Journal of Korea Multimedia Society Vol. 9, No. 9, September 2006(pp. 1247-1256)

가상 기계 코드를 위한 패턴 매칭 최적화기

이창환[†], 오세만[††]

요 약

가상 기계란 하드웨어로 이루어진 물리적 시스템과는 달리 소프트웨어로 제작되어 논리적인 시스템 구성을 갖는 개념적인 컴퓨터이다. 그러나 가상 기계는 실제 프로세서로 처리하는 것보다 실행 속도가 매우 느리기 때문에 실행되는 코드의 최적화가 매우 중요하다. 본 논문은 가상 기계 코드 최적화기의 실험대상으로 EVM(Embedded Virtual Machine)의 중간 코드인 SIL(Standard Intermediate Language)을 이용하였다. 현존하는 최적화 방법론에 관한 연구를 통하여 가상 기계 코드 특성을 고려한 최적화 방법론을 제시하고, 최적화된 코드를 생성하기 위한 코드 최적화기를 설계하고 구현하였다. 가상 기계 코드 최적화기는 주어진 패턴을 찾아서 패턴에 해당하는 부분을 최적화 코드로 바꾸어, 전체 코드의 크기를 줄이고 실행 속도의 개선효과를 가진다. 또한, 구현된 최적화기의 실험 결과를 도출하였다.

Pattern Matching Optimizer for Virtual Machine Codes

Changhwan Yi[†], Seman Oh[††]

ABSTRACT

VM(Virtual Machine) can be considered as a software processor which interprets the abstract machine code. Also, it is considered as a conceptional computer that consists of logical system configuration. But, the execution speed of VM system is much slower than that of a real processor system. So, it is very important to optimize the code for virtual machine to enhance the execution time. In this paper, we designed and implemented the optimizer for the virtual(or abstract) machine code(VMC) which is actually SIL(Standard Intermediate Language) that is an intermediate code of EVM(Embedded Virtual Machine). The optimizer uses the pattern matching optimization techniques reflecting the characteristics of the VMC as well as adopting the existing optimization methodology. Also, we tried a benchmark test for the VMC optimizer and obtained reasonable results.

Key words: Optimizer(최적화기), Pattern Matching(패턴 매칭), Virtual Machine Codes(가상 기계 코드)

1. 서 론

가상 기계는 하드웨어가 아닌 소프트웨어로 제작되어, 물리적인 시스템이 아닌 논리적인 시스템 구성을 갖는 개념적인 프로세서이다. 따라서 가상 기계는 실행 환경인 하드웨어와 운영체제에 종속적이지 않기 때문에 플랫폼 독립적 실행을 가능하게 한다. 대표적인 가상 기계로는 JVM(Java Virtual Machine), GVM(General Virtual Machine), KVM 등이 있다. 이러한 가상 기계가 최근에는 모바일 디바이스뿐만 아니라 디지털 TV, 셋탑 박스 등의 임베디드 시스템으로 확산되고 있다[12].

임베디드 시스템이란, 전용 동작을 수행하거나 또는 특정 임베디드 소프트웨어 응용 프로그램과 함께

※ 교신저자(Corresponding Author) : 오세만, 주소 : 서울시 중구 필동 동국대학교 원흥관 컴퓨터공학과 프로그래밍 언어 연구실(100-715), 전화 : 02)2260-3342, FAX : 02)2265-8742, E-mail : smoh@dongguk.edu

접수일 : 2006년 6월 23일, 완료일 : 2006년 7월 14일

[†] 링크젠
(E-mail : yich@dongguk.edu)
[††] 정회원, 동국대학교 컴퓨터공학과

사용되도록 디자인된 특정 컴퓨터 시스템 및 컴퓨팅 장치를 말한다. 임베디드 시스템을 위한 가상 기계 기술은 모바일 디바이스와 디지털 TV등에 탑재할 수 있는 핵심 기술로 다운로드 솔루션에서는 꼭 필요한 소프트웨어 기술이다.

EVM은 모바일 디바이스, 셋탑 박스, 디지털 TV 등에 탑재되어 동적인 응용 프로그램을 다운로드하여 실행할 수 있는 가상 기계 솔루션이다. 이러한 EVM은 가상 기계 코드로 SIL 코드를 사용한다.

SIL 코드는 임베디드 시스템을 위한 가상 기계의 표준 중간 언어로 설계되었다. SIL은 다양한 프로그래밍 언어를 수용하기 위해서 기존의 가상 기계 중간 언어들의 분석을 토대로 정의하였으며, 객체지향 언어와 순차적인 언어를 모두 수용할 수 있는 연산 코드의 집합을 가지고 있다. 그러나 SIL은 객체 지향적으로 설계되었기 때문에 연산 코드에 타입정보가 없었으나, 실질적으로 효율적인 실행을 위하여 연산 코드에 타입정보를 추가하여 새로이 확장된 SIL (Extended SIL)로 재정의 하였다. 본 논문은 확장된 SIL을 대상으로 하여 가상 기계 코드 최적화기를 설계하고 구현하였다.

