LLVM의 해석기 가속을 위한 명령어 셋 디자인

정동헌O 이석영 김재진 문수묵
서울대학교, 전기.컴퓨터 공학부
{clamp, sylee, kjj7999, smoon}@altair.snu.ac.kr,

Instruction Set Design for Accelerating the LLVM Interpreter

Dong-Heon JungO, Seok-Young Lee, Jaejin Kim, Soo-Mook Moon

요 약

LLVM(Low-level Virtual machine)은 최적화된 컴파일 코드 생성을 위한 컴파일러 프레임워크를 목적으로 제작되었다. LLVM은 C언어로 작성된 코드를 효과적으로 머신에 비종속적인 중간코드로 표현하여 사용하므로 이를 잘 활용한다면 C언어를 위한 머신 비종속적인 '가상머신'으로 사용할 수 있다. 하지만 LLVM은 효과적인 컴파일러라는 원래의 설계목적 때문에 전반적으로 동적 수행에 대해 큰 고민 없이 디자인되었다. 이러한 디자인상의 한계는 가상 머신으로서의 성능에는 좋지 않은 영향을 끼치므로 이에 대한 보안이 필요하다. 우리는 LLVM의 명령어 셋에 추가명령어를 제안하여 LLVM 해석기의 성능향상을 얻어낼 것이다.

1. 서론

하나의 단일 프로그램을 다양한 머신에서 동일하게 수행하기 위한 방법으로 가상 머신[1]을 많이 활용한다. 고수준의 프로그래밍 언어로 작성된 프로그램을 머신 독립적인 중간코드 형태로 컴파일하고, 변환된 중간코드는 가상 머신이 설치된 모든 기기에서 동일하게 수행될 수 있다. 대표적인 가상 머신으로 자바 가상 머신[2][3]이 있으며, 자바 언어로 작성된 프로그램은 자바의 중간 코드인 바이트코드 형태로 변환되며, 가상 머신이 설치된 모든 기기에서 자바 바이트코드를 수행할 수 있다.

Low Level Virtual Machine (LLVM)[4][5]은 다양한 언어로 작성된 프로그램의 컴파일 시간, 링크 시간 그리고 수행 시간 전반에 걸쳐 효과적인 최적화를 적용하기 위하여 디자인된 컴파일 시스템이다. C/C++등 다양한 언어로 작성된 프로그램을 다양한 머신에서 효율적으로 최적화를 진행하기 위하여 저수준의 중간코드인 비트코드를 정의하였다. 비트코드는 RISC 명령어와 유사한 형태로 언어와 머신에 비종속적이며, 최적화를 적용하기에 용이하다.

LLVM은 다양한 컴파일 환경을 지원한다. 먼저 일반적인 정적 컴파일러와 같이 C/C++등으로 작성된 프로그램을 비트코드로 바꾸고 이를 다시 머신코드로 변환하여 수행하는 방법을 제공한다.

두 번째는 변환된 비트코드를 LLVM를 통해 동적으로 수행하는 방법을 제공한다. 동적 수행을 위해 LLVM엔 비트코드를 위한 소프트웨어 수행 엔진인 해석기와 수행 시간에 비트코드를 머신코드로 변환하여 수행하는 즉시변환컴파일러 (JITC)[6]를 포함한다. LLVM의 해석기와 JITC를 이용하여 C/C++/Java 언어를 지원하는 가상 머신으로 사용할 수 있다.

하지만 기존 LLVM의 최초 목적은 좀더 최적화된 정적 컴파일러를 활용한 lifetime 최적화[5]를 적용하는 데에 있다. 그렇기 때문에 해석기를 이용하여 빠른게 중간코드를 수행하거나 JITC를 머신 코드로 변환하는데 드는 오버헤드는 거의 고려하고 있지 않다. 대신 비트코드 명령어를 최대한 이해하기 쉽고, 최적화 활용 용도에 맞게 명령어 셋을 디자인하였다.

