Itanium 상의 자바 적시 컴파일러를 위한 가벼운 루프 불변 코드 이동

(Lightweight Loop Invariant Code Motion for Java Just-In-Time Compiler on Itanium)

유 준 민 † 최 형 규 †† 문 수 묵 †††
(Jun-Min Yu) (Hyung-Kyu Choi) (Soo-Mook Moon)

요 약 루프 불변 코드 이동(loop invariant code motion, LICM) 컴파일러 최적화는 비교적 많은 분석 작업을 필요로 하기 때문에 컴파일 시간이 수행 시간의 일부가 되는 자바 적시(Just-In-Time) 컴파일러에는 사용하기 쉽지 않다. "전통적인" LICM 기법에서는 보통 코드를 분석하여 레지스터의 정의-사용 체인과 사용-정의 체인을 미리 만든 뒤 이를 바탕으로 코드 이동을 수행하는 데, 본 논문은 자바 가상 머신(Java virtual machine)이 스택 머신 (stack machine)이라서 좀 더 단순한 코드 형태를 생성한다는 특징을 이용하여 정의-사용 체인을 루프 불변 코드에 대해서만 만들고 사용-정의 체인 없이도 정확히 동작하는 알고리즘을 제시한다.

또한 기존의 방식보다 더 많은 루프 불변 코드 이동을 하게 하는 두 가지 방법을 제시한다. 우선, 간단하기 때문에 루프에 경로가 하나인 경우만 LICM을 적용하는 기존의 기법과 달리, 경로가 여러 개인 루프에서도 부분적으로 중복되는 코드에 대해서도 LICM을 안전하게 적용한다. 또한 부분적으로 중복되는 루프 불변 널(null) 포인터 체크 코드도 Itanium의 조건 수행(predication)을 이용하여 이동시키다.

제안된 기법은 Itanium 마이크로프로세서를 위한 인텔의 ORP(Open Runtime Platform) 자바 가상 머신 위의 적시 컴파일러에 구현하였다. SPECjvm98 벤치마크에 대해 실험을 수행한 결과 전체 적시 컴파일 시간을 1.3% 정도만을 증가시켰지만 전체 수행 시간을 기하 평균으로 2.2% 향상 시켰다.

키워드 : 루프 불변 코드 이동, 자바 적시(JIT) 컴파일러, 널 포인터 체크 코드, IPF

Abstract Loop invariant code motion (LICM) optimization includes relatively heavy code analyses, thus being not readily applicable to Java Just-In-Time (JIT) compilation where the JIT compilation time is part of the whole running time.

"Classical" LICM optimization first analyzes the code and constructs both the def-use chains and the use-def chains, which are then used for performing code motions. This paper proposes a light-weight LICM algorithm, which requires only the def-use chains of loop invariant code (without use-def chains) by exploiting the fact that the Java virtual machine is based on a stack machine, hence generating code with simpler patterns.

We also propose two techniques that allow more code motions than classical LICM techniques. First, unlike previous JIT techniques that uses LICM only in single-path loops for simplicity, we apply LICM to multi-path loops (natural loops) safely for partially redundant code. Secondly, we move loop-invariant, partially-redundant null pointer check code via predication support in Itanium.

The proposed techniques were implemented in a JIT compiler for Itanium processor on ORP (Open Runtime Platform) Java virtual machine of Intel. On SPECjvm98 benchmarks, the proposed technique increases the JIT compilation overhead by the geometric mean of 1.3%, yet it improves the total running time by the geometric mean of 2.2%.

Key words : loop invariant code motion, Java JIT compiler, null pointer check code, IPF

· 본 연구는 Intel Corporation과의 연구 과제와 HP-Intel의 IPF University Grant Program에 의하여 지원되었다.
† 비 회 원 : 삼성전자 반도체 사업부
aidijun@altair.snu.ac.kr
†† 비 회 원 : 서울대학교 전기컴퓨터공학부
hectoct@altair.snu.ac.kr
††† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
smoon@altair.snu.ac.kr
논문접수 : 2004년 2월 19일
심사완료 : 2005년 1월 19일

1. 서 론

현재 많은 자바 가상 머신(Java virtual machine)[1]들은 성능향상을 위해 수행 중에 자바 바이트코드를 기계어 코드로 번역하여 실행시키는 적시 (just-in-time) 컴파일러를 이용하고 있다[2]. 비록 수행 중에 번역을 수행해야 하므로 기존의 정적 컴파일러에서처럼 많은 최적화를 수행하기는 곤란하지만 가능한 한 최적화된 적시 코드를 수행하기 위한 연구[3-6]가 많이 진행되었으며 최근의 적시 컴파일러들은 자바의 성능을 C++의 성능에 육박하게 만들어 주고 있다[3].

