역컴파일링 기법을 이용한 가상기계 코드 실행 시스템

안 덕 기[†] · 이 창 환[††] · 오 세 만[†††]

요 약

일반적으로, 가상기계 플랫폼은 컴파일러와 어셈블러, 가상기계로 구성되어 있다. 가상기계 코드의 설계는 가상기계 플랫폼을 만드는데 필수적이며, 설계 결과물인 가상기계 코드의 검증은 매우 중요하다. 가상기계 코드의 검증과 코드의 실행을 위해서는 가상기계 코드의 실행 시스템을 구현하는 것이 필요하고, 컴파일링 기법과 인터프리팅 기법, 역컴파일링 기법으로 실행 시스템을 만들 수 있다.

본 논문에서는 가상기계 코드 실행을 위한 3가지 방법 중에서 역컴파일링 기법을 이용한 가상기계 코드 실행 시스템을 제안 및 구현한다. 제안하는 실행 시스템은 가상기계 코드로 EVM(Embedded Virtual Machine)의 중간언어인 SIL(Standard Intermediate Language)을 사용하였고, 이를 통해 역컴파일의 유용성을 확인하였다. 또한 제안한 역컴파일 기법은 가상기계 플랫폼을 구현할 때 발생할 수 있는 오류의 최소화에 사용할 수 있다

키워드 : 가상기계 시스템, 가상기계 코드, 역컴파일링 기법

Executing System of Virtual Machine Code using Decompiling Method

Dukki Ahn[†] · Changhwan Yi[††] · Seman Oh[†††]

ABSTRACT

Generally, Virtual machine platform is composed of a compiler, an assembler, and VM(Virtual Machine). To develop it, the design of VMC(Virtual Machine Code) is an essential task. And it is very important to verify the virtual machine platform. To do this and furthermore to execute VMC, it needs to implement VMC execution system using compiling method, interpreting method, or decompiling method.

In this paper, we suggested and implemented the executing system of VMC using decompiling method out of three methods to execute the VMC. In our implementation, the VMC is SIL(Standard Intermediate Language) that is an intermediate code of EVM(Embedded Virtual Machine). Actually, we verified the usefulness of the decompiling method. And the decompiling method suggested in this paper can be used to minimize the mistakes in developing Virtual machine platform.

Key Words : Grid, Virtual Machine System, Virtual Machine Code, Decompiling Method

1. 서 론

최근 임베디드 유비쿼터스 컴퓨팅을 위한 연구가 강조되고 있으며, 이를 위한 가상기계 플랫폼 구축 기술은 필수적이다. 가상기계 플랫폼은 고급 언어를 어셈블리어 수준의 가상기계 코드로 번역하는 컴파일러의 구현, 번역된 코드를 가상기계에서 실행 가능한 구조로 변환하는 어셈블러의 구현, 변환된 코드를 해석해서 실행하는 가상기계의 구현으로 구축된다.

이러한 플랫폼을 구축하기에 앞서 가상기계 코드를 설계하는 것은 필수적이며, 가상기계 코드가 정의된 후에는 반드시 가상기계 코드의 논리성을 확인하기 위한 시험 과정이 필요하다. 가상기계 코드를 시험하기 위한 가장 신뢰적인 방법은 모의 실행 시스템을 구현하여 가상기계 코드의 실행 결과를 확인하는 것이다.

가상기계 코드를 실행시키는 방법은 가상기계 코드가 실행되는 환경에 따라서 컴파일링 기법, 인터프리팅 기법, 역컴파일링 기법으로 구분할 수 있다. 컴파일링 기법은 후단부의 코드 생성 과정에서 전단부의 심벌 테이블을 참조하므로, 가상기계 코드 자체의 완전성을 증명하기에는 부족하다. 그리고 인터프리팅 기법은 인터프리터를 구현하는데 비교적 많은 시간과 노력이 요구된다. 역컴파일링 기법은 가상기계 코드를 고급 언어로 변환하고, 검증된 기존 컴파일러를 이용해서 실행시키는 방법이다. 이러한 기법은 시험 데이터로서 가상기계 코드만을 요구하므로, 시험 결과에 있어서 컴파일링 기법보다 신뢰적일 것이며, 실행 환경 구현에 있어

† 준 회 원 : 동국대학교 일반대학원 컴퓨터공학과 석사
†† 종신회원 : 동국대학교 산업기술연구원 겸임교수
††† 종신회원 : 동국대학교 컴퓨터공학과 교수
논문접수 : 2006년 9월 26일, 심사완료 : 2007년 3월 8일

서 기존의 컴파일러를 이용하므로 인터프리팅 기법보다 용이하다[1].

따라서, 본 논문에서는 가상기계 코드를 역컴파일하여 실행하는 기법을 제안하고, 그러한 기법을 EVM(Embedded Virtual Machine)의 중간 언어인 SIL(Standard Intermediate Language)에 적용하여 실제 가상기계 코드 실행 시스템을 구현한다. 본 논문의 시스템은 가상기계 플랫폼을 체계적으로 구축하는데 효과적으로 이용될 수 있으며, 가상기계 플랫폼이 구축된 후 발생될 수 있는 오류를 최소화하는데 도움이 될 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 최근의 여러 가상기계 코드를 토대로 하여 역컴파일 과정에서 요구될 수 있는 가상기계 코드의 특징을 기술하고, 이어서 EVM과 역컴파일링 기법에 대해서 기술한다. 3장에서는 본 논문에서 제안하는 가상기계 코드 실행 시스템의 구조를 기술한다. 4장에서는 본 논문의 역컴파일링 기법을 적용하여 SIL 코드 실행 시스템을 구현하고 실험 결과를 기술한다. 마지막으로, 5장에서는 본 논문의 결론을 맺고 향후 연구를 기술한다.

