Journal of Korea Multimedia Society Vol. 10, No. 3, March 2007(pp. 411-419)

ANSI C 컴파일러에서 중간코드의 검증과 분석을 위한 역컴파일러의 개발

김영근[†], 권혁주[††], 이양선[†††]

요 약

EVM(Embedded Virtual Machine)은 모바일 디바이스, 셋톱박스, 디지털 TV에 탑재되어 하드웨어에 독립적으로 수행되는 스택기반 가상기계이며, SIL(Standard Intermediate Language)은 EVM의 중간언어로 객체지향 언어와 순차적인 언어를 위한 연산 코드 집합을 갖고 있다. 기존에는 C 프로그램을 실행하기위해 플랫폼에 의존적인 목적코드로 변환하여 실행하였다. 이런 문제를 해결하기 위해 본 연구팀은 EVM을 개발하면서 목적기계의 코드가 아닌 플랫폼에 독립적인 스택기반의 SIL 코드를 생성하는 ANSI C 컴파일러를 개발하였다. 본 논문에서는 ANSI C 컴파일러가 생성한 SIL 코드를 3-주소 코드 형태의 재 표현된 ANSI C 프로그램으로 변환하는 SIL-to-C 역컴파일러(Decompiler) 시스템을 설계하고 구현하였다. 이와 같은 작업은 ANSI C 컴파일러가 생성한 SIL 코드가 올바른지를 확인할 수 있는 검증 방법을 제시하며, 소프트웨어 오작동 및 버전 호환이 이루어지지 않을 때 소프트웨어의 구조를 변경하고 수정하여 성능을 개선하는 작업을 용이하게 한다.

Development of a Decompiler for Verification and Analysis of an Intermediate Code in ANSI C Compiler

YoungKeun Kim[†], HyeokJu Kwon[††], YangSun Lee[†††]

ABSTRACT

Mounted on mobile device, set-top box, or digital TV, EVM is a virtual machine solution that can download and execute dynamic application programs. And the SIL(Standard Intermediate Language) is intermediate language of the EVM, which has a set of opcodes for object-oriented language and a sequential language. Since the C compiler used on each platform depends on the hardware, it converts C program to objective code, and then executes. To solve this problem, our research team developed ANSI C compiler and the EVM. Our ANSI C compiler outputs the SIL code based on stack machine. This paper presents the SIL-to-C decompiler in which converts the SIL code to three address code. Thus, the decompiler allows us to verify SIL code created by ANSI C compiler, and analyze a program from C language source level.

Key words: EVM(임베디드 가상기계), Intermediate Code(중간코드), Compiler(컴파일러), Decompiler (역컴파일러)

※ 교신저자(Corresponding Author) : 이양선, 주소 : 서울시 성북구 정릉4동(136-704), 전화 : 02)940-7292, FAX : 02) 919-0345, E-mail : yslee@skuniv.ac.kr
접수일 : 2007년 1월 19일, 완료일 : 2007년 2월 6일
[†] 준회원, 서경대학교 컴퓨터공학과 박사과정
(E-mail : ykkim@skuniv.ac.kr)
[††] 준회원, 서경대학교 컴퓨터공학과 박사과정
(E-mail : hjkwon@skuniv.ac.kr)
[†††] 종신회원, 서경대학교 컴퓨터공학과 교수
※본 연구는 문화관광부 및 한국문화콘텐츠진흥원의 '06 문화콘텐츠기술(CT) 개발지원사업의 연구결과로 수행되었음.

1. 서 론

EVM(Embedded Virtual Machine)은 모바일 디바이스, 셋톱박스, 디지털 TV에 탑재되어 하드웨어에 독립적으로 수행되는 스택기반 가상기계이며, SIL은 EVM의 중간언어로 객체지향언어인 Java 언어와 순차적 언어인 C 언어를 모두 수용하도록 설계되었다[1-4].

기존의 ANSI C 언어로 작성된 프로그램의 실행은 플랫폼 의존적인 컴파일러를 통해 목적기계의 코드로 변환하여 실행하는 방식이다. 이러한 방식은 플랫폼에 따른 목적코드를 변환하기 위해 플랫폼마다 컴파일러가 필요한 단점이 있다. 또한, 실행되는 플랫폼마다 코드가 다르므로 코드의 재활용성과 이식성이 떨어진다. 이를 보완하는 방법으로 본 연구팀은 스택기반의 가상기계인 EVM과 ANSI C 컴파일러를 개발하였다[5,6]. 기존의 ANSI C 언어로 작성된 프로그램을 본 연구팀이 개발한 ANSI C 컴파일러를 통해서 SIL 코드(*.sil)로 변환하고, 변환된 SIL 코드는 EVM이 탑재되어 있는 임베디드 시스템에서 실행된다. 이를 통해서 이 기종간의 코드 실행 시에 코드를 재 변환할 필요가 없어서 코드의 이식성과 재활용성의 증대를 가져올 수 있다.