기존의 RISC 나 CISC 프로세서 외에 새로운 프로세서들이 등장함에 따라 적시 컴파일러도 이들 프로세서를 위한 최적화 코드를 생성해야 한다. 특히 최근 들어서 대용량의 데이타 처리를 용이하게 하기 위해 64bit 컴퓨터의 필요성이 커지면서 Intel에서명령어 수준의 병렬처리가 가능한 EPIC 구조와 이를 적용한 Itanium 프로세서를 시판함에 따라 EPIC[7] 구조를 위한 효율적인 적시 컴파일러가 중요한 연구 주제로 떠오르고 있다.

Itanium은 현재 6개의 명령어까지 동시에 수행 가능하다. 그래서 어떤 코드를 수행할 때 중요한 경로 데이타 종속 관계상 가장 긴 경로 또는 자주 수행되는 경로에 있는 코드가 전체 수행시간을 지배하기 때문에 효율적인 스케줄링이 효율적인 코드 생성에 중요한 역할을 한다. 그러나 스케줄링은 루프를 기본 단위로 이루어지므로 루프 안의 코드가 밖으로 스케줄이 될 수는 없다. 이는 기존의 루프 최적화를 통해야 가능한 데 루프 최적화를 하기 위해서 많은 분석과 시간을 필요로 하는 것이 대부분이다.

본 논문에서는 그 중 하나인 루프 안의 불변 코드를 밖으로 꺼내는 루프 불변 코드 이동(loop invariant code motion, LICM)을 자바 가상 머신의 특징과 자바 바이트코드(bytecode)의 특징을 이용하여 가볍게 변형하고, 좀 더 효과적인 최적화를 시도 한다. 기존의 LICM을 하기 위해서 필요했던 자료 구조들의 분석 양을 줄이고, 또 고전적인 알고리즘으로는 움직이지 않았던 부분적 중복코드도 대상으로 삼아 더 많은 성능 향상을 기대한다. 또 Itanium의 기능을 이용하여 좀 더 적극적인 방법으로 최적화를 시도한다.

2장에서는 JVM과 본 논문이 구현되어 있는 VLaTTe 적시 컴파일러[8] 그리고 Itanium의 특징에 대해서 간단하게 알아보고, 3장에서는 배경과 기존의 LICM에 대해서 알아본다. 4장에서는 기존의 LICM에서 불필요한 자료구조와 수정된 알고리즘을 말한다. 5장에서는 적극적인 코드 이동 방법으로, 부분적 중복 코드(partially redundant code)에 대한 처리를 설명하고, LICM의 장벽이 되는 널 포인터 예외 확인 코드(null pointer exception check code)를 처리하는 2가지 방법을 말한다. 6장에서는 이것들을 적용한 실험결과를 제시하고, 7장에서는 결론을 내린다.

2. 자바 가상 머신과 IA64 ISA(EPIC)

2.1 자바 가상 머신의 특징과 VLaTTe의 소개

기본적으로 자바 가상 머신은 자바 바이트코드를 수행하는 스택 머신이다. 자바 바이트코드를 수행 가능한 코드(native code)로 변환하는 과정에 있어서 자바의 지역(local) 변수, 스택(stack) 변수, 그리고 임시(temporary) 변수가 생성된다. 이때 지역 변수를 제외한 모든 변수의 유효 범위(live range) 정보는 스택 머신이기 때문에 바이트코드로부터 직접 알 수가 있다. 그렇기 때문에 본 논문에서 제안한 알고리즘이 구현되어 있는 VLaTTe 적시 컴파일러에서는 스택 변수와 임시 변수에 모두 "last use"라는 것을 직접 표시하고 있다. 이는 자바 가상 머신의 피연산자 스택에서 변수가 빠지는 경우, 또는 더 이상 사용되지 않는 경우, 이 정보를 변수에 직접 표시하는 것이다.

스택 머신이기 때문에 스택, 임시 변수들은 유효 범위가 짧아서 대부분 한 basic block을 넘어서지 않는다. 또 자바 언어는 엄격한 프로그래밍(strict programming) 언어이기 때문에 어떤 변수가 절대 먼저 정의 되지 않고서는 사용될 수 없다. 이런 특징을 이용하여 최적화를 빠르게 수행할 수 있다. 반면에 최적화를 하는데 얻는 단점 중에서 가장 큰 것이 정확한 예외 모델(Precise Exception Model)[1]이다. 이는 원래 순서대로 예외(exception)가 발생해야 하고 만일 발생하였다면 효력을 가진 변수[1]들의 값이 최적화되기 전과 된 후에 절대 달라서는 안 된다. 예를 들어서 널 포인터 확인 코드는 다른 예외가 발생 가능한 곳을 넘어서 움직일 수 없고, 또 그 코드와 관련된 하위의 코드들도 움직일 수 없게 된다. 특히 LICM을 적용하는데 있어서 널 포인터 확인 코드는 장벽이 되므로 꼭 이 문제를 해결해야 한다.

