C++ 컴파일러에서 중간코드의 검증과 분석을 위한 역컴파일러의 설계 및 구현

배성균°, 김영근, 이양선
서경대학교 컴퓨터공학과
e-mail:{skbae, ykkim, yslee}@pl.skuniv.ac.kr

Design and Implementation of a Decompiler for Verification and Analysis of Intermediate Code in C++ Compiler

Sung-Kyun Bae°, Young-Keun Kim, Yang-Sun Lee
Dept. of Computer Engineering, SeoKyeong University

요 약

C++ 언어는 객체지향 프로그래밍 언어로, 기존의 C++ 프로그램은 각각의 플랫폼에 따른 컴파일러를 통해 목적기계의 코드(object code)로 변환되므로 실행되는 플랫폼에 의존적인 단점이 있다. 이러한 단점을 보완하는 방법으로 스택기반의 가상기계와 가상기계의 입력형태인 중간코드를 이용하는 기법이 있다. EVM(Embedded Virtual Machine)은 ANSI C, ISO/IEC C++ 언어와 SUN사의 Java 언어 등을 모두 수용할 수 있는 임베디드 시스템 기반의 가상기계이며, EVM에서 실행되는 중간코드인 SIL(Standard Intermediate Language)은 객체지향 언어와 순차적인 언어를 모두 수용하기 위한 명령 코드의 집합으로 설계되어 있다.

본 논문에서는 C++ 컴파일러를 통해 생성된 SIL 코드가 올바른지 검증하고 원시코드의 분석을 용이하게 하기 위해서 SIL 코드를 어셈블리 코드와 유사한 형태의 재 표현된 C++ 프로그램으로 역컴파일하는 시스템을 설계하고 구현하였다.

1. 서론

C++ 언어는 C 언어의 기능을 확장하여 만든 객체지향 프로그래밍 언어로, C의 특징을 모두 수용하고 있어 시스템 프로그램에 적당할 뿐만 아니라 클래스, 연산자 중복, 상속 등과 같은 객체지향의 특성을 지원하여 여러 응용분야에서 실용적인 언어로 이용되고 있다[1,2]. 이를 기반으로 작성된 프로그램은 각각의 플랫폼에 따른 컴파일러를 통해서 목적기계의 코드로 변환되므로, 실행되는 플랫폼에 의존적인 단점이 있다. 이러한 단점을 보완하는 방법으로 스택기반의 가상기계와 가상기계의 입력형태인 중간코드를 이용하는 기법이 있다.

가상기계에는 SUN사의 JVM과 마이크로소프트사의 .NET Platform이 있으며, 각각 중간 언어로 바이트코드와 MSIL 코드를 입력으로 받아 플랫폼에 독립적으로 실행된다. EVM은 휴대용 장치, 셋톱박스, 디지털 TV와 같은 임베디드 시스템을 위한 스택기반의 가상기계로, SIL 코드를 중간코드로 사용한다. SIL 코드는 객체지향 언어와 순차적인 언어를 모두 수용하기 위해 설계된 연산 코드의 집합이다.

기존의 C++ 언어로 작성된 프로그램을 C++ 컴파일러를 통해 SIL 코드를 생성하여 EVM 환경에서 실행함으로써 플랫폼에 관계없이 프로그램을 실행할 수 있는 기반을 제공하는 것이다.

본 연구팀은 EVM을 위한 C++ 컴파일러를 개발하였으며, 컴파일러를 통해 중간코드로 생성된 SIL 코드의 출력 결과가 올바른지 확인할 필요성을 가지게 되었다. 이를 위한 검증 방법으로 중간코드를 이용한 역컴파일 기법과 가상기계 인터프리터 기법이 있는데, 그 중 역컴파일 기법을 사용하였다.