본 논문에서는 EVM의 중간코드인 SIL 코드를 재 표현된 C 프로그램으로 변환하는 SIL-to-C 역컴파일러를 설계하고 구현하였다. SIL 코드는 SIL-to-C 역컴파일러에 의해 3-주소 코드 쿼드러플(quadruple) 형태의 재 표현된 C 프로그램으로 변환되고, Visual C 컴파일러를 통해 목적코드로 변환하여 실행하게 된다. 이와 같이 본 논문에서 구현한 SIL-to-C 역컴파일러를 이용하면 ANSI C 컴파일러가 SIL 코드를 올바르게 생성했는지를 확인할 수 있다. 그리고 네트워크와 하드웨어에 의한 소프트웨어 오작동 및 손실과 기존 소프트웨어와의 호환이 이루어지지 않을 때 어셈블리 형태의 SIL 코드보다 소프트웨어의 구조를 변경하고 수정하여 성능을 개선하는 작업을 용이하게 한다. 본 논문의 구성은 다음과 같다. 2장에서는 관련연구로 Java와 .NET 역컴파일러, EVM 가상기계와 중간언어 SIL 코드의 특징을 기술하였다. 3장에서는 SIL 코드를 C 프로그램으로 변환하기 위한 시스템의 개요와 역컴파일러 시스템의 구조 및 구현내용을 기술하였다. 4장에서는 ANSI C언어로 작성된 퀵정렬 프로그램의 컴파일 과정, 최적화 과정, 역컴파일 과정 및 실행 과정을 기술하였다. 끝으로 5장에서는 본 연구에 대한 요약 및 제언을 하면서 결론을 맺는다.

2. 관련연구

2.1 역컴파일러

역컴파일러는 컴파일러가 번역한 목적 코드를 소스 코드로 변환하는 시스템 소프트웨어를 말한다. 기존의 컴파일러는 기계 코드를 내는 방식이고 상당 부분의 정보를 숨겨서 번역하였다. 이러한 이유로 역컴파일러는 목적 코드를 소스 코드로 번역하는데 어려움이 많았고, 상당 부분을 번역하지 못했다[7,8].

자바 언어는 소스 코드를 바이트코드라는 중간 코드로 번역하고 이를 JVM 가상 기계에서 실행한다. 또한 .NET 언어도 MSIL 코드라는 중간 언어로 번역하여 .NET Framework 에서 실행한다. 이러한 중간 언어는 이식성을 강조하기 때문에 자바 바이트코드와 .NET MSIL 코드에는 소스 코드가 가지고 있는 대부분의 정보를 가지고 있고, 번역된 중간 코드는 명령어단위로만 나누어진 쉬운 구조를 지내고 있다.

현재 개발된 자바 역컴파일러는 공개용에서 상업용까지 다양하며, 공개용은 가장 먼저 개발된 자바 역컴파일러인 Mocha[9]와 최근 가장 많이 사용되는 Jad[10] 등이 있다. 그리고 상업용은 이노베이티브 소프트웨어사의 Java 개발환경인 'OEW for Java'의 일부로 제공되는 DejaVu[11]와 윙 소프트사가 개발한 WingDis[12] 등이 있다. 그리고 .NET 역컴파일러는 Remotesoft 사에서 개발한 Salamander[13]와 NET Decompiler에서 개발한 Dis#[14] 등이 있다. 대부분 올바른 번역을 하지만 Mocha는 Boolean 데이터를 정수형으로 번역하고, 배열의 초기화문에 대해서 유효하지 않은 코드로 번역하는 오류를 범하며, Jad는 반복문의 break/continue문과 중첩된 내부 반복문에서 내부와 외부의 break/ continue문을 사용할 때와 일부 예외 문장에 대해서도 처리가 불가능하다. DejaVu는 플로어 해석에 어려움이 있으며, WingDis는 증감연산자중에 선 증감연산자를 후 증감연산자로 오역을 한다. Dis#은 지역 변수의 이름을

자동으로 생성하여 소스 코드의 의미를 살리지는 못하는 단점을 가지고 있다. 하지만 플로우 해석과 변수의 이름의 이름을 생성해서 의미를 살리지 못하는 것은 중간코드에 정보가 담겨있지 않아서 나타나는 것으로 부가적인 디버그 정보를 추가하면 보다 원시 코드와 비슷한 코드로 번역이 가능하다. 또한, 위에서 소개한 자바 역컴파일러는 Jad를 제외하고는 자바 런타임 환경에서 수행이 가능하도록 설계되어서 런타임 환경에 의존적이다.

