은 스택 명령어와 연산 명령어가 반복된 후 코드의 마지막에 각 블록의 고유한 명령어가 나타나므로 일부 동일한 처리 루틴이 적용된다.

4.1 배정형

데이터 테이블을 참조하여 스택 및 배정 명령어에 대응되는 변수 및 상수 정보를 얻고, 고급언어 묘사 테이블을 참조하여 가상기계 연산 명령어에 대응되는 연산자를 얻어, 이를 고급언어의 배정문으로 변환한다.

4.2 무조건 분기형

특정 위치로 제어가 이동되는 명령어가 포함된 블록으로써, 일반적으로 무조건 분기 명령어와 분기 위치만 있으며, 고급언어의 무조건 분기를 의미하는 지정어와 가상기계 명령어의 분기 레이블을 조합하여 고급언어의 무조건 분기문으로 변환한다.

4.3 조건 분기형

조건에 따라 특정 위치로 제어가 이동되는 코드가 포함된 블록으로써, 배정형과 동일한 방법으로 변수 및 상수 정보와 고급언어의 연산자를 얻어, 고급언어의 조건문과 분기문으로 변환한다.

4.4 호출형

내장형 또는 정의형 프로시저의 호출 정보가 포함된 블록으로써, 실매개변수(actual parameter)와 프로시저 이름이 포함된 블록이다. 배정형과 동일한 방법으로 실매개변수를 얻어, 이를 프로시저 호출 명령어와 조합하여 고급언어의 프로시저 호출문으로 변환한다.

4.5 반환형

프로시저의 반환값(return value) 정보가 포함된 블록으로써, 배정형과 동일한 방법으로 반환값 정보를 얻어 고급언어의 반환문으로 변환한다.

5. 역컴파일러의 구현

5.1 구현 환경 및 실험 자료

본 논문에서 제안한 역컴파일러는 MS Visual C#을 이용하여 구현하였고, 역컴파일러의 입력 가상기계 코드는 EVM SIL 코드 그리고 목표 고급언어는 C 언어이다.

실험 자료는 모든 블록의 형이 생성될 수 있는 간단명료한 원본 C 코드를 준비하였으며, EVM ANSI C 컴파일러에 의해서 컴파일 된 SIL 코드와 함께 **(그림 5)**에 나타내었다.

원본 C 코드

```
#define _WIN32
#include "C:\Program Files\Microsoft Visual Studio\VC98\Include\stdio.h"
#define MAXDISC 64
int strlen(char *s)
{
        char *t=s;
        while (*s != '\0')
                s++;
        return s-t;
}
int main(void)
{
        char *s="Decompiler for Executing VMC";
        int size=strlen(s);
        printf("%d\n", size);
        return 0;
}
```

컴파일 된 SIL 코드

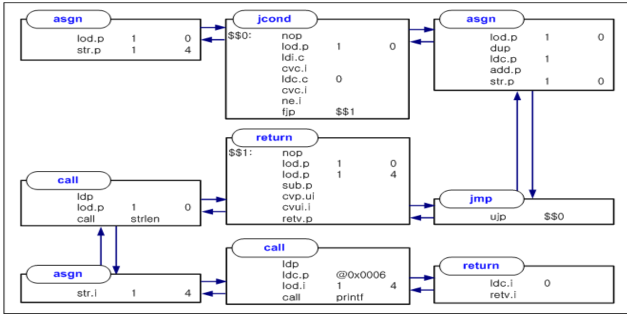
```
%%Data Section
%Structure Table
.struct         _iobuf          8   ...// 생략
.struct         fpos_t          2   ...// 생략
%Literal Table
.literal        0               3   ...// 생략
%External Symbol Table
.symx           _iob            t   ...// 생략
%Internal Function Table
.func           strlen          i   ...// 생략
.func           main            i   ...// 생략
%External Function Table
%%Code Section
%File : C:\SIL_Decompiler\Test_File\strlen.c
strlen: proc    8               1   ...// 생략
.sym            s               p   ...// 생략
.sym            t               p   ...// 생략
%Line 8 : char *t=s;
                lod.p           1   0
                str.p           1   4
%Line 9 : while (*s != '\0')
$$0:            nop
                lod.p           1   0
                ldi.c
                cvc.i
                ldc.c           0
                cvc.i
                ne.i
                fjp             $$1
```

```
%Line 10 : s++;
                lod.p           1   0
                dup
                ldc.p           1
                add.p
                str.p           1   0
                ujp             $$0
$$1:            nop
                lod.p           1   0
                lod.p           1   4
                sub.p
                cvp.ui
                cvui.i
                retv.p
%File : C:\생략\strlen.c
main: proc      8               1   1
.sym            s               p   ...// 생략
.sym            size            i   ...// 생략
%Line 16 : int size=strlen(s);
                ldp
                lod.p           1   0
                call            strlen
                str.i           1   4
%Line 17 : printf("%d\n", size);
                ldp
                ldc.p           @0x001d
                lod.i           1   4
                call            printf
                ldc.i           0
                retv.i
```

(그림 5) 원본 C 코드와 컴파일 된 SIL 코드

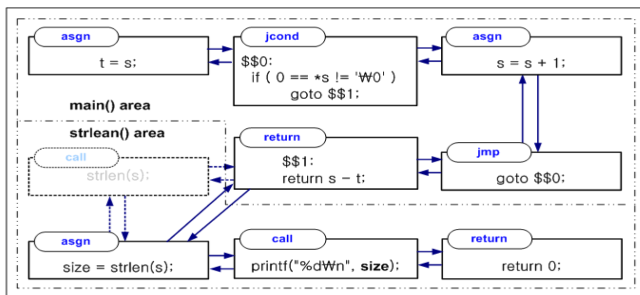
5.2 역컴파일 과정

역컴파일러는 **(그림 5)**의 컴파일 된 SIL 코드를 입력받아 필터링하고 기본 블록 생성기에 의해서 **(그림 6)**과 같이 이중 연결 리스트(double linked list)로 구성한다. 모든 블록은 기본적으로 블록의 형, SIL 코드 그리고 블록이 속한 함수의 이름으로 구성되며, 선택적으로 블록의 레이블이 구성된다. 각 블록이 속한 함수의 이름은 변수 및 함수 테이블을 참조할 수 있는 인터페이스 역할을 한다.