2. 관련 연구

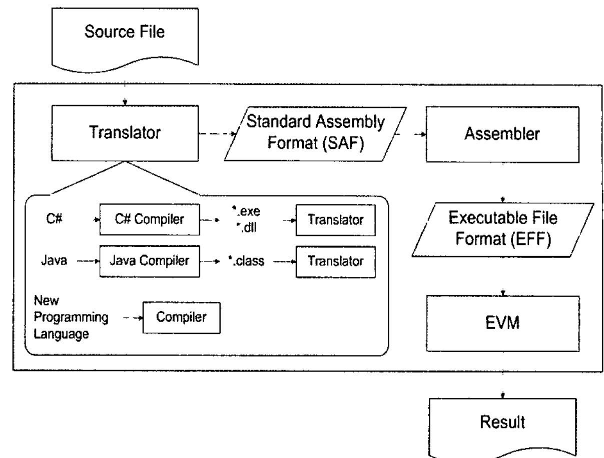
2.1 가상기계 코드

가상기계 코드란 가상기계에서 즉시 실행될 수 있는 이진 코드와 1:1로 대응되는 명령어의 집합을 말하며, 연산 방식에 따라서 스택 기반 코드와 레지스터 기반 코드로 구분할 수 있다. 이러한 가상기계 코드는 일반적으로 스택 기반 코드로 설계되며, 그러한 이유는 다음과 같은 장점이 있기 때문이다. 스택 연산은 피연산자의 위치를 묵시적으로 지정함으로써 복잡한 레지스터 할당기의 구현이 불필요하며, 컴파일러에서 스택 코드의 생성이 용이하고, 0-주소 코드로서 인터프리터의 구현이 용이하다. 따라서 가상기계 플랫폼을 보다 능률적으로 구축 할 수 있다. 그러나 최근의 연구 결과에 따르면 스택 기반 코드는 레지스터 기반 코드와 비교하여 명령어의 수, 바이트 코드 파일의 크기, 실행 시간 측면 등에서 단점이 있다[2, 3]. 대표적인 스택 기반 가상기계 코드는 Java bytecode[4], .NET IL[5], EVM SIL[6]이 있으며, 레지스터 기반 가상기계 코드는 Parrot bytecode[7]가 있다.

이러한 가상기계 코드의 일반적인 어셈블리 구조는 크게 데이터부(data section)와 코드부(code section)로 구분할 수 있다. 데이터부에는 연산에 이용될 데이터 정보가 표현되어 있으며, 코드부에는 가상기계 코드가 실행되는 연산 정보가 명령어로서 표현되어 있다. 이러한 데이터부와 코드부는 가상기계 코드가 수용하는 고급 언어의 기능, 특성, 수에 따라서 의사 코드와 연산 코드가 규칙적으로 산재되어 있다.

2.2 EVM

EVM은 모바일 장치, 셋톱박스, 디지털 TV 등에 탑재되어 동적으로 응용 프로그램을 다운로드하여 실행할 수 있는 솔루션이다. 또한 컨텐츠 개발을 쉽게 하기 위해서 다양한 언어를 지원하고 언어들 간에 통합도 가능하다. 이 시스템은 고급 언어로 작성된 프로그램을 SAF(Standard Assembly Format)로 변환하는 번역기, SAF를 입력 받아 EVM에서 실행 가능한 파일 포맷(EFF;Excutable File Format)을 생성하는 EFF 생성기, 하드웨어에 탑재되어 EFF를 실행하는 EVM으로 (그림 1)과 같이 구성된다[8].

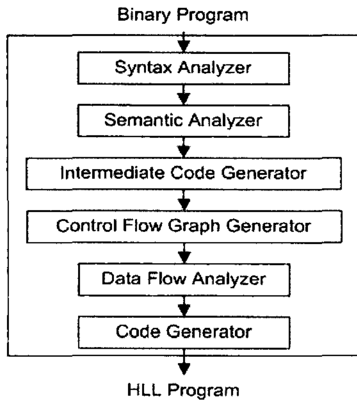


(그림 1) EVM 시스템 구성도

EVM의 가상기계 코드인 SIL은 다양한 고급 언어를 수용하기 위해 P-기계의 P-Code, EM 기계의 EM-Code, JVM의 bytecode, .NET의 .NET IL 등 기존의 가상기계 어셈블리 언어들의 분석을 토대로 정의되었다. 또한, 객체지향 언어와 순차적 언어를 모두 수용하기 위한 연산코드 집합을 갖고 있으며, 그러한 연산코드의 범주는 크게 산술 연산 명령, 제어 흐름 명령, 스택 운용 명령, 형 변환 명령, 객체 관련 명령, 기타 명령들로 구성된다.

2.3 역컴파일링 기법

역컴파일링 기법은 1960년 Maurice Halstead의 연구를 시작으로 1980년대까지 모두 경험적이며 개별적인 기법으로 제안되어 왔으며, 이러한 연구의 논점은 다음 3가지가 중심이 되었다. 이진 코드를 코드부와 데이터부로 구분하여 어셈블리 코드로 변환하는 역어셈블링 기법, 고급 언어 수준의 제어구조를 구성하기 위한 제어 흐름 분석 기법, 고급 언어 수준의 자료형을 추출하기 위한 타입 분석 기법이다[9-12]. 이러한 기법들을 기반으로 1994년 Cristina Cifuentes는 일반적인 역컴파일링 기법을 제안하였으며, 그 단계는 (그림 2)와 같다[12]. 최근의 레지스터 기반 코드 역컴파일러는 모두 이와 같은 과정으로 구현되었다[13-16]. 이상과 같은 연구의 대부분은 역컴파일률(본래 소스 코드와의 유사율)을 높이는 것에 치중하였으므로, 본 논문의 실행 시스템에 적용하기에는 비효율적이다. 또한 스택 기반 코드의 역컴파일 기법은 모두 공개하지 않고 있다. 따라서 본 논문에서는 (그림 2)의 제어 흐름 그래프 생성기와 데이터 흐름 분석기의 변형으로 기본 블록 생성기와 데이터 테이블 생성기를 정의하여 역컴파일러를 구현하고 이를 이용한 실행 시



(그림 2) Cifuentes의 역컴파일링 단계

스템을 제안한다.