본 논문에서 구현한 SIL-to-C 역컴파일러는 Jad와 .NET 역컴파일러와 같이 런타임 환경에 구애받지 않고 실행이 가능하다. 또한 자바 역컴파일러와 같이 자바라는 특정 언어에 국한되지 않고 .NET 역컴파일러처럼 다양한 언어를 지원한다. 이는 SIL 코드가 임베디드 시스템을 위한 중간 언어로 ANSI C 언어와 Java 언어 등을 수용하도록 설계되어서 다양한 소스 코드 형태로 변환이 가능하기 때문이다. 따라서 본 논문에서 구현한 SIL-to-C 역컴파일러는 C 프로그램으로 변환하지만 다른 언어로도 변환이 가능하다.

2.2 EVM

EVM은 모바일 디바이스, 셋톱박스, 디지털 TV 등에 탑재되어 동적 응용 프로그램을 다운로드하여 실행할 수 있는 가상기계 솔루션이다[15,16].

EVM은 크게 세 부분으로 구성된다. 첫 번째 부분은 C/C++/C# 또는 자바 등의 고급 프로그래밍 언어로 작성된 프로그램을 가상 기계를 위한 표준 중간 언어(SIL)로 번역하는 부분이다. 두 번째 부분은 SIL 코드들을 입력으로 하여 가상 기계에서 실행 가능한 형태인 EVM 파일 포맷으로 변환하는 어셈블러 부분이다. 마지막으로 세 번째 부분은 실제 하드웨어에 탑재되어 EVM 파일 포맷을 실행하는 가상 기계 부분이다. EVM의 가상 기계 부분은 계층적인 구조로 설계되어 retargeting 과정의 부담을 최소화한다. 다음 그림 1은 EVM의 시스템 구성도이다.

2.3 SIL

EVM의 가상기계 코드인 SIL은 일반적인 임베디드 시스템을 위한 가상기계 코드의 표준화 모델로 설계되었다. SIL은 스택 기반의 명령어 집합으로

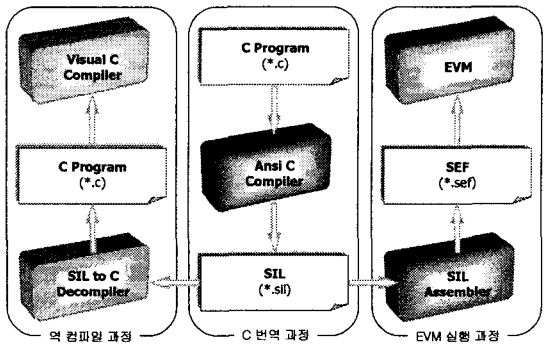


그림 1. EVM 시스템 구성도

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, 객체 지향 언어와 순차적 언어를 모두 수용하기 위한 연산 코드 집합을 갖고 있다.

SIL은 클래스 선언 등 특정 작업의 수행을 나타내는 의사 코드와 실제 명령어에 대응되는 연산 코드로 이루어져 있다[1-4,15,16].

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 정의되어 있다. 또한 최적화를 위한 short form 연산 코드를 갖고 있다.

SIL은 7개의 카테고리로 분류할 수 있으며 각각의 카테고리는 서브 카테고리를 갖는다. 그림 2는 SIL의 연산코드 카테고리 이다.

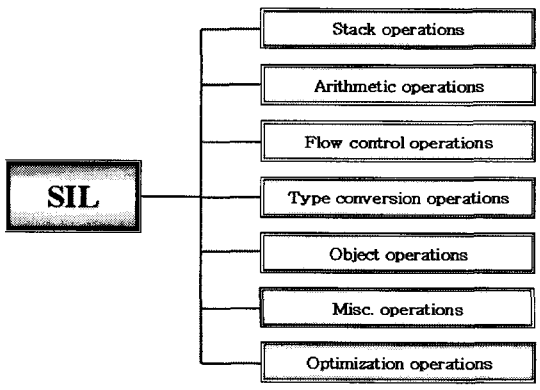


그림 2. SIL의 연산코드 카테고리

3. SIL-to-C 역컴파일러 시스템

3.1 역컴파일러 개요

SIL-to-C 역컴파일러(Decompiler)는 SIL(Standard Intermediate Language) 코드와 테이블 정보를 입력받아서 C 프로그램으로 변환한다. SIL 코드(*.sil)는 데이터 섹션과 코드 섹션으로 구성되고, 테이블 정보(*.stb)는 전역 변수와 타입 그리고 지역 변수와 타입으로 구성된다. 역컴파일러가 변환한 프로그램은 비주얼 C 컴파일러를 통해서 목적코드로 변환되고 실행된다. 그림 3은 역컴파일러의 실행과정을 나타낸 것이다.