본 논문에서는 가상 기계 코드 중 SIL 코드를 대상으로 최적화기를 설계하고 구현하였다. 최적화기를 설계할 때 가상 기계 코드 및 가상 기계의 특성을 고려한 패턴을 정의하였다. 가상 기계 코드 최적화기는 가상 기계 코드를 정해진 자료구조에 저장한 후 자료구조를 탐색하면서 패턴에 해당하는 부분을 최적화 코드로 바꾼다. 또한, 이렇게 바뀐 최적화 코드가 올바르게 바뀌었는지에 대한 검증이 필요하므로, 검증을 위한 방법론을 제시하고, 검증을 위한 역컴파일러를 제작하여 최적화된 코드를 검증하였다.

2. 배경 연구

2.1 가상 기계

가상 기계는 하드웨어가 아닌 소프트웨어로 제작된 기계로써, 물리적인 시스템이 아닌 논리적인 시스템 구성을 갖는 개념적인 프로세서이다. 따라서 가상 기계는 실행 환경인 하드웨어와 운영체제에 종속적이지 않기 때문에 플랫폼 독립적 실행을 가능하게 한다.

이런 가상 기계를 사용함으로써 얻을 수 있는 장점은 응용 프로그램의 실행 환경인 프로세서나 운영체제가 바뀌더라도 응용 프로그램을 수정하지 않고 사용할 수 있다는 점이다. 즉, 응용 프로그램이 가상 기계의 요구 사항을 맞추면, 응용 프로그램을 손보지 않아도 타 기종으로 이식할 수 있다는 것이다.

하지만 가상 기계의 가장 큰 단점은 속도의 저하이다. CPU에 독립적으로 실행을 하기 위해 CPU 내에 있는 레지스터의 사용을 피하고 특정 메모리를 레지스터처럼 사용하여 CPU를 에뮬레이션하기 때문에 불필요한 CPU 클럭을 소모한다. 이러한 단점 때문에 가상 기계가 잘 쓰이지 않았으나, 최근 기술의 발달로 인한 하드웨어의 성능이 좋아짐에 따라, 실행속도가 사람이 감인할 수 정도의 속도를 보장한다.

이런 제약점을 가지기 때문에 가상 기계에서 실행되는 응용 프로그램은 경량화와 고속화라는 두 가지 조건을 만족시켜야 하는 문제점을 가지게 된다. 본 논문에서는 이런 가상 기계에서 실행될 중간 코드의 경량화와 고속화 요건을 충족할 수 있는 최적화기를 설계하고 구현하였다.

2.2 최적화 방법론

코드 최적화란 주어진 입력 프로그램과 의미적으로 동등하면서 실행면에서 효율적인 코드로 바꾸는 것을 의미한다. 따라서 코드 최적화기는 가급적 계산의 횟수를 줄이고, 보다 빠른 명령을 사용하여 실행시간이 짧은 코드를 생성해야 하며, 기억 장소의 요구량을 최소화해야 한다.

코드 최적화 과정은 최적화 되어지는 관점에 따라 여러 가지로 분류된다. 먼저, 프로그램의 부분적인 관점에서 일련의 비효율적인 코드를 구분해내어 좀 더 효율적인 코드로 만드는 핍홀(Peephole) 혹은 국부 최적화와, 전체적인 관점에서 일련의 비효율적인 코드들을 구분해내어 좀 더 효율적인 코드로 만드는 전역 최적화로 나눌 수도 있다. 또한, 전단부에서 생성된 어셈블리에 대해 수행되는 기계 독립적인 최적화와 목적 기계의 특성에 따라 목적 코드에 대해 수행되는 기계 종속 최적화로 구분되어 질 수 있다.

원시 프로그램에 대한 컴파일과정 중 최적화 단계에서는 프로그램의 실행 속도를 개선시키고 코드 크기를 줄일 수 있는 다양한 최적화 방법을 수행한다. 일반적으로 최적화는 컴파일 과정에서 선택적인 단

계로 필요시에만 컴파일러 옵션으로 선택하여 실행할 수 있다. 최적화를 수행하는 주요한 기준은 첫째, 반드시 입력과 의미적으로 동등해야 한다. 둘째, 평균적으로 속도가 빨라져야 한다. 셋째, 최적화에 들인 노력에 대한 효과가 있어야 한다.

최적화기를 구현하는 기술인 패턴 매칭 방법은 이식성을 위한 코드 생성 연구로서 코드 생성 알고리즘 자체로부터 기계 표현을 분리하는데 중점을 두었다. 이 방법의 장점은 모든 기종에 대하여 하나의 코드 생성 알고리즘을 사용할 수 있는 잠재적인 능력이 있다는 것이다.