역컴파일러란 컴파일된 기계 코드나 어셈블리 코드로부터 컴파일 전의 본래 소스 코드를 추출해 내는 프로그램을 의미한다. 일반적으로 원 소스 코드와 동일한 소스 코드를 추출할 수 없으므로, 의미적으로 동등한 고급 언어 프로그램이 생성되도록 구현된다. 이것을 만드는 일반적인 목적은 컴파일된 프로그램의 검사, 알고리즘의 이해, 크로스 플랫폼에서의 실행, 최적화된 실행 파일의 생성, 실행 파일 에러의 수정 등을 위해서이다. 이러한 역컴파일러의 추출률은 데이터부의 명시적인 정보 또는 명시적인 정보의 조합으로부터 유추될 수 있는 데이터의 정보에 따라서 결정된다.

3. 가상기계 코드 실행 시스템

본 논문에서 제안하는 실행 시스템은 (그림 3)과 같이 역컴파일러와 실행부로 나눌 수 있다. 역컴파일러는 전단부와 후단부로 구성되며, 실행 시스템의 주요 기능을 수행한다. 이 역컴파일러는 다음과 같은 제한을 가지고 설계되었다. 첫째, 역컴파일러의 입력 소스는 어셈블리 수준의 가상기계 코드이다. 둘째, 복잡한 제어 흐름 분석을 수행하지 않기 위해서 출력 고급 언어의 반복문은 모두 조건문과 분기문의 조합으로 구성한다.

3.1 역컴파일러

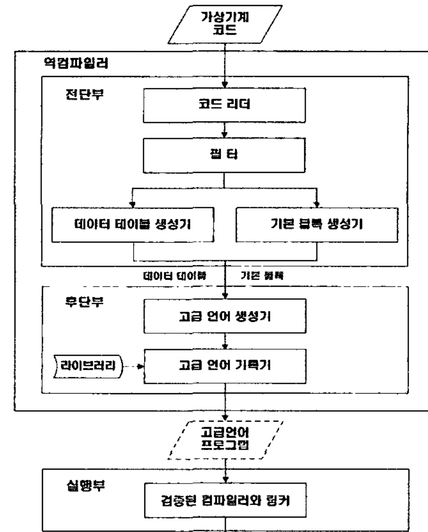
역컴파일러는 가상기계 코드에 관계된 작업을 처리하는 전단부와 고급 언어와 관련된 작업을 처리하는 후단부로 구성된다.

(1) 전단부

전단부는 후단부에서 참조할 데이터 테이블과 처리할 기본블록을 생성하며, 코드 리더, 필터, 데이터 테이블 생성기, 기본 블록 생성기로 구성된다.

코드리더는 어셈블리 파일을 읽어 들여 다음 과정에서 처리할 수 있도록 메모리에 저장하는 역할을 한다.

필터는 역컴파일 과정에서 불필요한 코드를 제거하며, 제



(그림 3) 가상기계 코드 실행 시스템

거되는 코드에는 어셈블리의 주석문과 라이브러리를 사용하기 위한 의사 코드가 있다. 라이브러리 사용을 위한 의사 코드에는 일반적으로 라이브러리 내부의 구조형, 함수, 변수, 상수 정보가 포함된다. 이러한 정보는 미리 정의한 라이브러리 의사 코드 목록을 참고하여 제거한다.

데이터 테이블 생성기는 필터링된 가상기계 코드의 데이터부를 참조하여 구조형, 함수, 변수, 문자열 상수 테이블을 생성한다.

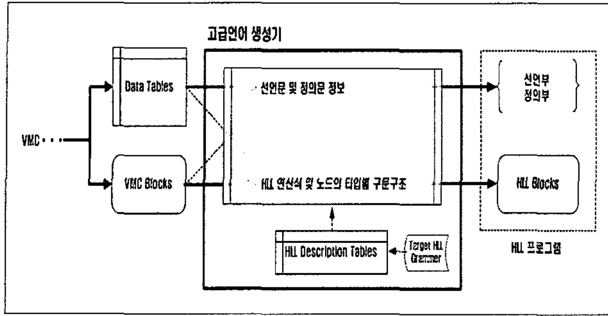
기본 블록 생성기는 필터링된 가상기계 코드의 코드부를 참조하여 기본 블록을 생성한다. 이 기본 블록은 고급 언어의 기본문을 기준으로 구분되며, 배정문과 제어문, 호출문, 반환문이 있다. 또한 이 기본문에 따라 기본 블록은 배정형, 무조건 분기형, 조건 분기형, 호출형, 반환형으로 구분할 수 있다. 이런 기본 블록의 종류는 가상기계 코드부에서 배정 명령어와 무조건 분기 명령어, 조건 분기 명령어, 함수 호출 명령어, 반환 명령어를 찾아 구분한다.

(2) 후단부

후단부는 전단부에 의해서 생성된 데이터 테이블을 참조하면서 기본 블록을 처리하여, 고급 언어 프로그램을 생성한다. 이러한 후단부는 고급 언어 생성기와 고급 언어 기록기로 구성된다.

고급 언어 생성기는 미리 정의한 고급 언어 묘사 테이블 (HLL Description Table)과 전단부에서 만들어진 데이터 테이블을 참조하여, 기본 블록을 고급 언어문으로 변환한다. (그림 4)는 고급 언어 생성기를 개념적으로 표현한 것으로서, 해석적 코드 생성 기법에 기초하여 기본 블록을 고급 언어로 변환하는 과정을 보여주고 있다.

고급 언어 생성기의 고급 언어 묘사 테이블(HLL Description Tables)은 목표 고급 언어의 특성을 미리 기술한 것으로서, 목표 고급 언어에 따라 변경되는 부분이다. 여기에는 목표



(그림 4) 고급 언어 생성기의 구조 및 기능

고급 언어의 지정어, 자료형, 연산자, 구문 구조와 같은 정보가 있다.