3.2 역컴파일러 구성

역컴파일러는 크게 전단부와 후단부로 구성된다. 전단부는 어휘분석기, 구문분석기, SDT 모듈로 구성되며, 전단부에서는 SIL 코드를 토큰으로 분할하고, 문법 검사를 수행하며 중간 트리 형태인 추상 구문 트리(Abstract Syntax Tree)를 출력한다. 또한, 전단부에서는 데이터 섹션에서 필요한 정보를 재구성해 테이블에 저장하고 변환할 때 참조한다. 후단부는 디코더 모듈로 구성되며, 후단부에서는 추상 구문 트리를 탐색하면서 해당 코드에 대한 변환을 한다.

역컴파일하는 과정에서 참조되는 자료구조는 크게 선언부를 위한 심벌 테이블과 문장부를 위한 런타임 스택, API 매핑을 위한 테이블이 존재한다. 심벌 테이블은 변수가 선언될 때 정보를 저장하기 위한 테이블인 동시에 변수가 사용될 때 참조를 위한 테이블로 사용하기 위한 자료구조이다. 런타임 스택은 SIL 코드를 3-주소코드로 변환하는 과정에서 실행코드에 대한 중간 결과 값을 저장하기 위한 자료구조이다. API 매핑 테이블은 표준 라이브러리 함수의 호출에 대해서 그와 동등한 코드로 변환하기 위한 자료구조이다.

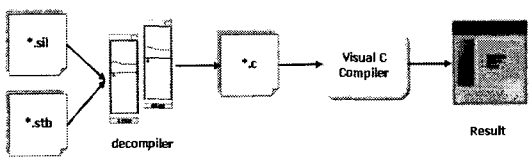


그림 3. 역컴파일러 실행과정

본 SIL-to-C 역컴파일러 시스템은 윈도우즈 XP 환경 하에서 비주얼 C 컴파일러로 구현하였다. 그림 4는 역컴파일러의 시스템 구성도를 나타낸 것이다.

3.3 역컴파일러 구현

역컴파일러는 정형화된 형태로 구성하기 위해 각 모듈단위로 구현되었다. 첫 번째는 SIL 명령어를 분석하여 LALR(1) 문법을 고안하고 파서 생성기(Parser Generating System)에 고안한 문법을 입력하여 어휘정보와 파싱테이블을 얻는다. 두 번째는 생성된 어휘정보에 나타난 토큰의 종류에 따라 입력된 SIL 프로그램을 분류한다. 세 번째는 분류된 토큰과 파싱테이블을 참조하여 구문분석을 한다. 구문분석 중에 올바르지 않은 프로그램은 에러 메시지를 출력하며 종료하고 구문분석이 올바르게 끝이 나면 추상 구문 트리를 생성한다. 네 번째 단계는 디코더 부분으로 생성된 추상 구문 트리를 탐색하면서 C 프로그램으로 변환한다. 변환과정은 선언부 노드와 문장부 노드에 따라 변수, 타입, 함수원형 또는 3-주소코드로 변환한다. 선언부 노드에는 PROGRAM, DATA_SECTION, CODE_SECTION, LOCAL_DCL 등이 있으며 각 노드에서 의미하는 선언문을 심벌 테이블을 참조해서 출력한다. 문장부 노드는 FUNCTION, OPERATION, OPERATOR, OPERANDS 등으로 구성되며 각 노드를 순회하면서 해당 코드에 따른 3-주소 코드로 변환한다. 그림 5는 역컴파일의 알고리즘을 나타낸 것이다.

선언부의 PROGRAM 노드에서는 선언된 사용자 타입과 변환과정 중에 사용된 임시 변수를 출력, DATA_SECTION 노드에서는 전역 변수를 출력, CODE_SECTION 노드에서는 표준 라이브러리와 함수원형을 출력, LOCAL_DCL 노드에서는 지역 변