패턴 매칭 코드 생성 방법은 Weingart에 의해 처음으로 소개되었는데 목적 기계 명령어 집합을 패턴 매칭 과정을 쉽게 하기 위해 하나의 패턴 트리로 부호화하며 코드 생성기는 파서로 패턴을 받고 동등한 패턴을 찾기 위해 패턴 트리를 검색하는 일종의 트리 운행기이다. 만약 동등한 것이 존재한다면 대응하는 코드가 발생되고 더 이상 다른 변형이 수행되지 않기 때문에 컴파일되는 언어는 트리의 기계 표현과 아주 밀접하게 닮아야 한다.

이 방법의 중요한 장점은 어떤 코드를 생성할지를 결정할 때 목적 컴퓨터의 명령어 집합에 의존하기 때문에 모든 가능한 명령어들은 코드 생성기에 이용 가능하고 명령어 집합을 조심스럽게 순서화한다면 표준화하지 않은 명령어들에 대한 특별한 경우도 코드 생성기에서 자동적으로 발견 가능하다는 것과 새로운 기종에 대한 컴파일러를 재목적화하기 위해서는 단지 트리 구성기에 입력하여 새로운 명령어 집합으로 대치만 해주면 새로운 패턴 트리를 만들어 동일한 코드 생성기를 사용할 수 있다는 점이다[11].

2.3 표준 중간 언어(SIL: Standard Intermediate Language)

SIL은 임베디드 시스템을 위한 가상 기계의 표준 중간 언어(SIL : Standard Intermediate Language)로 설계되었다. 따라서 다양한 프로그래밍 언어를 수용하기 위해서 JAVA bytecode, .NET IL등 기존에 널리 사용되고 있는 가상 기계 중간 언어들의 분석을 토대로 정의되어 있다.

SIL은 클래스 등 특정 작업의 수행을 의미하는 의사코드와 가상 기계에서 실행되는 실제 명령어에 대응되는 연산 코드로 이루어져 있다. 연산코드는 스택 기반의 명령어 집합이며, 특정 프로그래밍 언어에 종속되지 않는 언어 독립성과, 하드웨어 및 플랫폼 독립적인 하드웨어 독립성을 가지고 있다. 또한 연산코드의 니모닉(Mnemonic)은 특정 하드웨어나 소스 언어에 종속되지 않는 추상적인 형태를 지닌다.

SIL은 객체지향 언어와 순차적 언어를 모두 수용하기 위한 연산 코드 집합으로 이루어져 있으며 중간 언어 수준의 디버깅을 용이하게 하기 위해 일관성 있는 이름 규칙을 적용하여 코드의 가독성을 확보하고 있다.

SIL 코드는 6종류로 분류할 수 있으며, 총 397개의 코드를 가지고 있다. 이중 최적화 관련 코드는 203개가 있으며, 이 중 가상 기계 코드 최적화기에서 사용하는 코드는 187개이다. 가상 기계 코드 최적화기에서 적용되지 않은 16개의 코드는 가상 기계내부에서 적용하는 코드이다. 다음의 [표 1]과 [표 2]는 SIL 코드의 분류와 최적화가 적용되는 코드에 대한 분류를 나타낸 것이다.

SIL은 객체 지향적으로 설계되어 타입정보를 가지지 않았으나, 실질적으로 효율적인 실행을 위하여 연산 코드에 타입을 추가하여 확장된 SIL로 재정의하였다. 본 논문에서는 확장된 SIL을 가지고 가상 기계 코드 최적화기를 구현하였다[12].

3. 가상 기계 코드 최적화 시스템 설계

본 논문에서 제안하는 가상 기계 코드 최적화기는 가상 기계 코드를 입력으로 받아 이를 이중 구조 연결 리스트로 된 자료구조에 코드의 모든 정보를 저장한다. 최적화기는 정보가 저장된 자료 구조를 탐색하면서 최적화 가능 패턴을 발견하면, 최적화 코드에 대한 자료구조를 만든 후 최적화 패턴에 해당하는 부분과 대치한 후 기존의 자료구조는 버린다. 위와 같은 과정을 자료구조의 끝까지 진행한 후 원래의 SAF(Standard Assembly Format) 형태로 코드를 복원하면 최적화된 가상 기계 코드가 나오도록 구성하였다.