기존에는 루프 전체를 벗겨냄(loop peeling)으로 먼저 루프를 한번 수행한 다음부터 순환하게 하여 이 문제를 해결하였다. 그러나 루프에 부분적으로 중복되는 널 포인터 확인 코드가 있다면 이러한 기존의 방법으로는 해결이 불가능 하다. 그렇기 때문에 Itanium의 조건 수행(predication)을 이용하여 해결하는 새로운 방법을 5장에서 제시한다.

2.2 IA64 ISA (EPIC)의 소개

IA64 ISA(instruction set architecture) 다른 말로

1) 지역(local)변수를 의미한다.

EPIC은 Itanium이라는 프로세서에 적용되어 나왔고, 현재 그것의 새로운 기능[7]을 이용하는 연구가 많이 진행중에 있다. 그 중에서 조건 수행은 최근에 여러 프로세서에서 지원이 되고 있는 기능으로 명령어에 1 bit 조건 레지스터(predicate register)를 가리키도록 하여 그 조건 레지스터의 값이 0이면 그 명령을 수행하지 않고, 1이면 수행한다. 즉 선택적으로 명령어를 수행할 수 있어서 제어 구문을 쉽게 변형할 수 있다.

그러나 이런 새로운 기능과 달리 Itanium은 명령어 수준의 병렬처리가 가능하기 때문에 중요한 경로에 있는 명령어를 루프 밖으로 옮기지 않는다면, 전체 루프의 수행 시간은 전혀 줄지 않을 것이다. 그렇기 때문에 어떤 경로가 비록 부분적으로 중복되는 코드라 할지라도 Itanium의 풍부한 레지스터 파일(register file)과 수행 유닛(functional unit)의 자원을 최대한 이용하는 한계 안에서 루프의 수행 시간을 줄일 수 있도록 적극적으로 코드를 움직여야 한다. 특히 tree region[2]을 단위로 스케줄을 하는 VLaTTe 자바 적시 컴파일러는 루프 밖으로 코드가 스케줄 될 수 없기 때문에 스케줄 되기 이전의 IR(intermediated representation) 단계에서 최대한 루프 밖으로 코드들을 옮겨야 한다.

3. 배경과 이전의 작업

3.1 고전적인 LICM의 기본 알고리즘

LICM은 루프 안의 불변 코드를 루프 밖으로 빼내는 것이다[9]. 이 LICM을 적용하는 방법은 크게 세 부분으로 루프 찾기, 루프 불변 코드(loop invariant code) 찾기, 마지막으로 코드 이동(code motion)이다. 루프를 찾는 문제는 본 논문과 관련이 없으므로 생략하고 둘째, 셋째 알고리즘을 설명한다. 여기에서 사용하는 최적화 컴파일러 관련 용어들은 [9]에 기술되어 있다.

루프 불변 코드란 항상 같은 계산 결과를 산출하는 명령어를 뜻한다. 이것을 찾는 문제는 일반적으로 반복 전진 비트-벡터 문제(iterative forward bit-vector problem)[9]이다. 그 알고리즘은 아래와 같다. 여기서 "def"는 어떤 명령어가 정의 하는 변수를 나타내고, "use"는 어떤 명령어가 사용하는 변수를 나타낸다. 그리고 루프 헤더(loop header)는 루프의 시작 basic block을, 루프 프리헤더 (loop preheader)[3]는 루프 불변 코드가 루프 바디(loop body)에 들어오기 전에 미리 수행되어야 하는 코드들이 있는 basic block이다.

• 루프 불변 코드 구하기

1. 루프 헤더 basic block까지 살아 들어오는 변수들을 구해 비트-벡터 IN에 설정
   1) reaching definition[4]을 사용하여 루프 헤더 basic block까지 도달하는 def 변수들을 루프 헤더 basic block의 비트-벡터 IN으로 설정한다.
2. basic block의 IN 비트-벡터를 시작 값으로 해서 그 basic block의 마지막 명령어까지 다음 조건들을 적용해 비트-벡터 OUT을 구하기
   1) 모든 피연산자가 상수이면 비트-벡터에 그 def 변수를 불변이라고 표시
   2) 다음 같은 경우 비트-벡터에 def 변수를 불변이라고 표시
      ① 모든 피연산자가 불변일 때 그것들의 정의가 루프 밖에 있거나
      ② 루프 안에서 단 한번 정의될 때
3. basic block의 OUT 비트-벡터를 이용해 다음 방정식을 풀어 그 하위 basic block의 IN을 구하기
   1) 하위 basic block이 하나의 상위 basic block을 가진다면 하위의 IN은 상위의 OUT이다.
   2) 하위 basic block이 두개 이상의 상위basic block을 가진다면 하위 IN은 상위 OUT 들의 교집합이다.
4. 루프 안의 모든 basic block에 대해서 2~3번을 적용하여 OUT과 IN을 구하기
5. 루프 안의 모든 basic block마다 OUT의 값이 변화가 없을 때 까지 위의 2번부터 4번 항목까지 반복