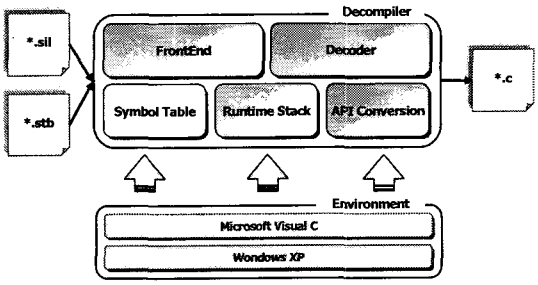
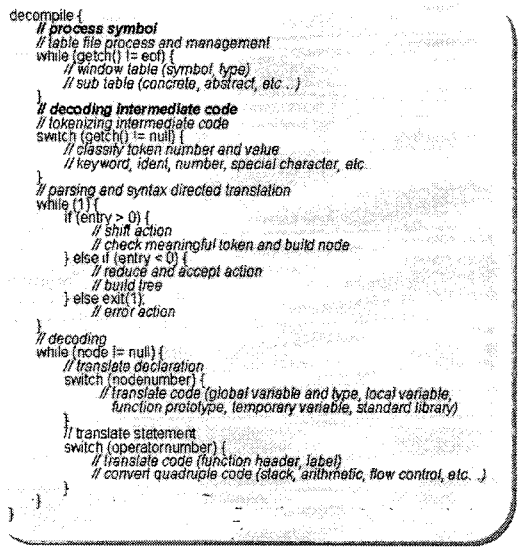


그림 4. 시스템 구성도



```
decompile {
    // process symbol
    // table file process and management
    while (getch() != eof) {
        // window table (symbol, type)
        // sub table (concrete, abstract, etc ..)
    }
    // decoding intermediate code
    // tokenizing intermediate code
    switch (getch() != null) {
        // classify token number and value
        // keyword, ident, number, special character, etc
    }
    // parsing and syntax directed translation
    while (1) {
        if (entry > 0) {
            // shift action
            // check meaningful token and build node
        } else if (entry < 0) {
            // reduce and accept action
            // build tree
        } else exit(1);
            // error action
    }
    // decoding
    while (node != null) {
        // translate declaration
        switch (nodenumber) {
            // translate code (global variable and type, local variable,
               function prototype, temporary variable, standard library)
        }
        // translate statement
        switch (operatornumber) {
            // translate code (function header, label)
            // convert quadruple code (stack, arithmetic, flow control, etc. .)
        }
    }
}
```

그림 5. 역컴파일 알고리즘

수를 출력한다.

문장부의 FUNCTION 노드는 함수 정의를 나타내는 노드로 해당 함수의 이름과 리턴타입, 매개변수의 코드를 출력하며, OPERATION 노드는 하나의 명령어와 명령어 매개변수를 나타내는 노드로 하위에 OPERATOR 노드를 가지고 있다. OPERATOR 노드는 SIL 명령어를 하위 노드로 가지며 명령어에 따라 emitUnary, emitBynary 함수 또는 기타 변환과정을 거친다. 변환과정에서 참조되는 SIL 명령어의 매개변수는 OPERANDS 노드로 이루어진다. OPERANDS 노드는 변수의 베이스(Base)와 오프셋(Offset), 레이블의 정보를 가지고 있다. 베이스와 오프셋 정보를 통해 변수가 전역변수인지 지역변수인지를 심벌 테이블에서 찾고 해당 변수의 이름으로 변환을 한다. 레이블 정보는 분기문과 함수 호출에 대한 분기 위치를 나타내며 분기문의 레이블은 분기 위치로 이동하는 코드로 변환이 되며, 함수 호출의 레이블은 호출하려는 함수가 표준 라이브러리인지를 검사하여 조건에 맞는 변환을 한다. 그림 6은 재표현된 C 프로그램의 출력형태이다.

4. 실험결과 및 분석

다음은 ANSI C 언어로 작성된 퀵소트 프로그램을 ANSI C 컴파일러를 통해 EVM의 중간언어인

```
Store to XXX operations
format : operatorName - rhs;
Increase/Decrease operations
format : tempVarialble = stack[top].operandValue++;
Conditional jump operations
format : tjp - if(operation) goto operand;
         fjp - if(!(operation)) goto operand;
Unconditional jump operations
format : ujp - goto operand;
         ret - return;
         retv.t - return operand;
Function call operations
format : tempVariable - operatorName(operation);
Activation record operations
format : return_type functionName(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, ...)
```

그림 6. 재표현된 C 프로그램 출력 형태

SIL 코드를 생성하고 생성된 코드를 SIL 최적화기를 통해 최적화후 본 논문에서 구현한 SIL-to-C 역컴파일러를 통해서 C 프로그램으로 변환하고, 실행하는 예제이다. 프로그램 1은 ANSI C 언어로 작성된 프로그램을 나타내며, 프로그램 2는 SIL 코드, 프로그램 3은 최적화된 SIL 코드, 프로그램 4는 역컴파일러를 통해 생성된 C 프로그램을 나타낸 것이다. 그림 7은 전체적인 실행과정을 나타낸 것이다.

```
#define _WIN32
#include "C:\Program Files\Microsoft Visual
Studio\VC98\Include\stdio.h"
void qsort(int v[], int left, int right);
void swap(int v[], int i, int j);
void main() {
        int v[] = {6, 3, 1, 4, 7, 0, 5, 2};
        int i, l = 0, r = 7;
        qsort(v, l, r);
...
void qsort(int v[], int left, int right) {
        int i, last;
        if (left >= right) return;
        swap(v, left, (left + right) / 2);
        last = left;
        for (i = left + 1; i <= right; i++)
            if (v[i] < v[left])
                swap(v, ++last, i);
        swap(v,left, last);
        qsort(v, left, last-1);
        qsort(v, last + 1, right);
}
...
```