3.1 시스템 구성도

가상 기계 코드 최적화기는 ANSI C 컴파일러의 결과물로 나온 SAF을 입력으로 받아, 동등한 의미를 가지면서 원본 코드보다 '저용량'과 '고속성'을 가지

표 1. SIL 코드의 분류

카테고리	설 명
Stack operations	스택 제어 연산 코드와 저장 장소에 따른 연산 코드 집합
Arithmetic operations	수학적 연산과, 비교 연산, 비트 연사, 그리고 논리 연산 코드 집합
Flow control operations	분기와 메소드 호출등 프로그램의 흐름 제어와 관련된 연산 코드 집합
Type conversion operations	데이터의 타입을 특정 타입으로 변환하는 연산 코드 집합
Optimize operations	최적화를 위한 코드 집합
Object operations	객체 생성과 필드 접근에 관련된 연산 코드 집합

표 2. 최적화 적용 코드의 분류

카테고리	설 명
Arithmetic Code	수학적 연산 부분에 대한 최적화 코드. (53개)
Parameter Optimize Code	코드의 파라미터를 줄일 수 있는 최적화 코드. (12개)
Inc-Dec Optimize Code	변수의 값을 일정 범위안에서 증감하는 부분에 대한 최적화 코드. (12개)
Array Optimize Code	배열 연산 부분에 대한 최적화 코드. (36개)
Jump Optimize Code	점프 연산 부분에 대한 최적화 코드. (28개)
Assign Optimize Code	배정 연산 부분에 대한 최적화 코드. (36개)

는 최적화된 SIL 코드로 변환한다. [그림 1]은 이러한 과정을 도식화한 것이다.

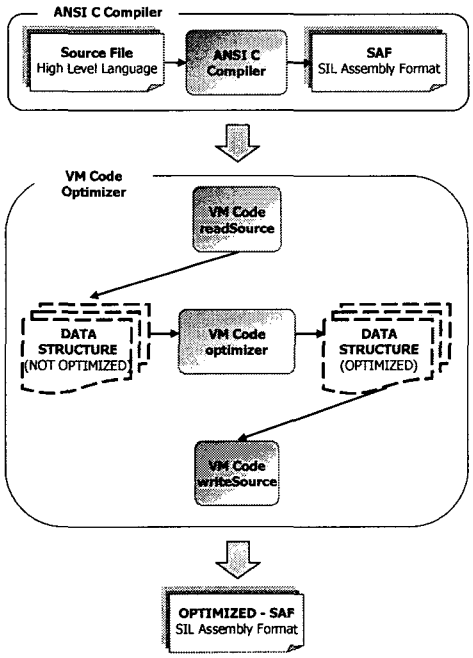

그림 1. 코드 최적화기 구성도

3.2 자료구조

본 논문에서 실험대상으로 하고 있는 SIL 코드는 크게 데이터 부분과 코드부분으로 나누어 볼 수 있다. 데이터 부분은 함수의 선언등과 같은 선언부에 대한 정보를, 코드 부분은 함수 안의 문장부 및 함수 안에서 선언된 정보를 가지고 있다. 본 논문에서 제안하는 자료구조는 가상 기계 코드 한 라인의 정보를 모두 저장하고, 저장된 정보들을 순서 있게 저장하기 위하여 이중 연결 리스트 구조를 사용하였다. [그림 2]는 정해진 자료구조를 도식화 한 것이다.

자료구조에서는 stringInfo 항목은 컴파일러에 의해 생성된 가상 기계 코드의 의미 없는 내부 정보를 저장하는데 사용된다. 의미 없는 내부 정보들에는 C 소스의 라인 정보, 변수의 선언정보와 SAF에 필요한 정보 등을 말한다. C소스의 라인 정보를 제외하면 실제 가상 머신에서 돌아가는데 필요한 정보이지만, 최적화기 내에서는 불필요한 부분이므로 stringInfo에 저장하여 일반 문자열로 처리하였다.

3.3 최적화 알고리즘

가상 기계 코드 최적화기의 목적은 ANSI C 컴파

```
typedef struct silNode {
    char label[5]; // label 정보
    char instruction[10]; // instruction 정보
    char type[5]; // instruction type 정보
    int base1, offset1, base2, offset2, base3, offset3,base4, offset4;
    //주소 정보를 기억하기 위한 공간
    //(2 ~ 4번은 최적화를 위한 부분)
    int value; // code value에 대한 정보
    char jumpLabel[10]; // 점프 label
    char *stringInfo; // SIL 코드내의 디버깅 정보
    int codeShape; // SIL code의 형태정보를 가리킴
    // 출력형태를 결정하기 위해 사용되는 정보
    struct silNode *pre; // 전 노드를 가리킴
    struct silNode *next; // 다음 노드를 가리킴
} SILCodeInfomation;
```

그림 2. 최적화기의 자료구조

일러가 생성한 코드들을 조작하여, 동등한 의미를 가지면서 작은 크기 및 실행 효율이 높은 코드들로 바꾸는데 있다. 가상 기계 코드 최적화에서 사용되는 최적화 알고리즘은 패턴 매칭 방법이다. 주어진 자료구조에 저장된 정보를 탐색하고, 정해진 패턴을 발견하면 패턴부분을 최적화 코드로 바꾸어 자료구조에 저장한 후, 패턴부분을 삭제하고 새로 정의된 자료구조를 연결하는 방식으로 진행된다.