(그림 6) 기본 블록으로 분리된 SIL 코드

C 언어 생성기에 의해서 각 블록은 C 언어문으로 변환되며, asgn 블록은 C 언어의 문법에 기초하여 선택적으로 다른형과 병합되어 변환된다. (그림 7)은 C 언어 생성기에 의해서 변환된 블록의 논리적 구조를 나타낸다. 이 과정에서 각 블록의 함수 이름을 기준으로 블록이 구분되며, 구분 위치에 함수의 헤더 정보와 변수 선언 코드가 추가된다.

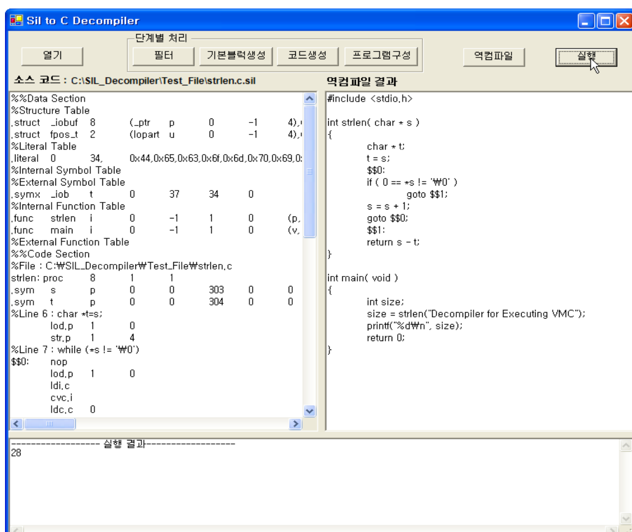


(그림 7) C 언어문으로 변환된 기본 블록

마지막으로 함수 테이블과 call 블록의 함수 이름을 참조하여 라이브러리 바인딩 코드가 추가되고, 불필요한 코드가 제거되어 C 언어 프로그램이 생성된다.

5.3 역컴파일러 실행

SILtoC 역컴파일러는 SIL 코드를 바로 실행하거나 모듈 단위의 결과를 수정해가며, 실행할 수 있는 GUI를 제공함으로써, SIL 코드를 보다 유연하게 시험해 볼 수 있다. (그림 7)은 SILtoC 역컴파일러 환경에서 EVM SIL 코드를 실행시킨 결과이다.



(그림 7) SIL 코드 실행 결과

6. 결론 및 향후 연구

본 연구에서는 가상기계 시스템의 컴파일러를 검증하기 위해서, 가상기계 코드를 실행할 수 있는 역컴파일링 기법을 제안하였다. 또한, 제안한 기법에 따라서 구현한 역컴파일링 실행 시스템은 SIL 코드와 의미적으로 동등한 C 언어를 생성과 동시에 실행시킴으로써, 컴파일러의 검증 여부를 효과적으로 확인할 수 있도록 하였다. 이러한 결과는 EVM 시스템의 어셈블러 그리고 가상기계를 보다 완전하게 구현하도록 기여할 것이다.

최근 가상기계 시스템은 다양한 언어에 독립적으로 수행되도록 구축되고 있으며, 이를 위해서 가상기계 코드는 객체지향 언어를 기반으로 하여, 다양한 고급 언어를 수용하도록 설계되고 있다. 따라서 본 연구를 바탕으로 언어에 독립적인 가상기계 코드에서 크로스 고급언어를 추출할 수 있는 개선된 역컴파일링 기법 제안과 그러한 역컴파일러를 구현할 계획이다.

참고문헌

[1] Cristina Cifuentes, et al., "Assembly to High-Level Language Translation," ICSM'98, 1998.

[2] Cristina Cifuentes, "Reverse Compilation Techniques," PhD dissertation, Queensland University of Technology, School of Computing Science, July 1994.

[3] Fabian Fagerholm, "Perl 6 and the Parrot Virtual Machine," April 1, 2005.

[4] James Gosling, "Java Intermediate Bytecodes," ACM SIGPLAN Workshop on Intermediate Representations, 1995.

[5] John Aycock, "Converting Python Virtual Machine Code to C," In Proc. of 7th Intl. Python Conf., 1998.

[6] John Meyer & Troy Downing, Java Virtual Machine, O'REILLY, Mar. 1997.

[7] MSIL Instruction Set Specification Version 1.9 Final, Microsoft Corporation, 10 October 2000.

[8] Jin.B.j, SAF Specification Version 0.9, SINJISOFT, 2005. 2.

[9] 남동근, SIL Specification Version 1.0, SINJISOFT, 2005. 8.

[10] 남동근ㆍ윤성림ㆍ오세만, "가상기계를 위한 어셈블리 언어", 정보처리학회 2003 춘계 학술 발표 논문집, Vol.11, No.1, pp.519-522, Mar. 2004.

[11] 오세만, 컴파일러 입문 개정판, 정익사, 2004.2.