SIL코드를 재 표현된 C++ 언어로 바꾸어주는 역컴파일러는 SIL 코드를 검증하기 위해서 3-주소 코드의 쿼드러플(quadruple) 형태의 C++ 프로그램으로 역컴파일 하여, 기존의 Visual C++ 컴파일러를 통해 실행하는 방법과 실행을 위한 검증만이 아닌 원시코드의 분석을 위한 방법을 병행하기 위해 필요한 시스템이다.

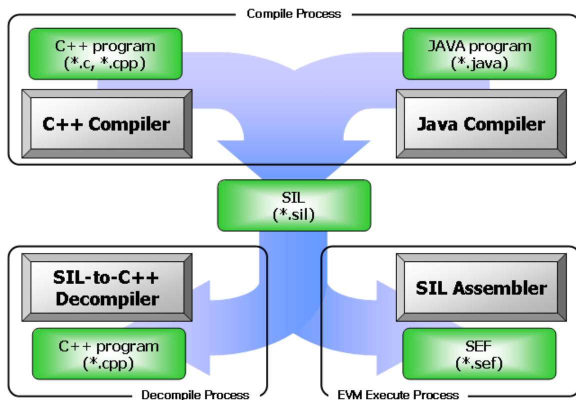
본 논문에서는 이를 위해 EVM의 중간코드인 SIL 코드를 재 표현된 C++ 언어로 바꾸어 주는 역컴파일러를 설계 및 구현하여 객체지향 언어 기반의 중간코드인 SIL 코드를 검증하는 시스템을 구축하였다.

2. 관련연구

2.1 EVM

EVM은 휴대용 장치, 셋톱 박스, 디지털 TV 등에 탑재되어 동적 응용 프로그램을 다운로드하여 실행할 수 있는 가상기계 솔루션이다.

EVM은 크게 컴파일러, 어셈블러, 가상기계의 세 부분으로 구성된다. EVM은 계층적인 구조로 설계되어 리타겟팅 과정의 부담을 최소화 한다[5]. 다음 [그림1]은 EVM의 시스템 구성도 이다.



[그림1] EVM 시스템 구성도

2.2 SIL

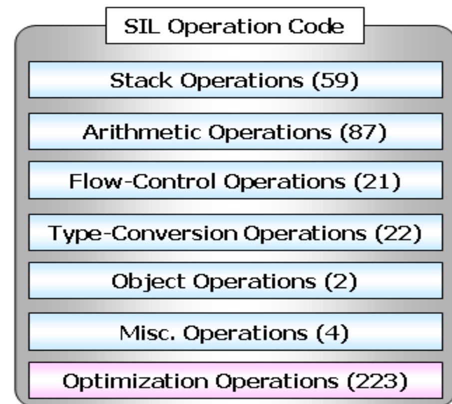
EVM의 가상기계 코드인 SIL은 일반적인 임베디드 시스템을 위한 가상기계 코드의 표준화 모델로 설계되었다.

SIL은 스택 기반의 명령어 집합으로 언어 독립성과 하드웨어 및 플랫폼 독립성을 갖고 있다. SIL은 다양한 프로그래밍 언어를 수용하기 위해서 바이트 코드, .NET IL 등 기존의 가상기계 코드들의 분석을 토대로 정의 되었으며, 객체지향 언어와 순차적 언어를 모두 수용하기 위한 연산 코드의 집합으로 구성되어 있다.

SIL은 클래스 선언 등 특정 작업의 수행을 나타내는 의사 코드와 실제 명령어에 대응되는 연산 코드로 이루어져 있다[5].

연산 코드는 특정 하드웨어나 소스 언어에 종속되지 않는 추상적인 형태를 지니며, 어셈블리 언어 수준의 디버깅을 용이하게 하기 위해 일관성 있는 이름 규칙을 적용하여 가독성 높은 니모닉으로 정의되어있다. 또한 최적화를 위한 short form 연산 코드를 갖고 있다.

SIL은 다음과 같은 7개의 카테고리로 분류할 수 있으며 각각의 카테고리는 서브 카테고리를 갖는다. [그림2]는 SIL의 연산 코드 카테고리 이다.