기본 블록을 고급 언어문으로 변환하는 루틴은 (그림 5)와 같이 기본 블록으로 구분된 명령어를 고급 언어의 연산식으로 변환하는 공용 처리와 각 블록의 형에 맞는 문장을 구성하는 전용 처리로 구성된다. 이와 같이 공용 처리와 전용 처리로 분리한 이유는 구성된 각 블록에 스택 운용 명령어와 연산 명령어가 반복된 후 코드의 마지막에 각 블록의 고유한 명령어가 나타나기 때문이다.

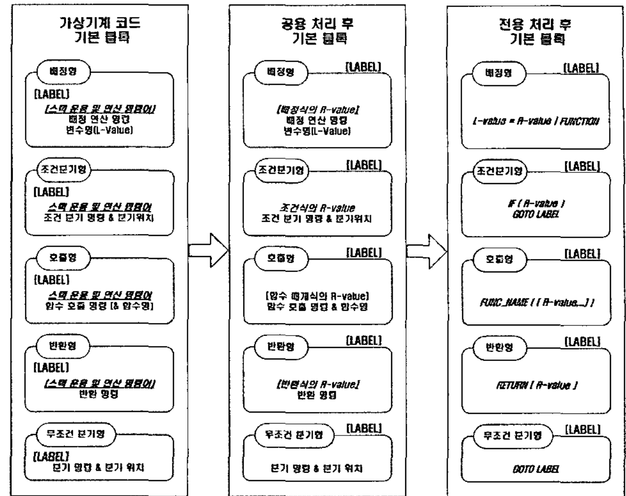
공용 처리는 역폴리시 표기법(RPN;Reverse Polish Notation)으로 표현된 모든 연산 명령어를 사이 표기법(IN;Infix Notation)으로 변환한다. 또한 고급 언어에서는 묵시적으로 표현되는 자료형이 가상기계 코드에서는 명시적으로 표현되므로, 변수의 자료형을 나타내는 타입 기호는 이 과정에서 무시될 수 있으며, 후에 데이터 테이블을 참조하여 정의문과 선언문 코드를 생성함으로써 보완된다. 그러나 의사 코드로부터 완전한 데이터 테이블을 생성하기 어려운 경우에는 변수의 이름에 타입 정보를 추가해야 한다. 또한 연산자는 고급 언어 묘사 테이블을 참조하여 추출되며, 구조형과 변수 정보는 데이터 테이블을 참조하여 추출된다. 이 과정에서 각 블록의 분기 레이블을 블록의 멤버 요소로 구성함으로써 블록을 간소화할 수 있다.

전용 처리는 데이터 테이블과 고급 언어 묘사 테이블을 참조하여 정의부 및 선언부를 구성하고, 공용 처리된 각 블록을 완전한 고급 언어문으로 구성한다. 정의부 및 선언부 구성 루틴은 데이터 테이블과 고급 언어 묘사 테이블을 참조하여, 구조형 및 함수를 기준으로 정의문 및 선언문을 배치한다. 정의부는 클래스, 인터페이스, 또는 구조체, 공용체의 정의를 말하며, 선언부는 정의형의 선언과 함수 및 변수의 선언을 말한다.

고급 언어 기록기는 가상기계 코드에서 사용한 라이브러리 목록을 고급 언어 생성기의 결과 코드에 추가하여 컴파일이 가능한 고급 언어 코드를 생성한다. 라이브러리문은 호출형 블록내의 함수명과 함수 테이블을 비교하여 생성한다.

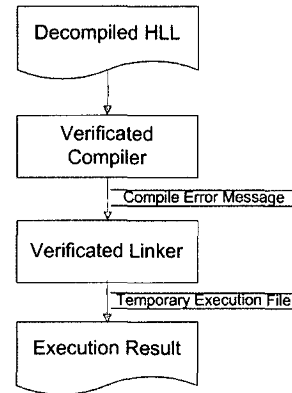
3.2 실행부

실행부는 (그림 6)과 같이 구성되며, 역컴파일러에 의해서 생성된 고급 언어 프로그램을 기존의 컴파일러와 링커를 이용하여 실행한 후 결과를 출력한다. 이 과정에서 컴파일 에



※ 'R-value'는 배정 연산자가 제외된 식을 의미, '[ ]'는 없을 수도 있음을 의미, '|'는 택일을 의미.

(그림 5) 기본 블록 변환 과정



(그림 6) 실행부 구성도

러 또는 실행 에러가 발생되는 경우, 에러 메시지가 실행부로 전달되어 출력된다.

4. 실험

실행 시스템의 구현은 C# 언어를 사용하였으며, 개발 도구로는 Microsoft사의 Visual Studio .NET 2003을 사용하였다. 또한 EVM의 중간 언어인 SIL을 대상으로 가상기계 코드 실행 시스템을 구현하고 시험하였으며, 목표 고급 언어는 C 언어를 선택하였다. 생성된 결과 프로그램의 실행에는 Microsoft사의 Visual C++ 6.0을 사용하였다.

4.1 구현 과정

코드 리더와 필터의 구현은 미리 정의한 라이브러리 목록과 C#의 문자열 관련 메소드를 이용하였다. 데이터 테이블 생성기는 <표 1>과 같은 SIL 코드의 의사 코드를 참조하여 구성한다. <표 1>의 지시어를 매개 변수로 하여 테이블 목록을 얻어오고, 데이터 테이블 어댑터를 통하여 각 테이

블을 생성한다.

데이터 테이블의 자료 구조는 모두 클래스로 정의하였으며, 각 클래스들은 클래스간의 연관성에 따라서 상속 또는 내재 관계로 구성하였다. (그림 7)은 이러한 테이블의 구조를 나타낸다.

기본 블록의 자료구조는 리스트를 사용하였으며, 블록의 형을 구분하기 위해서 <표 2>와 같이 배정형은 "ASGN", 무조건 분기형은 "JUMP", 조건 분기형은 "CJUMP", 호출형은 "CALL", 반환형은 "RET"로 정의하였다.