본 논문에서 일반적으로 사용되는 최적화 방법에는 도달할 수 없는 코드의 제거, 수행에 불필요한 코드의 제거, 중복 된 코드의 제거, 상수들로 구성된 연산의 제거, 제어 관련 최적화 등이 있다. 그리고 가상 기계 코드 의존적인 최적화 방법으로는 스택 관련 최적화, 배정 관련 최적화, 배열관련 최적화 등을 수행하였다. 스택 관련 최적화는 스택 기반 언어인 가상 기계 코드가 수행한 스택에 대한 연산에 관련된 최적화를 뜻한다.

3.4 최적화 패턴

패턴 매칭으로 수행되는 최적화기의 성능은 패턴 내용에 크게 좌우되기 때문에 패턴 작성은 매우 중요하며, 최대한의 최적화 효과를 반영할 수 있도록 구성되어야 한다. 최적화 패턴은 입력된 가상 기계 코드 명령어 집합을 최적화된 가상 기계 코드로 교체하기 위한 수단으로써 사용된다.

최적화 패턴은 가상 기계 코드의 특성을 고려하여 작성된 패턴 표현을 통해 패턴 테이블을 생성하는 방법으로 찾는다. 패턴 생성기는 이 패턴 테이블을 입력으로 받아 가상 기계 코드 최적화기에 들어가는 패턴을 출력한다[11].

패턴에 반영된 최적화 내용은 도달할 수 없는 코드의 제거, 스택 관련 연산, 산술/논리 연산, 배정 연산, 제어 연산 등을 포함하는 패턴으로 구성되어 있다. [그림 3]은 최적화에 사용될 가상 기계 코드 패턴의 형식을 보여주고 있다.

4. 가상 기계 코드 최적화 시스템 구현

컴파일러에 의해 나온 SIL 코드의 어셈블리 형태인 SAF를 가지고 최적화기를 구현하였다. 본 절에서는 입력으로 받은 SAF 형태의 SIL 코드 중에서 최적화 패턴을 찾고, 찾은 패턴을 최적화된 코드로 바꾸는 가상 기계 코드 최적화기의 구현에 대해 설명하고 있다. 본 논문에서 설계하고 구현한 최적화기는 ANSI C로 구현하였고, C 컴파일러로는 Microsoft사의 Visual C++ 6.0을 사용하였다.

4.1 최적화 과정

입력부분은 최적화기의 입력으로 들어온 SAF

```
// 배열
 [lda]c1 [ldc.i] c2 [ldc.i]c3 [mul.i] [cvi.ui] [cvui.p] [add.p] [ldi.i]
 => [ldele.i] c1 c2
.....

// 파라미터 최적화
 [ldc.i]c1
 => [ldc.i.1] (단, c1의 값이 1인 경우)
.....

// 증감 최적화
[lod.i]c1 [ldc.i]c2 [add.i] [str.i]c1
 => [incm.i]c1 (단, c2의 값이 1인 경우)
.....
```

그림 3. SIL 코드에 대한 패턴

를 정해진 자료구조에 저장하고, 저장된 자료구조를 순서 있게 나열하는 역할을 한다. 입력부분에서는 SAF를 1라인씩 읽으면서 1라인의 정보를 분석하여 3.2절에서 정의한 자료구조에 추출한 정보를 저장한다. 이때 저장하는 주요 내용은 인스트럭션 정보, base 정보, offset 정보, 레이블 정보 및 타입 정보 등이다. 1라인씩 저장된 정보는 링커에 의해서 순서 있게 나열되어 다음 최적화 부분에서 처리된다.

최적화 부분은 입력부분에서 받은 자료구조를 탐색하여 최적화 패턴을 찾고, 패턴에 해당하는 부분의 정보를 모아 자료구조에 저장하고 이 부분을 기존의 최적화 패턴과 바꾼다. 만약 최적화 패턴을 찾지 못하면, 다음 자료구조로 이동하여 최적화 패턴을 검색한다. 이러한 패턴 탐색 및 변환동작을 코드의 끝까지 반복한다.

출력 부분은 최적화 부분으로 넘겨받은 자료구조를 원래의 SAF 형태로 복원하는 역할을 한다. 이때 출력을 위한 부분인 codeShape 정보를 받아 그 형태에 알맞은 구조로 자료구조의 정보를 변환한다.

4.2 최적화 검증

본 절에서는 위에서 구현한 최적화기의 결과를 검증하는 최적화기 검증 방법 및 디컴파일러를 제시하고 있고, 검증 실험결과를 보여주고 있다.