본 논문에서는 LLVM을 고성능 가상 머신의 용도로 활용하기 위해 비트코드에 몇몇 추가 명령어를 제안하여 LLVM의 해석기 성능을 높이는 것을 목적으로 한다. 특히 프로그램 수행 시 매우 빈번하게 일어나는 수치 연산 명령어를 빠르게 수행하기 위한 명령어 셋을 디자인하였다. LLVM에는 각 수치연산에 대해서 단 하나의 명령어 셋만이 정의되어있다. 우리는 수치연산에서 사용하는 오퍼런드의 타입에 따라 각 명령어를 추가로 정의하였다. 그리고 C/C++에서 쓰이는 switch-case문을 효과적으로 처리할 수 있는 Table Switch 명령을 추가하였다. 그리고 이 효과를 측정하기 위해 간단한 benchmark 프로그램을 제작하였으며, 실험결과를 분석을 통해 이들 명령 추가의 타당성을 검토할 것이다.

2장에선 LLVM에 대하여 설명을 할 것이며, 3장에선 LLVM의 해석기를 위하여 새로운 명령어 셋을 제안할

*. 본 연구는 지식경제부 및 한국산업기술평가관리원의 IT 산업원천기술개발사업(정보통신)의 일환으로 수행하였음 [KI002119, 고성능 가상머신 규격 및 기술 개발]

것이다. 4장에서는 LLVM의 성능을 비교 분석할 것이며 5장에서 본 연구를 정리하도록 하겠다.

2. LLVM 전체 구조

LLVM은 LLVM의 중간 코드인 비트코드를 활용하여 다양한 최적화를 수행한다.

비트코드는 LLVM에서 프로그래밍 언어와 머신 언어에 비종속적인 중간 코드이다. 다양한 언어에서 사용하는 연산들을 사용하기 위하여 수치 연산, 메모리 연산등 대부분의 RISC 머신에서 사용하는 저수준의 명령어 셋들이 포함되어있다. 그리고 객체지향 언어의 특성을 지원하기 위하여 가상 함수 호출등을 지원한다. 명령어뿐만 아니라 다양한 변수 타입을 정의하고 있다. 일반적인 32bit, 64bit 정수 및 실수 타입을 사용자가 정의하여 사용할 수 있으며, 대부분의 언어에서 지원하고 있는 배열 및 구조체, 그리고 공용체등의 타입들을 선언할 수 있는 명령어들을 제공하고 있다.

그리고 자바와 같이 널리 사용되는 언어의 중간 코드인 바이트코드는 스택을 이용한 명령어 수행을 진행하지만 비트코드는 레지스터 기반[7]으로 동작하도록 하며 더 나아가 하나의 변수엔 단 한번의 assign만 있도록 하여 use-def 관계를 쉽게 분석할 수 있는 Static Single Assignment (SSA)[8]형태로 구성되어있다.

LLVM은 비트코드를 활용하여 2가지 수행을 할 수 있다.

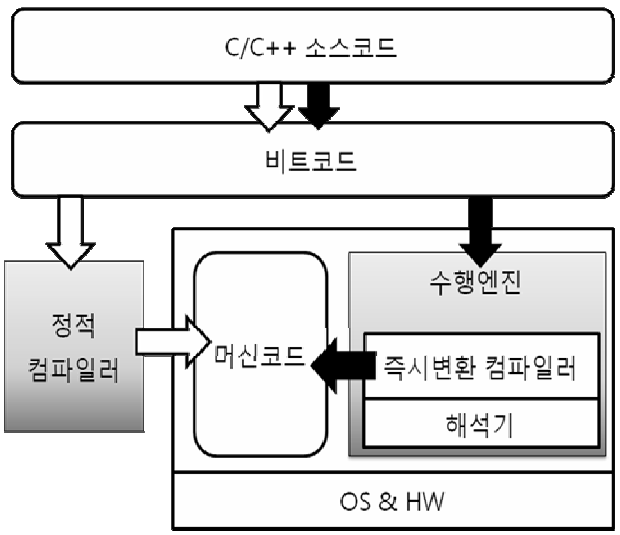

그림 1. LLVM 구성도

위의 그림1 에서와 같이 C/C++로 작성된 파일은 먼저 비트코드로 변환된다. 변환된 비트코드는 정적 컴파일러를 이용하여 머신코드로 변환하여 수행할 수 있으며, 또한 수행 시간에 비트코드를 머신코드로 변환하는 즉시 변환 컴파일러를 이용하여 수행되거나 혹은 소프트웨어 수행엔진인 해석기를 통하여 수행될 수 있다.