프로그램 1. ANSI C 프로그램

```
%%Data Section
...
%%Code Section
%File : qsort.c
main: proc          44          1          1
.sym   v            i           0          0
       0            8           32
0x00000006          0x00000003  0x00000001
0x00000004          0x00000007  0x00000000
0x00000005          0x00000002
...
%Line 10 : int i, l = 0, r = 7;
       ldc.i        7
       str.i        1           40
%Line 12 : qsort(v, l, r);
       ldp
       lda          1           0
       lod.i        1           36
       lod.i        1           40
       call         qsort
...
%File : qsort.c
swap: proc          16          1          1
...
%Line 36 : temp = v[i];
       lda          1           0
       lod.i        1           4
       ldc.i        4
       mul.i
       cvi.ui
       cvui.p
       add.p
       ldi.i
       str.i        1           12
...
```

프로그램 2. SIL 코드

```
...
%Line 27 : if (v[i] < v[left]) swap(v, ++last, i);
       ldp
       lda          1           0
       incm.i       1           16
       lod.i        1           16
       lod.i        1           12
       call         swap
$$7:   nop
$$6:   nop
       lod.i        1           12
       incm.i       1           12
       ujp          $$4
$$5:   nop
...
%Line 38 : v[j] = temp;
       setav.i      1           0          1
       8            1           12
```

프로그램 3. 최적화된 SIL 코드

```
#include "qsort.c.sil.opt.sil.h"
/* global sym dcl */
// C SourceFile : qsort.c
void main() {
/* local sym dcl */
int silSym_0[8]={6, 3, 1, 4, 7, 0, 5, 2};
int silSym_1;
int silSym_2;
int silSym_3;
...
void qsort(int v[], int left, int right) {
...
        // C SourceLine(27) : if (v[i] < v[left])
swap(v, ++last, i);
        silSym_8++;
        tmpInt_18 = silSym_8;
        tmpInt_19 = silSym_7;
        swap(v, tmpInt_18, tmpInt_19);
$$7:{/* label not operation */}
$$6:{/* label not operation */}
        tmpInt_20 = silSym_7;
        silSym_7++;
        goto $$4;
$$5:{/* label not operation */}
...
void swap(int v[], int i, int j) {
/* local sym dcl */
int silSym_12;
        // C SourceLine(36) : temp = v[i];
        silSym_12 = v[i];
        // C SourceLine(37) : v[i] = v[j];
        v[i] = v[j];
        // C SourceLine(38) : v[j] = temp;
        v[j] = silSym_12;
}
```

프로그램 4. 역컴파일된 C 프로그램

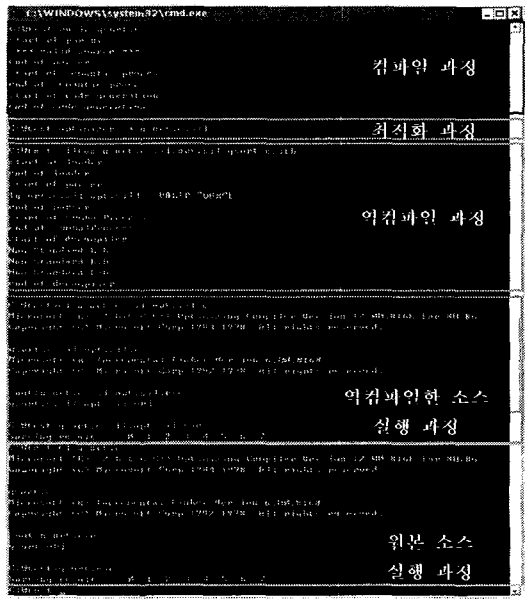


그림 7. 실행화면

다음은 SIL-to-C 역컴파일러 시스템의 성능을 평가하기 위한 실험이다. 표 1은 원시 C 프로그램과 SIL 코드, 최적화된 SIL 코드, 역컴파일러가 생성한 재 표현된 C 프로그램의 코드 크기를 비교한 것이다. 역컴파일된 C 프로그램의 코드 크기는 소스 파일과 헤더 파일 순으로 표시된 것이다.