4.2.1 최적화 검증 방법

최적화기의 결과를 검증하는 방법은 크게 2가지로 나눌 수 있다. 첫 번째는 SIL 코드를 실행하는 가상 기계의 동작을 보고 검증하는 방법을 생각할 수 있고, 두 번째는 최적화기 결과를 다시 원 소스로 돌려 최적화 결과를 검증하는 방법이 있다. 첫 번째 방법의 경우는 가상 기계의 제작이 필요하고, 두 번째 방법은 가상 기계 코드를 다시 원 소스로 변환하는 디컴파일러가 필요하다.

본 논문은 두 번째 방법을 이용하여 최적화 코드를 검증하였다. 최적화 전의 SIL 코드와 최적화 후의 SIL 코드를 디컴파일하여 나온 결과의 비교를 가지고 최적화에 대한 검증을 하였다.

4.2.2 최적화 검증 시스템 구성

최적화 검증을 위한 디컴파일러도 하나의 컴파일러이기 때문에 컴파일러 제작과 동일한 과정을 거쳐서 구현하였다. 디컴파일러를 위한 문법을 설계하고, 이를 토대로 파서, 스캐너, SDT 및 Decode의 모듈을 제작하여 하나의 컴파일러를 구현하였다. 일반적인 컴파일러의 구성 요소 중 CG(Code Generator) 대신 Decoder가 들어가 있어 SIL 정보를 C소스로 다시 q복원한다.

본 논문에서 사용한 최적화 검증 시스템의 구조는

[그림 4]와 같다. SIL 코드와 최적화기를 거친 SIL 코드를 가지고 모두 디컴파일하여 나온 소스 파일(*.c)을 다시 컴파일하여 나온 결과의 실행 값을 비교하여, 최적화의 결과가 맞는지를 검증하였다. 2개의 실행 결과가 같으면 최적화된 코드가 올바르게 된 것이다. 5.2절에서 최적화 검증에 대한 실험 내용을 언급한다.

5. 실험 결과

본 논문에서 제안한 최적화기의 실험의 실행 시간 측정에 많은 영향을 주는 환경 요소는 CPU와 메모리, 운영체제이다. 본 논문에서는 CPU는 AMD사의 애슬론 2500+, 메모리는 1GB, 운영체제는 Microsoft사의 Windows XP Pro SP2를 사용한 환경에서 실험을 하였다.

5.1 최적화 결과

가상 기계 최적화기를 통해 나온 결과는 아래 그림과 같다. 다음은 자연수 의 약수의 합이 자기 자신이 되는 완전수를 구하는 프로그램을 컴파일한 SIL 코드(최적화 전의 SIL 코드)와 SIL 코드를 최적화한 코드(최적화 후의 SIL 코드)이다. [그림 5]는 최적화 전의 SIL 코드이고, [그림 6]은 최적화 후의 SIL 코드이다.

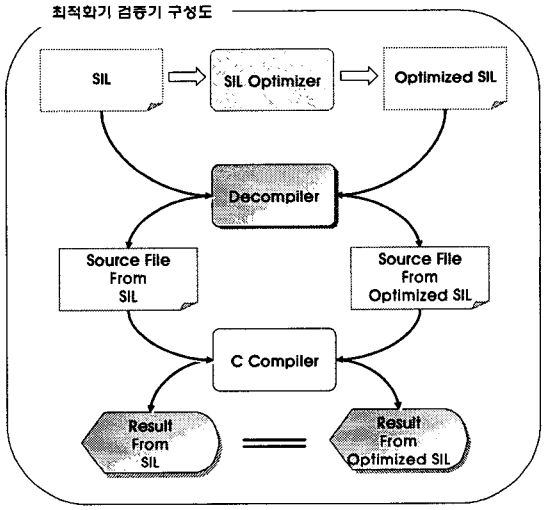

그림 4. 최적화 검증기 구성도

```
%Line 15 : for(i = 500; ; i++) {
        ldc.i       500
        str.i       1           0
$$0:    nop
        ldc.i       1
        fjp         $$1
%Line 16 : s = 0;
        ldc.i       0
        str.i       1           8
%Line 17 : for (j = 1; j <= i/2; j++)
        ldc.i       1
        str.i       1           4
$$3:    nop
        lod.i       1           4
        lod.i       1           0
        ldc.i       2
        div.i
        le.i
        fjp         $$4
%Line 18 : if (i % j == 0) s += j;
        lod.i       1           0
        lod.i       1           4
```

그림 5. 최적화 전의 SIL 코드

```
%Line 15 : for(i = 500; ; i++) {
     setvc.i  1        0        500
$$0: nop
     ldc.i.1
%Line 16 : s = 0;
     setvc.i  1        8        0
%Line 17 : for (j = 1; j <= i/2; j++)
     ldc.i.1
$$3: nop
     lod.i    1        4
     lod.i    1        0
     ldc.i.2
     le.i
     fjp      $$4
%Line 18 : if (i % j == 0) s += j;
     modv.i   1        0        1        4
     ldc.i    0
     eq.i
     fjp      $$6
     addv.i   1        8        1        4
     str.i    1        8
```