C 코드 생성기는 패턴 처리, 공용 처리, 전용 처리로 구성된다. 패턴 처리 단계는 먼저 블록 처리의 복잡성을 감소시키기 위해서, 블록을 순회하면서 분기 위치가 되는 레이블을 설정한다. SIL 코드에서 레이블은 "$$순번:\tnop"와 같이 표현되며, 연속적으로 여러 개의 레이블이 올 수 있다. C 코드 생성기의 마지막 과정인 전용 처리 단계 외에는 레이블을 전혀 참조하지 않기 때문에 이 레이블은 따로 보관되도록 설정된 후 블록에서 제거된다. 또한, 참조 연산 명령과 기본 연산 명령에서 공통으로 이용되는 명령어를 쉽게 구분하기 위해서 참조 연산을 의미하는 명령어를 기준으로 패턴을 정의하여 사용하였다. 이에 따라서 정의된 패턴은 구조형, 배열, 사용자의 입력을 요구하는 내장 함수와 관련된 것으로서, 이들 모두 참조 연산으로 간주하여 처리한다.

공용 처리 단계에서는 SIL 코드를 사이 표기법으로 변환하여, <표 3>과 같은 각 블록의 명령어를 고급 언어의 R-value로 변환한다. 정의한 스택 연산이 수행된 후의 스택에는 공용 처리의 결과만 남게 되며, 처리되지 않은 SIL 명령어는 "NEXT"로 표기되어 전용 처리 단계에서 처리된다. 또한 공용 처리에서는 구조형 변수의 목표 자료형이 정해진다. 정해진 자료형은 전용 처리 단계에서는 정의부와 선언부를 구성하는데 사용한다.

전용 처리 단계는 구조형 정의와 함수 및 변수 선언문을 구성한 후, 함수의 헤더문을 구성한다. 또한 지역 변수 선언문을 구성하고, 각 블록의 전용 처리를 수행한다. 이 결과에 라이브러리 바인딩문만 추가하면, 바로 컴파일한 후 실행이 가능한 프로그램이 된다. 정의부 및 선언부를 생성하는 루틴은 데이터 테이블을 참조해가며, C 언어 묘사 코드에 따라 수행된다. 블록 전용 처리 루틴에서는 공용 처리 단계에서 미구성된 코드를 완전한 C 구문으로 구성한다. 구성 과정 중 "CALL" 블록과 "ASGN" 블록의 경우 병합이 이루어질 수 있다. 이 결과에서 요구되는 헤더 파일은 함수 테이블과 "CALL"블록을 비교하여 내장 함수를 추출하고, 추출된 내장 함수를 라이브러리 함수와 비교하여 함수가 정의된 헤더 파일을 알아낸다. 알아낸 헤더 파일 정보를 추가하여 최종적인 C 프로그램을 생성한다.

마지막으로 실행 단계에서는 C 프로그램을 컴파일과 동시에 실행 파일을 임시적으로 생성하며 결과를 출력한다.

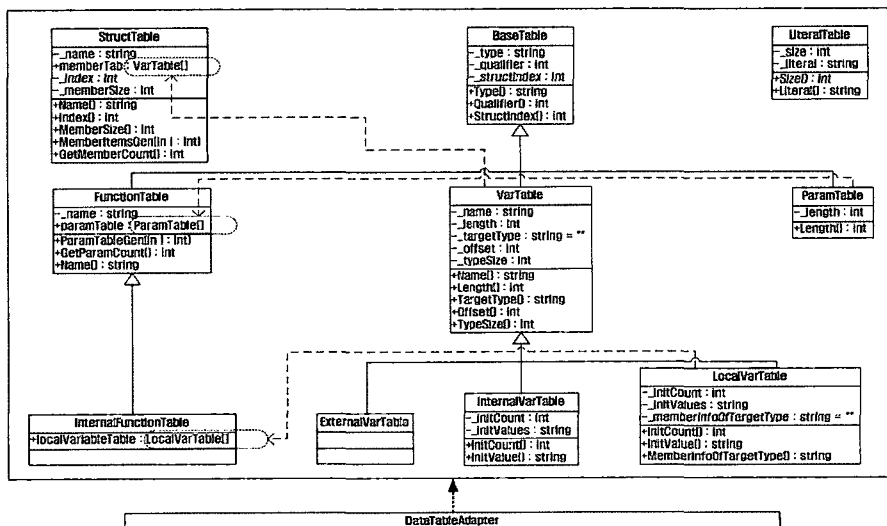
<표 1> 데이터 테이블 구성을 위한 SIL 의사 코드

테이블	의사 코드
리터럴 상수	.literal\t[Offset]\t[Size]\t[Data]
구조형	.struct\t[Name]\t[Member Count]\t[Members]
외부 변수	.symx\t[Name]\t[Type]\t[Qualifier]\t[Struct Index]\t[Offset]\t[Length]
내부 / 지역 변수	.sym\t[Name]\t[Type]\t[Qualifier]\t[Struct Index]\t[Offset]\t[Length]\t [Init Count]\t[Init Values]
내부 / 외부 함수	.func\t[Name]\t[Type]\t[Qualifier]\t[Struct Index]\t[Param Count]\t [Param Types]

※ '\t'는 공백 'tab'을 의미.