LLVM은 위의 모든 단계에서 최적화를 수행하는 lifetime 최적화를 적용한다. 먼저 C/C++ 코드를 비트코드로 변환하며 최적화를 적용한다. 그리고 정적 컴파일러로 오브젝트 파일을 생성 시에 최적화가 일어나며 생성된 오브젝트를 링크하여 하나의 실행 파일을 생성할 때에도 링크타임 최적화를 수행한다. 그리고 즉시 변환기에서도 다양한 최적화들이 수행되며, 한번 변환되었던 머신 코드들도 수행 중 수집한 프로파일 정보를 이용하여 좀 더 최적화된 머신 코드들로 변환할 수 있다.

3. LLVM 명령어 셋 디자인

3.1 정수(integer) 연산과 실수(float) 연산

Java, C등과 같은 언어에서는 저장되는 값의 종류에 따라 다양한 타입으로 분류한다. 일반적으로 수치값을 저장하는 변수의 타입 종류는 크게 정수형(integer)과 실수형(floating point)으로 구분하여 사용하고 있으며, 이들을 이용한 계산은 하드웨어적으로도 다른 계산 방식을 취하기 때문에 소프트웨어 레벨에서도 구분해서 사용하고 있다.

LLVM 비트코드는 C언어에서 사용 가능한 모든 종류의 변수 타입을 지원하고 있다. 비트코드의 명령어는 오퍼런드에 타입정보를 표시하고 오퍼런드에 표시한 타입에 따라 서로 다른 타입의 연산을 수행하는 방식을 이용하고 있다. 즉, 그림 2의 (a)와 같이 LLVM에서는 add의 연산부호(opcode) 뒤에 <ty> 라 표기 되어 있는 타입 정보를 명시하게 되어 있다.

그리고 해석기에서는 그림 2의 (b)에서 보듯이 add라는 연산과 float 혹은 int라는 타입이 따로 기재되기 때문에 LLVM 해석기에서는 먼저 연산을 확인하고 그 이후에 다시 타입 정보를 확인하는 작업을 추가로 거친 후에 그에 맞는 연산을 수행 할 수 있게 된다.

(a) LLVM 명령어 예제

```
add' Instruction
Syntax: <result> = add <ty> <op1>, <op2>
예) c = add float a, b
    c = add i32 a, b
```

(b) add 연산 수행을 위한 LLVM 해석기 예제

```
switch(opcode)
    …
    case 'add'
        switch (type)
            case 'int' : a + b
            case 'double' : doubleADD(a,b)
```

```
        case 'float' : floatADD(a,b)
    …
```

그림 2. Add 연산 예제

수치 연산(arithmetic operation)은 매우 빈번히 일어나고 이때마다 명령어의 종류와 변수의 타입 정보를 따로 확인하는 것은 큰 낭비일 수 있다. 그러므로 그림 3의 (a)와 같이 add 연산에 관련된 명령어를 타입에 따라 명령어를 추가한다. 즉, 각각 수치 연산을 정수(integer) 덧셈과 실수(floating point) 덧셈을 하는 용도로 사용하면 그림 3의 (b)와 같이 정수 연산시 타입 체크 과정이 줄어들어 해석기의 수행속도가 빨라질 것이다.

(a) 최적화된 LLVM 명령어 예제

```
add Instructions
addi : 정수 덧셈
addf : 실수 덧셈
예) c = addf float a, b
     c = addi i32 a, b
```

(b) add 연산 수행을 위한 최적화 LLVM 해석기 예제

```
switch(opcode)
   case 'addi' :
        a + b
   case 'addf' :
        switch (type)
            case 'double' : doubleADD(a,b)
            case 'float': floatADD(a,b)
```

그림 3. 최적화된 Add 연산 예제

3.2 32비트 수치 연산

LLVM 해석기에서 덧셈, 뺄셈, 곱셈, 나눗셈 등의 수치 연산은 operator overloading을 통하여 수치 연산에 사용하는 변수 타입의 bit 수에 따라 서로 다르게 수행된다. 예를 들면 그림 3-2의 정수 덧셈('a+b')의 경우 먼저 실제 변수a, b가 몇 bit인지 확인한다. 확인을 끝난 후에야 연산을 수행한다. 특히 32bit가 아닌 경우 캐리 값(carry)을 반복적으로 넘겨주어 계산하는 bit add를 이용해서 구현되어 있다.