그림 6. 최적화 후의 SIL 코드

5.2 최적화 검증

5.1절에서 최적화 코드가 정상적으로 나오는 것을 알 수 있다. 이런 최적화기의 결과는 원 코드의 실행 결과와 항상 같아야 한다. 최적화 결과 검증을 위해, 본 논문에서는 최적화된 코드를 디컴파일러를 통해 C 코드로 복원한 후, 이미 확인된 다른 C 컴파일러를 통해 같은 결과가 나오는지 확인하는 방식으로 검증하였다. 최적화 검증은 [표 3]과 [표 4]의 성능 평가를 위해 사용한 프로그램으로 검증하였다.

5.3 최적화기의 성능 평가

본 논문에서 구현한 가상 기계 코드 최적화기의 성능을 평가하기 위한 실험으로 가상 기계 코드의 크기 및 가상 기계 코드 라인 수를 비교 측정하였다 가상 기계는 스택 기반 머신이기 때문에 단순한 코드 여러 개보다 복잡한 코드 한 개를 실행하는 게 더

표 3. 최적화 전후의 크기 및 라인 수 비교

*.saf	Code Size (byte)				Line Count				코드사이즈 최적화율 (Size)
	최적화 전		최적화 후		최적화 전		최적화 후		
	고정	적용	고정	적용	고정	적용	고정	적용	
Hanoi.saf	1965	1918	1965	1366	48	195	48	130	29%
Heap.saf	2266	2679	2266	1572	64	274	64	133	42%
Magic.saf	2363	2071	2363	1770	49	192	49	151	15%
Merge.saf	3176	3311	3176	2126	91	330	91	176	36%
Pal.saf	1394	614	1394	543	32	53	32	42	12%
Perfect.saf	947	567	947	461	25	53	25	39	19%
Prime.saf	921	507	921	435	26	49	26	40	15%
Qsort.saf	1891	1540	1891	1065	52	153	52	91	31%
결과분석	13207		9338		1299		802		30%

표 4. 최적화 전후의 실행 시간 비교

*.saf	최적화 전 (msec.)	최적화 후 (msec.)	최적화 율 (Second)
Hanoi.saf	78.2	72.9	7%
Heap.saf	87.0	79.2	9%
Magic.saf	128.0	123.5	4%
Merge.saf	89.4	82.0	9%
Pal.saf	41.8	40.3	4%
Perfect.saf	34.2	32.4	6%
Prime.saf	27.7	26.0	7%
Qsort.saf	97.3	92.1	6%
결과분석	583.6	548.4	7%

빠르다. 따라서 최적화 코드의 라인 수도 최적화 율을 측정하는 중요한 기준이 된다. 또한 가상 기계 코드의 실행속도를 측정하기 위하여 인터프리터를 구현하고, 인터프리터를 통한 실행 시간을 가지고 최적화 전과 최적화 후의 실행속도를 비교하였다.

다음 [표 3]은 최적화 전의 SIL 코드와 최적화 후의 SIL 코드의 크기 및 라인 수를 비교한 것이다. 고정 부분은 최적화에 영향을 끼치지 않는 코드 정보 등을 나타내는 것이고, 적용 부분은 최적화의 대상이 되는 SIL 코드의 집합을 가리킨다.

다음 [표 4]는 [표 3]의 프로그램의 실행 시간을 정리한 것이다. 실행 시간을 구하는 방법은 2가지가 있다. 첫 번째 방법은 VM을 구현하여 직접 기계에서 실행하는 방법이 있고, 두 번째 방법은 인터프리터를 구현하여 정확하지는 않지만 근사치에 가까운 값을 가상적으로 구하는 방법이다. 본 논문에서는 인터프리터를 직접 구현하여 실행하였고, 각각의 프로그램을 1만 번 반복 실행한 후, 나온 결과를 가지고 실행 시간을 구하였다.

실험 결과를 분석하면 실험에 사용된 SIL 코드들은 크기 및 라인 수에서 20~30%를, 시간적인 면에선 평균 7%의 최적화율을 보여주었다. 특히, 배열연산이 많은 소스 프로그램일수록 다른 프로그램에 비해서 좋은 최적화율을 보여주고 있다.

6. 결론 및 향후연구

가상 기계는 하드웨어가 아닌 소프트웨어로 제작된 논리적인 시스템 구성을 갖는 개념적인 프로세서이다. 가상 기계는 운영체제와 플랫폼에 독립적인 실행이 가능하여, 응용 프로그램을 이식하는데 실제 기계에 비해 이점이 많다. 그러나 논리적으로 구성된 프로세서다 보니 실행 속도가 매우 느렸으나, 하드웨어가 발달하면서 프로세서의 속도가 빨라져 사용자가 감내할 수 있는 수준이 되었다. 가상 기계의 장점이 부각되어 현재 모바일, 디지털 TV와 셋탑 박스 등의 임베디드 머신에 널리 사용되고 있다. 가상 기계가 필요로 하는 '경량화'와 '고속화'를 충족시키려면, 가상 기계에서 수행되는 프로그램의 최적화가 매우 중요하다.