<표 2> SIL 기본 블록 구분 명령어

블록형	명 령 어
배정형	str.c, str.s, str.i, str.u, str.l, str.p, str.f, str.d, str.t, sti.c, sti.s, sti.i, sti.u, sti.l, sti.p, sti.f, sti.d, sti.t
무조건 분기형	ujp
조건 분기형	tjp, fjp
호출형	call, calli, calls
반환형	ret, retv.c, retv.s, retv.i, retv.ui, retv.l, retv.p, retv.f, retv.d, retv.t



(그림 7) 데이터 테이블의 클래스 구조

〈표 3〉 공용 처리 스택 운용 명령어

스택 명령어	수행 작업	적용 명령어
PUSH	삽입	PUSH, ldp, ldc.[c,s,i,u,l,p,f,d], lda, lod.[c,uc,s,us,i,ui,l,ul,p,f,d,t], ldi.[c,uc,s,us,i,ui,l,ul,p,f,d,t], ldftn
POP_OP_PUSH	추출 후 삽입	neg.[i,l,f,d], bcom.[i,l], not.[i,l], inc.[i,ui,l], dec.[i,ui,l]
POP_CV_PUSH	추출 후 삽입	cvc.[i,ui], cvs.i, cvi.[c,s,ui,l,d], cvui.[c,i,ul,p], cvl.[i,ul,d], cvul.[ui,l], cvp.ui, cvf.d, cvd.[i,l,f]
PEEK	맨 위의 값 복사 추출 후 수정해서 삽입	PEEK
KEEP_POP	계속해서 추출 후 삽입	call, calli, calls
POP2_OP_PUSH	두 번 추출 후 삽입	add.[i,ui,l,p,f,d], sub.[i,ui,l,p,f,d], mul.[i,ui,l,f,d], dvi.[i,ui,l,f,d], mod.[i,ui,l], eq.[i,l,f,d], ne.[i,l,f,d], ge.[i,ui,l,f,d], gt.[i,ui,l,f,d], le.[i,ui,l,f,d], lt.[i,ui,l,f,d], band.[i,l], bor.[i,l], bxor.[i,l], shl.[i,l], shr.[c,s,i,l], ushr.[c,s,i,l], and[i,l], or.[i,l].
POP_STRASGN	추출 후 삽입 또는 삽입	str.[c,s,i,u,l,p,f,d,t]
POP_STIASGN	추출 후 삽입 또는 두 번 추출 후 삽입	sti.[c,s,i,u,l,p,f,d,t]

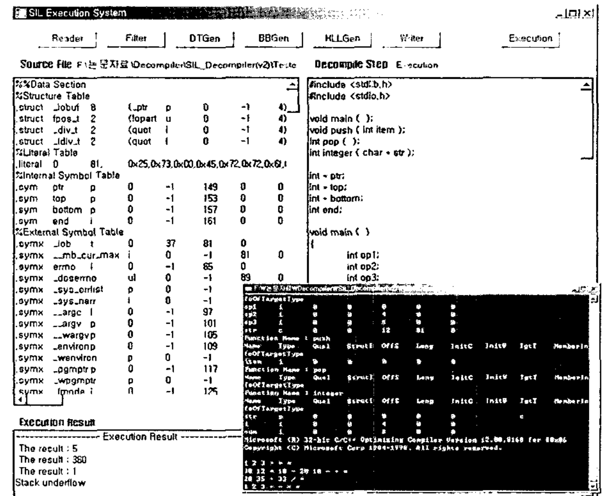
※ '[ ]'는 택일을 의미하며, c:char, uc:unsigned char, s:short, us:unsigned short, i:int, u:unsigned int, ui:unsigned int, l:long, ul:unsigned long, p:*, f:float, d:double, t:struct를 의미하는 기호이다. 그리고 "적용 명령어"항목의 PUSH와 PEEK는 패턴 처리된 명령어를 의미한다.

4.2 실행 시스템의 시험 결과

구현된 실행 시스템의 시험에 이용된 데이터는 <표 5>와 같다. 이 코드는 <표 4>의 코드를 EVM ANSIC 컴파일러에 의해서 컴파일한 SIL 코드로서 지면관계상 의사코드와 main함수의 연산코드만 나타내었다.

<표 6>은 SIL 코드의 역컴파일된 결과 코드를 나타낸 것이며, (그림 8)은 SIL 코드가 실행 시스템에 의해서 실행된 화면을 나타낸다.

(그림 8)의 왼쪽 상단 창은 리더에 의해서 로딩된 SIL 코드를 나타내며, 오른쪽은 역컴파일 단계별 진행 결과 코드를 출력한다. 그리고 아래 창은 SIL 코드의 실행 결과를 출력한다. 또한 실행 시스템의 서브 명령창은 데이터 테이블의 세부정보와 기본 블록이 처리되는 과정을 출력함으로써 SIL 코드의 실행 오류나 각 단계를 추적하고자 할 때 검토할 수 있도록 하였다.



(그림 8) SIL 코드 실행 결과

〈표 4〉 원본 C 코드

RPNCalculator.c

```
#include "C:\Program ...생략... \stdio.h"
#include "C:\Program ...생략...\stdlib.h"
#define MAX 100

int integer(char*);
void push(int);
int pop();

int *ptr, *top, *bottom;
int end;

void main(){
        int op1, op2, op3 = 1;
        char str[81];

        ptr = (int*)malloc(MAX * sizeof(int));
    top = ptr;
        bottom = ptr + MAX - 1;
    end = 1;

        do{
                scanf("%s", str);
                switch(*str){
                        case '+' :
                                op2 = pop();
                                op1 = pop();
                                push(op1 + op2);
                                break;
                        case '-' :
                                op2 = pop();
                                op1 = pop();
                                push(op1 - op2);
        break;
                        case '*' :
                                op2 = pop();
                                op1 = pop();
                                push(op1 * op2);
                                break;
                        case '/' :
                                op3 = pop();
                                op1=pop();
                                if(op3 == 0){
                                        printf("Error : Divide by 0\n");
                                        break;
                                }
                        push(op1 / op3);
                                break;
                        case '=' :
                                printf("\n The result : %d\n", pop());
                                break;
                        default : push(integer(str));
                }
        }while(op3 && end && *str != 'e' && *str != 'E');
}
void push(int item){
        if(ptr > bottom){
                printf("Stack overflow\n");
                end = 0;
        }
        else *ptr++ = item;
}
int pop(){
        ptr--;
        if(ptr < top){
                printf("Stack underflow\n");
                end = 0;
        }
        else return (int)*ptr;
}
int integer(char *str){
        int i, num = 0;
        for(i = 0; str[i] >= '0' && str[i] <= '9'; i++){
                num *= 10;
                num += str[i]-'0';
        }
    return num;
}
```