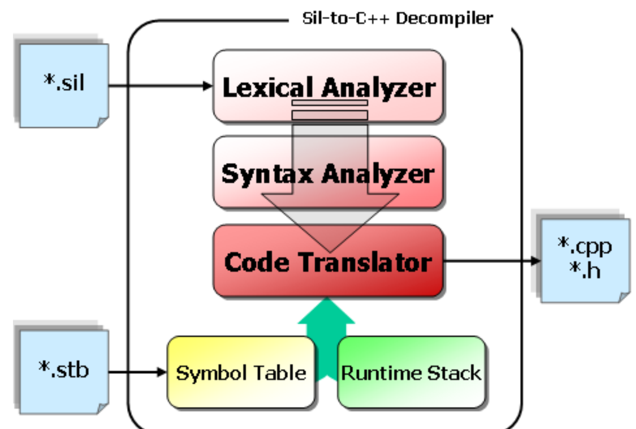


[그림2] SIL의 연산 코드 카테고리

3. SIL-to-C++ 역컴파일러

3.1 시스템 구성도

SIL-to-C++ 역컴파일러는 SIL 코드 및 데이터를 포함한 SAF(Standard Assembly Format)와 명칭에 대한 정보를 저장한 심벌 테이블 자료구조 STB를 입력으로 받는다.



[그림3] SIL-to-C++ 역컴파일러 구성도

SAF는 어휘 분석 및 구문 분석 과정을 통해 SIL

연산 코드의 트리를 작성하고, 이는 스택 기반의 가상기계 기법을 이용하여 재 표현된 C++ 프로그램을 생성한다. STB는 명칭 선언과 사용자 정의 타입 선언으로 이루어진 C++ 프로그램의 선언부를 생성한다. [그림3]은 SIL-to-C++ 역컴파일러 시스템 구성도이다.

3.2 자료구조

SIL-to-C++ 역컴파일러는 C++ 프로그램에서 선언된 변수들과 타입을 참조하기 위한 심벌 테이블의 자료구조와 연산 코드의 실행을 위한 런타임 스택(Runtime stack), C++ 에서 지원되는 API 함수 사용에 따른 헤더 파일 선언을 처리하기 위한 API 테이블이 존재한다. 심벌 테이블은 변수의 정보를 저장하는 symbolTable과 사용자 정의 타입을 저장하기 위한 typeTable을 중심으로 구성되어 각각의 테이블은 다시 세부 테이블로 이루어진다. 런타임 스택은 operator번호와 operator에 대한 parameter, operand로 구성 된다. API 테이블은 해당 API 함수의 사용 여부를 저장한다.

3.3 역컴파일러

역컴파일 과정은 크게 Detranslate, Assemble, Execute의 3부분으로 이루어진다.

Detranslate 과정은 심벌 테이블에 저장된 정보를 이용하여 C++ 프로그램의 선언부를 복원한다.

[표1] 재 표현된 C++ 코드의 출력 형식

Operations	Format
Store to XXX operations	format : operatorName = (rhs); objectName.operatorName = (rhs);
Increase/Decrease operations	format : tempVarialble = stack[top].operandValue++;

Jump operations	format : tjp - if(operation) goto operand;
	fjp - if(!(operation)) goto operand;
	ujp - goto operand;
	ret - return;
	retv.t - return operand;

Function call operations	format : tempVariable = operatorName(operation); tempVariable = objectName.operatorName(operation);
Activation record operations	format : return_type functionName(arg_type arg_name); return_type typeName::funcName(arg_type arg_name);
Optimization operations	format : arithmetic (a+b, a-b, a*b, a/b, a[i]+b[j], ...) array element (a[i], a[a[i]], a[i] = j, a[a[i]] = j, ...)

Assemble 과정은 코드를 변환 하는 과정이다. 변수를 읽는 코드(lod)는 심벌 테이블을 참조하여 변수의 이름으로 변경하고, 변경된 변수의 이름은 임시 변수와 런타임 스택에 저장한다. 산술 연산 코드와 비교 연산 코드는 연산의 결과를 임시 변수에 저장하고 임시 변수를 런타임 스택에 저장한다.