하지만 일반적으로 사용자들이 많이 사용하는 머신은 32비트 머신으로, 32비트 연산을 처리하기 적합하게 되어 있다. 특히 C에서 주로 쓰는 자료형인 'int'의 경우는 대부분 32비트로 되어 있어 LLVM 전단부(front-end)의 컴파일러에서 나온 LLVM 중간코드의 대다수 피연산자(operand)는 32bit integer 형으로 되어 있다. 이렇게 실제 대다수의 명령이 32bit 연산인 상황에서 정수 연산전 bit 수를 체크하는 것은 interpreter에게는 시간 낭비 일 수 있다.

이처럼 주로 많이 사용되는 32 비트 연산을 위한 새로운 명령어를 추가함으로서, 매 연산 때마다 bit수를 체크하는 과정을 줄여 바로 계산하게 만들어 32비트 연산의 수행 속도를 상승시킬 수 있다. 이에 따라 대다수 연산이 32bit 정수 연산인 실제 프로그램들을 LLVM 해석기에서 수행했을 때 성능을 높일 수 있다.

3.3 switch-case문

Switch 문은 조건 값에 따라 특정한 지점으로 점프를 하는 명령이다. C 스타일의 코드에서 이를 살펴보면 그림 4의 (a)와 같은 문법으로 사용한다. 그림 4의 (a)에서 switch 문을 통해 주어진 조건 값(condVar)에 해당하는 case의 영역이 실행되게 된다. 이를 변환한 LLVM 비트코드도 이와 크게 다르지 않다. 그림 4의 (b)를 보면 switch 문은 조건 값을 오퍼런드(1, 2 …)와 비교하여 같은 값일 경우 오퍼런드의 label로 점프한다.

(a) C의 switch-case

```
switch(condVar){
     case 1:   aa()
                break;
     case 2 : bb()
               break;
…
}
```

(b) LLVM 비트코드의 switch-case

```
switch condVar   [1, label1 , 2,   label2 … ]
…
<label1>
          aa()
          branch to end point of switch
<label2>
          bb()
          branch to end point of switch
```

그림 4. Switch-case 예제

LLVM 비트코드에서나 여타 컴파일러들에서도 이들 switch 문이 변환된 machine code는 기본적으로 if-else의 반복과정과 같다.

```
if (condVar == 1) aa();
else if (condVar == 2) bb();
…
```

그림 5. switch-case문이 변환된 if-esle문

그림 4의 과 그림 4-3은 같은 내용의 코드이다. 이럴 경우 해당되는 값이 나올때까지 비교를 반복하는 상황이 발생하고 만약 n개의 case가 있는 switch 문에서는 O(n)의 비교를 해야 된다. 그렇기 때문에 현대의 컴파일러에나 JVM같은 가상 머신에서는

몇가지 제한적 상황에 한해 최적화를 진행한다.

보통의 경우 프로그래머가 작성한 switch-case의 case는 일정하게 증가 혹은 감소 하는 값을 가지는 경우가 많다. 즉, "case 0:, case 1:, case 2:, … "의 순으로 case값이 일정하게 증가하는 경우가 대부분이다. 그렇기 때문에 이런 일정하게 증가하는 case 값을 가지는 switch의 경우 case별로 점프를 할 주소를 table에 저장한다. 그리고 case 값을 인덱스로 하여 점프할 주소 값을 읽어와서 바로 점프하도록 한다. 즉, if문 비교 없이 O(1)으로 빠르게 수행할 수 있다.

LLVM의 중간코드에서 switch의 역할을 하는 명령어는 'Switch'하나이다. 그리고 이 'Switch'명령어는 조건 값과 case 값의 반복적 비교를 통해 다음 실행할 부분이 무엇인지 찾는다. 우리는 switch-case문을 빠르게 수행하기 위하여 table switch라는 새로운 명령어를 추가하였다. Table switch문은 기존의 switch-case의 case값이 일정하게 증가하는 경우 table switch 명령어로 생성하였다.

LLVM 해석기에서 'Switch' 명령어는 그림 6의 (a)와 같이 조건 값과 case의 값에 해당하는 오퍼런드를 비교하한 후에 점프하는 구조이다. 이런 경우 수행시간은 O(n)이다. 하지만 그림 6의 (b)에서와 같이 조건 값을 이용하여 table에서 jump할 주소를 찾아내는 table switch에서는 관계없이 O(1)으로 빠르게 수행할 수 있다.