〈표 5〉 컴파일된 SIL 코드

RPNCalculator.c.sil

```
.literal  0        81,       0x25,0x73,............생략......,0x6e,0x00
.sym      ptr      p         0      -1      149     0      0
.sym      top      p         0      -1      153     0      0
.sym      bottom   p         0      -1      157     0      0
.sym      end      i         0      -1      161     0      0
.func     main     v         0      -1      0       0
.func     push     v         0      -1      1       0      (i,   0,   -1,  4)
.func     pop      i         0      -1      0       0
.func     integer  i         0      -1      1       0      (p,   0,   -1,  4)
main:  proc                  93     1       1
.sym      op1      i         0      0       587     0      0
.sym      op2      i         0      0       588     0      0
.sym      op3      i         0      0       589     0      0
.sym      str      c         0      0       590     81     0
          ldc.i    1
          str.i    1         8
          ldp
          ldc.u    400
          call     malloc
          str.p    0         68
          lod.p    0         68
          str.p    0         72
          lod.p    0         68
          ldc.i    400
          cvi.ui
          cvui.p
          add.p
          ldc.i    4
          cvi.ui
          cvui.p
          sub.p
          str.p    0         76
          ldc.i    1
          str.i    0         80
$$0:      nop
          ldp
          ldc.p    @0x0000
          lda      1         12
          call     scanf
          lda      1         12
          ldi.c
          ldc.i    43
          eq.i
          tjp      $$4
          lda      1         12
          ldi.c
          ldc.i    45
          eq.i
          tjp      $$5
          lda      1         12
          ldi.c
          ldc.i    42
          eq.i
          tjp      $$6
          lda      1         12
          ldi.c
          ldc.i    47
          eq.i
```

```
      tjp      $$7
      lda      1      12
      ldi.c
      ldc.i    61
      eq.i
      tjp      $$8
      ujp      $$3
$$4:  nop
      ldp
      call     pop
      str.i    1      4
      ldp
      call     pop
      str.i    1      0
      ldp
      lod.i    1      0
      lod.i    1      4
      add.i
      call     push
      ujp      $$2
$$5:  nop
      ldp
      call     pop
      str.i    1      4
      ldp
      call     pop
      str.i    1      0
      ldp
      lod.i    1      0
      lod.i    1      4
      sub.i
      call     push
      ujp      $$2
$$6:  nop
      ldp
      call     pop
      str.i    1      4
      ldp
      call     pop
      str.i    1      0
      ldp
      lod.i    1      0
      lod.i    1      4
      mul.i
      call     push
      ujp      $$2
$$7:  nop
      ldp
      call     pop
      str.i    1      8
      ldp
      call     pop
      str.i    1      0
      lod.i    1      8
      ldc.i    0
      eq.i
      fjp      $$9
      ldp
```

```
            ldc.p    @0x0003
            call     printf
            ujp      $$2
$$9:        nop
            ldp
            lod.i    1         0
            lod.i    1         8
            div.i
            call     push
            ujp      $$2
$$8:        nop
            ldp
            ldc.p    @0x0019
            ldp
            call     pop
            call     printf
            ujp      $$2
$$3:        nop
            ldp
            ldp
            lda      1         12
            call     integer
            call     push
$$2:        nop
            lod.i    1         8
            dup
            fjp      $$12
            pop
            lod.i    0         80
$$12:       nop
            dup
            fjp      $$11
            pop
            lda      1         12
            ldi.c
            cvc.i
            ldc.c    101
            cvc.i
            ne.i
$$11:       nop
            dup
            fjp      $$10
            pop
            lda      1         12
            ldi.c
            cvc.i
            ldc.c    69
            cvc.i
            ne.i
$$10:       nop
            fjp      $$1
            ujp      $$0
$$1:        nop
push:       proc     4         1      1
...생략...
pop:        proc     0         1      1
...생략...
integer:    proc     12        1      1
...생략...
```

5. 결론 및 향후 연구

본 논문에서는 가상기계 코드의 실험을 용이하게 하기 위해서, 역컴파일링 기법을 이용한 가상기계 코드 실행 시스템을 제안하였다. 역컴파일링 기법은 여러 가상기계 코드를 분석하여 도출하였으므로, 다른 목적을 위해 가상기계 코드를 역컴파일하는 경우, 중요한 참고 자료가 될 수 있을 것이다. 또한 제안한 시스템의 신뢰성을 위해서 EVM의 중간 언어인 SIL 코드를 기반으로 실험하였다. 구현한 실행 시스템은 SIL 코드의 검증 여부는 물론, 코드 변환 과정을 단계적으로 제시함으로써, SIL 코드의 완전성을 시험하기 위한 유용한 도구로 이용될 수 있다. 이러한 결과는 현재 구축 과정에 있는 EVM 시스템의 어셈블러와 가상기계를 보다 체계적으로 구현하도록 할 것이며, EVM 시스템의 신뢰성을 가증시킬 것이다.

향후 연구로는 제안된 시스템의 설계를 쉽게 하기 위해 가정한 제한을 제거하는 것이다. 이를 위해 역컴파일러에서 실행 포맷 파일을 직접 읽고, 복잡한 제어 흐름을 분석하도록 하는 시스템을 개선하는 일을 할 수 있을 것이다. 또한 현재 컴파일러의 제작에는 많은 자동화 방안이 연구되어 사용되고 있다. 이와 유사한 방법을 제안된 시스템의 주요 부분에 적용하여 역컴파일러 제작의 자동화 방안을 연구하는 것이다.

참 고 문 헌

[1] 안덕기, 오세만, "가상기계 코드 실행을 위한 역컴파일러", 정보처리학회 2006 춘계 학술 발표 논문집, 제13권, 제1호, pp.383-386, 2006.