Execute 과정은 해당 연산 코드에 따라 수행하게 되는 해당 C++ 코드를 생성한다. [표1]은 실행에 관련된 출력 형태를 나타낸 것이다.

4. 실험 및 결과

다음은 클래스를 기반으로 한 행렬 연산 프로그램으로 C++ 컴파일러로 컴파일해서 SIL(*.sil)과 STB(*.stb)를 받아 역컴파일한 결과(*.cpp)와 이를 Visual C++ Compiler로 컴파일한 것이다.

C++ 프로그램 (*.cpp)

```
#define _WIN32
#define _MT 1
#define _WCHAR_T_DEFINED 1
#include "C:\Program Files\Microsoft Visual Studio\VC98\Include\stdio.h"
class matrix2x2
{
public:
      int a1, a2;
      int b1, b2;

      void mult(matrix2x2 mat2x2);
      void add(matrix2x2 mat2x2);
      void substract(matrix2x2 mat2x2);
      void print();
      matrix2x2(int a, int b, int c, int d);
      ~matrix2x2() { }
};
//    ... 클래스 함수 선언부 생략 ...
void main()
{
      matrix2x2 m1(3,7,5,2);
      matrix2x2 m2(1,0,0,1);
      m1.add(m2);
      printf("= m1 =\n");    m1.print(); printf("\n");
//    ... 중간 생략 ...
}
```

SIL 코드 (*.sil)

```
%%HeaderSectionStart
      %DefinedLiteralCount10
      %InitializedVariableCount  0
      %UninitializedVariableCount      0
      %ExternalVariableCount   1
      %ExternalFunctionCount   0
      %EntryFunctionName       &main
%%HeaderSectionEnd
%%CodeSectionStart
//    ... 중간 생략 ...
      %FunctionStart
            .func_name        &main
            .func_type 2
            .param_count     2
            .opcode_start
                  proc 32     1     1
                  ldp
                  lda    1    0
```

```
                ldc.i  3
                ldc.i  7
                ldc.i  5
                ldc.i  2
                call   &matrix2x2::matrix2x2$5
//    ... 중간 생략 ...
                ldp
                lda    1      0
                lod.t  1      16     16
                call   &matrix2x2::add$2
                ldp
                ldc.p  @0x0032
                call   &printf
                ldp
                lda    1      0
                call   &matrix2x2::print$4
                ldp
                ldc.p  @0x003b
                call   &printf
//    ... 중간 생략 ...
                ret
            .opcode_end
        %FunctionEnd
%%CodeSectionEnd
%%DataSectionStart
        %LiteralTableStart
            .literal_start      @0    0      29
0x61,0x31,0x3d,0x25,0x64,0x2c,0x20,0x61,0x32,0x3d,0x25,0x64,0x5
c,0x6e,0x62,0x31,0x3d,0x25,0x64,0x2c,0x20,0x62,0x32,0x3d,0x25,0x
64,0x5c,0x6e,0x00
            .literal_end
//    ... 중간 생략 ...
        %LiteralTableEnd
        %InternalSymbolTableStart
        %InternalSymbolTableEnd
        %ExternalSymbolTableStart
            .evar_decl $_iob
        %ExternalSymbolTableEnd
%%DataSectionEnd
```