(a) LLVM 해석기의 'Switch' 명령어 수행

```
executeSwitch(instruction I)
c := condVar(I); // c 조건 값
for a = 0 to Num of operand(I) {
  if c  == case_operand[a] {
    jump to address_operand[a]
  }
}
```

(b) LLVM 해석기의 'TSwitch' 명령어 수행 예제

```
executeTSwitch(instruction I)
c := condVar(I)
jump to address_table[c]
```

그림 6. LLVM 해석기의 switch-case 수행 예제

4. 실험 결과

4.1 실험 환경

실험은 LLVM 2.5에 기반한 LLVM 해석기에 제안한 명령어들을 추가하여, Linux v2.6.28 OS가 설치된 인텔 펜티엄4 싱글 코어 2.80GHz에서 실험하였다.

4.2 실험 결과 및 분석

정수와 실수의 분리를 한 경우 반복적인 피보나치 연산에서 인터프리터가 약 10% 정도 빨라지는 결과가 나왔다. 정수형 연산과 실수형 연산을 나누는 것은 인터프리터의 성능을 향상시키는 것을 확인할 수 있었다. 이러한 내용은 2009년 10월 말에 발표된 LLVM 2.6 버전에서 LLVM 바이트코드 명령어 셋에도 반영되어, 실수형 사칙 연산인 fadd, fsub, fmul, fdiv 명령어가 추가 되었다.

32 bit 전용연산을 추가한 경우에는 피보나치 연산에서 32비트 전용 연산을 추가한 쪽이 인터프리터가 약 4.7% 정도 빨라지는 결과가 나왔다. 인터프리터 수행과정에서 32비트 이하임을 확인하는 체크 과정이 줄어들어서 이와 같은 성능 향상이 나온 것으로 보인다.

Table Switch문은 실험을 위해 실험의 다른 부가적 요소들인 Function call이나 변수 선언등 다른 명령의 실행을 되도록 줄이고 switch case의 반복적인 실행을 실험하기 위한 benchmark를 따로 구현하여 실험하였다.

그림 7은 switch-case문의 실험 성능 측정 결과이다. 그림 7에서 y축 좌표는 benchmark의 수행시간, x축은 switch-case의 case 개수이다. 그림 7에서 나타나듯 일반 switch 명령어는 O(n)의 수행시간을 보이며, case의 의 수에 비례하여 수행시간이 늘어난 반면 tswitch 명령어는 O(1)의 수행시간으로 수행시간이 operand수와 관계없이 일정하게 나타났다.

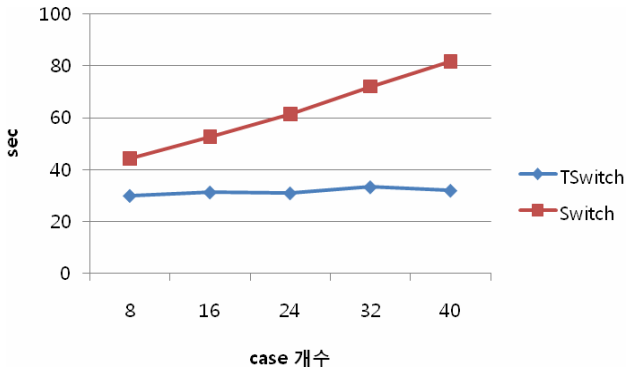

그림7. Switch와 tswich 수행시간 비교

5. 결론

우리는 LLVM의 해석기와 비트코드를 분석하였다. 그리고 분석을 통해 해석기의 산술 연산 처리, switch-case문 처리에 비효율성이 있는 것을 확인할 수 있었다. 우리는 LLVM의 비트코드에 새로운 비트코드 명령어를 추가함으로써 LLVM 해석기의 성능을 높을 수 있었다.

[1] Smith nair, Virtual Machine versatile platforms for systems and processes, Elsevier

[2] JAVA Virtual Machine, O'REILLY, Jon Meyer & Troy Downing

[3] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, 2nd edition, , Prentice Hall, 1 Jul 1999

[4] http://www.llvm.org/
[5] Chris Lattner and Vikram Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, Proc. Of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004
[6] J. Aycock, A Brief History of Just-in-Time, ACM Computing Surveys, 35(2), Jun 2003.
[7] Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl, Virtual Machine Showdown: Stack Versus Registers, VEE'-5, June 11-12, 2005.
[8] Cyron, rom, ferrante, heanne, rosen, barry K. Wgman, Mark N. and zadeck, F. Kenneth, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Transactions on Programming Languages and Systems 13(4) 451-490