[2] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron, "The case for virtual register machines," In Proceedings of ACM SIGPLAN, pp.41-49, 2003.

[3] Y. Shi, D. Gregg, A. Beatty, and M.A Ertl, "Virtual machine showdown: stack versus registers," In Proceedings of ACM SIGPLAN, pp.153-163, 2005.

[4] J. Gosling, "Java Intermediate Bytecodes," ACM SIGPLAN NOTICES, pp.111-118, 1995.

[5] MSIL Instruction Set Specification Version 1.9 Final, Microsoft Corporation, 2000.

[6] 남동근, 윤성림, 오세만, "가상기계를 위한 어셈블리 언어", 정보처리학회 2003 춘계 학술 발표 논문집, 제10권, 제1호, pp.783-786, 2003.

[7] D. Sugalsk, The Parrot Bytecode information, http://www.parrotcode.org/.

[8] 오세만, 이양선, 고광만, "임베디드 시스템을 위한 가상기계의 설계 및 구현", 멀티미디어학회 논문지, 제8권, 제9호, pp.1282-1291, 2005.

[9] M.H. Halstead, Machine-independent computer programming, Spartan Books, 1962.

〈표 6〉 역컴파일된 C 코드

DCPD_RPNCalculator.c

```
#include <stdlib.h>
#include <stdio.h>

void main (  );
void push ( int item );
int pop (  );
int integer ( char * str );

int * ptr;
int * top;
int * bottom;
int end;

void main (  )
{
                int op1;
                int op2;
                int op3;
                char str[81];

                op3 = 1;
                ptr = malloc(400);
                top = ptr;
                bottom = (ptr + 400 - 4/sizeof(ptr[0]));
                end = 1;
$$0:
                scanf("%s", &str[0]);
                if ( (*&str[0] == 43) )
                                goto $$4;
                if ( (*&str[0] == 45) )
                                goto $$5;
                if ( (*&str[0] == 42) )
                                goto $$6;
                if ( (*&str[0] == 47) )
                                goto $$7;
                if ( (*&str[0] == 61) )
                                goto $$8;
                goto $$3;
$$4:
                op2 = pop();
                op1 = pop();
                push((op1 + op2));
                goto $$2;
$$5:
                op2 = pop();
                op1 = pop();
                push((op1 - op2));
                goto $$2;
$$6:
                op2 = pop();
                op1 = pop();
                push((op1 * op2));
                goto $$2;
$$7:
                op3 = pop();
                op1 = pop();
                if ( !(op3 == 0) )
                                goto $$9;
                printf("Error : Divide by 0\n");
                goto $$2;
$$9:
                push((op1 / op3));
                goto $$2;
$$8:
                printf("\n The result : %d\n", pop());
                goto $$2;
$$3:
                push(integer(&str[0]));
$$2:
                if ( !op3 || !end || !((int)*&str[0] != (int)101) || !((int)*&str[0] != (int)69) )
                                goto $$1;
                goto $$0;
$$1:;
}
void push ( int item )
{
                if ( !((int)(unsigned int)ptr > (int)(unsigned int)bottom) )
                                goto $$13;
                printf("Stack overflow\n");
                end = 0;
                goto $$14;
$$13:
                *ptr++= item;
$$14:;
}
int pop (  )
{
                ptr--;
                if ( !((int)(unsigned int)ptr < (int)(unsigned int)top) )
                                goto $$15;
                printf("Stack underflow\n");
                end = 0;
                goto $$16;
$$16:
$$15:
                return *ptr;
}
int integer ( char * str )
{
                int i;
                int num;

                num = 0;
                i = 0;
$$17:
                if ( !((int)*&str[i] >= (int)48) || !((int)*&str[i] <= (int)57) )
                                goto $$18;
                num = (num * 10);
                num = (num + (int)(*&str[i] - 48));
$$19:
                i++;
                goto $$17;
$$18:
                return num;
}
```


[10] F.E. Allen and J. Cocke, "A program data flow analysis procedure," Communications of the ACM, Vol.19, No.3, pp.137-147, 1976.

[11] B.S. Baker, "An algorithm for structuring flowgraphs," Journal of the ACM, Vol.24, No.1, pp.98-120, 1977.

[12] C. Cifuentes, Reverse Compilation Techniques, PhD dissertation, Queensland University of Technology, School of Computing Science, 1994.

[13 The REC Decompiler, http://www.backerstreet.com/rec/rec.htm.

[14] ExeToC Decompiler, http://sourceforge.net/projects/exetoc.

[15] asm21toc translator, http://www.cs.rhul.ac.uk/research/languages/projects/reverse_compilation.html.

[16] relipmoC, https://sourceforge.net/projects/relipmoc.

안 덕 기

2003년 2월 가야대학교 경영정보학과(학사)

2006년 8월 동국대학교 대학원 컴퓨터공학과(공학석사)

현재 (주)다원미디어 연구원

관심분야 : 프로그래밍 언어, 가상기계, 임베디드 시스템 등

이 창 환

1998년 2월 동국대학교 컴퓨터공학과(학사)

2000년 2월 동국대학교 대학원 컴퓨터공학과(공학석사)

2003년 2월 ~ 현재 동국대학교 대학원 컴퓨터공학과(공학박사)

2006년 9월 ~ 현재 ㈜ 링크젠 책임연구원

2007년 3월 ~ 현재 동국대학교 산업기술연구원 겸임교수

관심분야 : 프로그래밍 언어, 컴파일러, 임베디드 시스템

오 세 만

1985년 3월 ~ 현재 동국대학교 컴퓨터공학과 교수

1993년 3월 ~ 1999년 2월 동국대학교 컴퓨터 공학과 대학원 학과장

2001년 11월 ~ 2003년 11월 한국정보과학회 프로그래밍언어연구회 위원장

2004년 6월 ~ 2005년 12월 한국정보처리학회 게임연구회 위원장

관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