역컴파일 결과 (*.cpp)

```
#include "matrix.cpp.sil.h"
/* global sym dcl */
//    ... 중간 생략 ...
// C++ SourceFile : matrix.c
void main()
{
/* local sym dcl */
        matrix2x2 silSym_9;
        matrix2x2 silSym_10;

        tmpCharPointer_112 = (char *)&silSym_9;
        tmpCharPointer_113 = (char *)(tmpCharPointer_112 + 0);
        tmpIntPointer_28 = (int *)tmpCharPointer_113;
        *tmpIntPointer_28 = 3;
        tmpCharPointer_114 = (char *)&silSym_9;
        tmpCharPointer_115 = (char *)(tmpCharPointer_114 + 4);
        tmpIntPointer_29 = (int *)tmpCharPointer_115;
        *tmpIntPointer_29 = 7;
        tmpCharPointer_116 = (char *)&silSym_9;
        tmpCharPointer_117 = (char *)(tmpCharPointer_116 + 8);
        tmpIntPointer_30 = (int *)tmpCharPointer_117;
        *tmpIntPointer_30 = 5;
        tmpCharPointer_118 = (char *)&silSym_9;
        tmpCharPointer_119 = (char *)(tmpCharPointer_118 + 12);
        tmpIntPointer_31 = (int *)tmpCharPointer_119;
        *tmpIntPointer_31 = 2;
//    ... 중간 생략 ...
        silSym_9.add(silSym_10);
        tmpInt_80 = printf("= m1 =\n");
        silSym_9.print();
        tmpInt_81 = printf("\n");
}
```

실행 결과

```
C:\WINDOWS\system32\cmd.exe

C:\siltocpp\testsource>matrix.exe
= m1 =
a1=4, a2=7
b1=5, b2=3

= m1 =
a1=4, a2=7
b1=5, b2=3

= m1 =
a1=3, a2=7
b1=5, b2=2

C:\siltocpp\testsource>
```

```
C:\WINDOWS\system32\cmd.exe

C:\siltocpp\testsource>matrix.cpp.sil.exe
= m1 =
a1=4, a2=7
b1=5, b2=3

= m1 =
a1=4, a2=7
b1=5, b2=3

= m1 =
a1=3, a2=7
b1=5, b2=2

C:\siltocpp\testsource>_
```

5. 결론

본 논문은 컴파일러를 통해 생성된 가상기계 기반의 SIL 코드가 올바른지 검증하기 위해서 SIL 코드를 재 표현된 C++ 언어로 생성하여 Visual C++ 컴파일러를 통해 실행하였다. 따라서 구현한 C++ 컴파일러가 C++ 프로그램으로부터 SIL 코드를 올바르게 생성하였음을 확인할 수 있었다. 또한, SIL 코드가 어셈블리 형태를 지니므로 코드에 대한 분석에 어려움이 있었으나, 본 논문에서 제시한 역컴파일러를 통해 생성된 C++ 코드는 소스 레벨의 분석이 가능하다는 장점이 있다.

참고문헌

[1] Bjarne Stroustrup, "The C++ Programming Language", Addison-Wesley, 2000

[2] INTERNATIONAL STANDARD ISO/IEC 14882: 1998(E) "Programming Language - C++", ISO/IEC, 1998

[3] 최성규 · 박진기 · 이양선, ".NET 언어를 위한 중간 언어 번역기", 멀티미디어학회 2003 추계 학술 발표 대회 논문집, Vol.6, No.2, pp.533-536, Nov. 2003.

[4] 정지훈 · 박진기 · 이양선, "자바 언어를 위한 중간 언어 번역기", 멀티미디어학회 2003 추계 학술 발표 대회 논문집, Vol.6, No.2, pp.537-540, Nov. 2003.

[5] 김영근 · 권혁주 · 이양선, "구문 트리를 이용한 자바 바이트코드에서 SIL로의 번역기", 정보처리학회 2004 춘계 학술 발표 대회 논문집, Vol.11, No.1, pp.519-522, Mar. 2004.

[7] 김영근 · 권혁주 · 이양선, "EVM SIL에서 C 프로그램 생성을 위한 역컴파일러의 설계 및 구현", 한국정보처리학회 2005 춘계 학술 발표 대회 논문집, Vol.12, No.1, pp.549-552 ,Mar. 2005.

[8] 손민성 · 배성균 · 이양선, "C++ 컴파일러를 위한 심벌 테이블 역번역기의 설계 및 구현", 한국멀티미디어학회 2005 추계학술발표논문집, Vol.8, No2, pp.181, Nov. 2005