위의 표 1에 나타난 코드 크기는 역컴파일된 C 프로그램이 상대적으로 더 크다. 이는 역컴파일러의 출력 형태가 3-주소 코드 쿼드러플 방식을 이용하기 때문이다. 이 방식은 우측의 연산 결과를 저장하기위해 임시 변수를 사용한다. 따라서 임시 변수의 사용에 따른 변수 선언을 해야 한다. 또한, 역컴파일러가 생성한 C 프로그램은 텍스트 형식의 크기이며 목적코드로 변환하면 원시 C 프로그램과 비슷한 수준의 코드 크기를 나타낸다.

표 1. 코드 크기 비교

목 록	원시 C 프로그램 (byte)	SIL 코드 (byte)	최적화된 SIL 코드 (byte)	역컴파일된 C 프로그램 (byte)
palindrome_number	561	1970	1936	2092+633
perfect_number	491	1483	1374	1614+473
prime_number	379	1413	1330	1666+505
heap_sort	1543	4860	3561	4726+1081
marge_sort	1594	6370	4870	5465+1039
quick_sort	763	3337	2784	3155+878
magic_square	1052	4335	4014	5514+1361

표 2는 원시 C 프로그램의 실행과 EVM에서 실행, 역컴파일된 C 프로그램의 실행 속도를 비교한 것이다. 측정 방법은 그림 8의 성능 측정 환경에서 각각의 코드를 10번 실행한 평균시간으로 구한 것이다.

EVM
ANSI C Compiler
Visual Studio 6.0
Windows XP Service pack 2
OS - Windows XP professional
CPU - Intel Pentium4 2.8G / RAM - 512M

그림 8. 성능 측정 환경

표 2. 실행 속도 비교

목 록	원시 C 프로그램 (sec)	SIL 코드 (sec)	역컴파일된 C 프로그램 (sec)
palindrome_number	0.101	0.114	0.098
perfect_number	0.119	0.245	0.111
prime_number	0.101	0.123	0.100
heap_sort	0.100	0.198	0.100
marge_sort	0.100	0.103	0.097
quick_sort	0.098	0.131	0.098
magic_square	0.105	0.161	0.108

원시 C 프로그램과 역컴파일된 C 프로그램은 같은 환경의 비주얼 C 컴파일러를 이용해서 목적 코드로 변환한 후 실행한 것이며, SIL 코드는 EVM 가상 기계에서 실행한 것이다.

위의 표 2에서 나타난 실행 속도는 EVM의 실행 속도가 조금 느린 속도를 나타낸다. 이는 가상 기계의 단점으로 대부분 명령어 패치에 걸리는 시간이다. 하지만, 원시 C 프로그램과 역컴파일된 C 프로그램의 실행 속도는 비슷하게 나타나는 것을 볼 수 있다. 일부 역컴파일된 C 프로그램이 조금 빠르게 나타난 것은 역컴파일된 C 프로그램이 3-주소 코드의 형태이기 때문에 나타나는 현상으로 예상된다.

5. 결 론

본 논문에서는 스택기반의 가상기계인 EVM의 중간언어 SIL 코드를 재 표현된 C 프로그램으로 변환하는 SIL-to-C 역컴파일러를 설계하고 구현하였다. 역컴파일러는 기존의 컴파일러의 정형화 기법을 이용하여 어휘분석기, 구문분석기, SDT 모듈의 전단부와 3-주소 코드의 쿼드러플 형태로 출력하는 디코더 모듈로 구성하였다. 따라서, 모듈별로 재사용이 가능하다. 역컴파일러를 이용하여 SIL 코드를 재 표현된 C 프로그램으로 변환하고, 변환된 C 프로그램은 Visual C 컴파일러를 통해서 목적코드로 변환되고 실행하게 된다. 이와 같이 개발한 역컴파일러를 통해 SIL 코드가 올바르게 번역되었음을 확인할 수 있다. 그리고 네트워크와 하드웨어에 의한 소프트웨어 오작동 및 손실과 기존 소프트웨어와의 호환이

이루어지지 않을 때 어셈블리 형태의 SIL 코드보다 소프트웨어의 구조를 변경하고 수정하여 성능을 개선하는 작업을 용이하게 한다.