본 논문에서는 기존의 최적화 방법들의 분석을 기반으로 하여, 가상 기계의 특성을 고려한 최적화기를 구현하였다. 가상 기계 코드 최적화기는 가상 기계 코드에 대한 각 명령어의 특성 및 가상 기계에서 실행되는 동작을 분석하고, 가상 기계 코드에 대한 최적화 방법들을 이용하여 공통식의 제거, 연산 강도 경감, 루프 불변 코드 이동등과 같은 최적화를 진행하였다. 특히, 가상 기계 코드에 의존적 최적화 방법인 배정, 스택, 배열에 관련된 최적화를 수행하였다.

가상 기계는 하나의 명령어 실행시간이, 명령의 복잡도보다는 명령어의 개수에 더 많은 영향을 받는다. 즉, 명령어의 개수를 많이 줄일수록 실행속도는 그 만큼 좋아진다. 본 논문에서 설계하고 구현한 가상 기계 코드 최적화기는 여러 기능을 수행하는 코드의 새로운 정의가 가능하여, 적용될 패턴의 개수는 더 늘어날 수 있다. 또한 컴파일러의 출력 코드의 분석을 통하여 새로운 패턴을 찾을 수도 있다.

향후 연구 방향은 가상 기계 코드 최적화기에 사용되는 코드 최적화 알고리즘을 보다 효율적으로 보완하여, 양질의 최적화된 코드를 생성하는데 있다. 또한 기존의 패턴을 분석하여 좀 더 다양한 패턴을 찾아야 한다. 최적화를 위한 가상 기계 코드를 확장해서 코드 최적화를 극대화하여야 할 것이다.

참 고 문 헌

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compiler Principles, Techniques, and Tools*, Addison-Wesley, 1985.

[2] Andrew S. Tanenbaum, Hans van Staveran, and John W. Stevenson, "Using Peephole

Optimization on Intermediate Code," *ACMTrans. on Prog. and System*, Vol. 4, No. 1, pp. 21-36, 1982.

[3] B. J. Mckenzie, "Fast Peephole Optimization Techniques," *Software Practice and Experience*, Vol. 19, No. 2, pp. 1151-1162, 1989.

[4] Christopher Earnest, "Some Topics in Code Optimization," *Journal of the Association for Computing Machinery*, Vol. 21, No. 1, pp. 76-102, 1974.

[5] Has Van Staveren, *Internal Documentation on the peephole optimizer from the Amsterdam Compile Kit*, Dept. of Mathmatics and Computer Science Vrje Universiteit Amsterdam, The Netherlands, 1991.

[6] Jack W. Davidson and David B. Whalley, "Quick Compilers Using Peephole Optimization," *Software Practice and Experience*, Vol. 19, No. 1, pp. 79-97, 1989.

[7] Karen A. Lemone, *Design of Compilers : Techniques of Programming Language Translation*, CRC Press, 1992.

[8] R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation," *Conf. Record Fifth ACM Symp. Principles of Programming Languages*, pp.231-240, Jan. 1978.

[9] S. W. Weingart, *An Efficient and Systematic Method of Compiler Code Generation*, PhD Thesis, Yale University, 1975.

[10] 김정숙, 트리패턴매칭기법을 이용한 중간코드 최적화 시스템의 설계 및 구현, 공학박사학위 논문, 동국대학교, 1998.

[11] 김은경, 패턴 매칭을 이용한 GVM SAL 코드 최적화, 공학석사학위 논문, 동국대학교, 2005.

[12] 남동근, 가상 기계를 위한 어셈블리 언어의 설계, 공학석사학위 논문, 동국대학교, 2003.

이 창 환

1998.02 동국대학교 컴퓨터공학과 졸업(학사)
2000.02 동국대학교 대학원 컴퓨터공학과(공학석사)
2003.02 동국대학교 대학원 컴퓨터공학과(공학박사)
현재 링크젠 책임연구원
관심분야 : 프로그래밍 언어, 컴파일러 구성, 임베디드 시스템 등

오 세 만

1985. 03~현재 동국대학교 컴퓨터공학과 교수
1993.03~1999.02 동국대학교 컴퓨터 공학과 대학원 학과장
2001.11~2003.11 한국정보과학회 프로그래밍언어연구회 위원장
2004.06~2005.12 한국정보처리학회 게임연구회 위원장
관심분야 : 프로그래밍 언어, 컴파일러, 모바일 컴퓨팅