본 역컴파일러 시스템은 ANSI C 언어와 같은 순차적인 언어에 대한 SIL 코드와 최적화 코드만 역컴파일한다. 하지만 EVM이 순차적인 언어와 객체 지향 언어를 수용하는 가상 기계이고 SIL 코드는 객체 지향 코드도 정의되어있다. 따라서, 향후 연구로 SIL 코드의 객체 지향 관련 코드에 대한 역컴파일 과정도 추가할 예정이다. 또한, 원시 코드의 분석이 용이하도록 하기위해서 SIL 코드를 분석하여 보다 원시 코드와 비슷한 코드를 출력하기위한 연구를 병행할 예정이다.

참 고 문 헌

[1] 권 혁주, 김 영근, 박 진기, 이 양선, "임베디드 C 컴파일러를 위한 EVM SIL 코드 최적화기," 한국멀티미디어학회, 2005 춘계학술발표논문집, 제8권, 제1호, pp. 108-111, 2005. 5.

[2] 김 영근, 권 혁주, 이 양선, "구문 트리를 이용한 자바 바이트코드에서 SIL로의 번역기," 한국정보처리학회 2004 춘계학술발표대회논문집, 제11권, 제1호, pp. 519-522, 2004. 5.

[3] 김 영근, 권 혁주, 이 양선, "자바 바이트코드의 EVM SIL 코드 매핑," 한국멀티미디어 학회 2004 추계학술발표논문집, 제7권, 제2호, pp. 915-918, 2004. 11.

[4] 남 동근, 윤 성림, 오 세만, "가상 기계를 위한 어셈블리 언어," 한국정보처리학회 2003 춘계학술발표논문집, 제10권, 제1호, pp. 783-786, 2003. 5.

[5] Christopher Fraser and David Hanson, *A Retargetable C Compiler : Design and Implementation*, Addison-Wesley, 1995.

[6] Brian W.Kernighan and Dennis M.Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[7] 최 은만, "역공학을 이용한 소프트웨어 재사용 시스템에 관한 연구," 한국정보처리학회 논문지, 제4권, 제1호, 1997. 1.

[8] 권 혁주, 김 영근, 박 진기, 이 양선, "ANSI C 컴파일러를 위한 심볼 테이블로부터 C 프로그램 역번역기의 개발," 한국멀티미디어학회 2005 춘계학술발표논문집, 제8권, 제1호, pp. 69-72, 2005. 5.

[9] Hanpeter Van Vliet, Mocha, http://www.brouhaha.com/~eric/com puters/mocha.html.

[10] Pavel Kouznetsov, Jad, http://www.kpdus.com/jad.html.

[11] Innovative Software, DejaVu, http://www.isg.de/OEW/Java.

[12] Wingsoft, WingDis, http://www.wingsoft.com/

[13] Remotsoft, Salamender, http://www.remotesoft.com/salamander/index.html.

[14] Netdecompiler, Dis#, http://netdecompiler.com/index.html.

[15] Yang-Sun Lee and Jin-Ki Park, "Translation Java Bytecode to EVM SIL Code for Embedded Virtual Machine," *Korea MultiMedia Society*, Vol. 8, No. 12, pp. 827-835, 2005. 12.

[16] 이 양선, "임베디드 시스템을 위한 가상기계 기술," 한국멀티미디어 학회지, 제6권, 제2호, pp. 36-44, 2002. 6.

김 영 근

2004년 서경대학교 컴퓨터공학과(공학사)

2006년 서경대학교 대학원 컴퓨터공학과(공학석사)

2006년 3월 ~ 현재 서경대학교 컴퓨터공학과 박사과정

관심분야 : 프로그래밍 언어, 유비쿼터스 컴퓨팅, 임베디드 시스템, IPTV 기술

권 혁 주

2004년 서경대학교 컴퓨터공학과(공학사)
2006년 서경대학교 대학원 컴퓨터공학과(공학석사)
2006년 3월 ~ 현재 서경대학교 컴퓨터공학과 박사과정
관심분야 : 프로그래밍언어, 유비쿼터스 컴퓨팅, 임베디드 시스템, IPTV 기술

이 양 선

1985년 동국대학교 전자계산학과(공학사)
1987년 동국대학교 대학원 컴퓨터공학과(공학석사)
1993년 동국대학교 대학원 컴퓨터공학과(공학박사)
1994년 3월~현재 서경대학교 컴퓨터공학과 교수
1996년 3월~2000년 2월 서경 대학교 전자계산소 소장
2000년 2월~현재 한국멀티미디어학회 이사
2005년 1월~2006년12월 한국멀티미디어학회 총무이사
2004년 5월~현재 한국정보처리학회 게임연구회 부위원장
2005년 12월~현재 한국정보처리학회 게임연구회 위원장
2006년 1월~현재 한국정보처리학회 이사
관심분야 : 프로그래밍언어, 유비쿼터스 컴퓨팅, IPTV 기술, CT 기술, 임베디드 시스템 등