• 코드 이동(code motion) 하기

1. 코드이동을 할 수 있는 후보 고르기
   1) 명령어의 def 변수가 불변 코드이고
   2) 그 명령어가 있는 basic block이 루프의 중간에서 루프 밖으로 빠져나가는 basic block들(early exit basic block)을 지배해야 하고,
   3) 그 명령어의 def 변수가 있는 basic block이 그 변수의 모든 use 변수가 있는 곳을 지배할 때
2. 후보들을 루프 프리헤더로 옮기기
   1) 루프 안에서 def 변수의 사용이 끝난다면 def 변수와 그 변수의 모든 use 변수를 임시 변수로 바꾸기
   2) 루프 밖에서 def 변수의 사용이 존재한다면 그 임시 변수를 그 def 변수로 복사하는 명령어를 원래 루프 안의 위치에 남기기
3. 루프안의 모든 명령어에 대해서 위의 1번부터 2번 항목까지 반복하기

2) 이는 두개 이상의 선행 basic block을 가지는 곳부터 시작하여 그 아래로 모두 하나의 선행 basic block을 가지는 것들의 영역을 의미한다. 즉 extended basic block과 같은 뜻이다[9].

3) 루프가 시작 되는 헤더(header) basic block 이전에 먼저 실행되게끔 삽입된 basic block

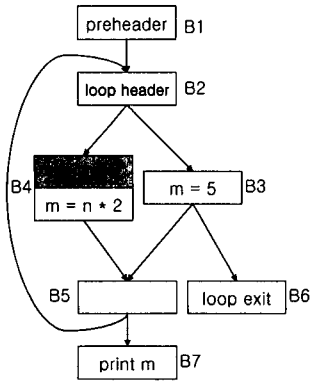
4) 어떤 변수의 정의가 어디까지 유효하게 사용될 수 있는가를 나타내는 자료 구조이다[9].

3.2 고전적인 LICM을 위한 데이타 분석

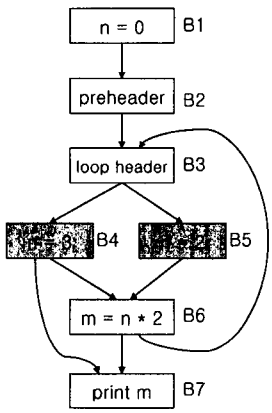
불변 코드를 찾기 위해서는 reaching definition이 필요하고, 변수가 가변 또는 불변인지 그 상태를 나타내는 비트-벡터가 필요하다. 그리고 또 어떤 변수가 루프 안에서 정의가 몇 개가 있는지 알기 위해서 사용-정의 체인(사용-정의 체인, UD)[5] 자료가 필요하다[9]. 코드 이동을 하기 위해서는 움직인 코드의 정의와 그 변수를 사용하는 모든 명령어들의 관계가 필요하다. 이는 정의-사용 체인(def-use chain, DU)[6]자료를 통해 알 수 있다[9]. 이 두 자료를 생성하는 알고리즘은 최악의 경우 $O(n^2)$이기 때문에 비교적 적지 않은 분석 시간을 요구한다. 그리고 고전적인 알고리즘에 의하면, 모든 basic block과 early exit basic block과의 지배 관계가 필요하다. 그 이유는 루프 안의 어떤 basic block, BB_k가 있을 때, 모든 early exit basic block들의 지배자(dominator)에 BB_k가 하나라도 없다면 BB_k는 한 번도 수행되지 않고 루프를 빠져 나갈 수 있기 때문이다. 그러므로 이런 경우를 단순하게 LICM의 대상으로 삼게 되면, 불필요한 코드 이동이었거나 또는 잘못된 경우가 된다. 그래서 복잡한 분석을 통해야만 효과적으로 안전하게 코드를 움직일 수 있다. 그러므로 고전적인 LICM 알고리즘에서는 이 경우는 적용하지 않는 것이다. 간단한 예로 그림 1의 (a)를 보면, B4의 "n = 2" 회색 코드는 비록 불변 코드이지만 n 이라는 변수가 정의되지 않고 바로 루프를 빠져 나갈 수 있으므로 문제가 된다. B6 이후에 n이 한 번도 사용 되지 않는다면 아무런 문제가 없지만 B6 이후의 어떤 basic block이 다른 곳에서 온 n 이란 변수와 충돌이 된다면 정확성에 문제가 생기게 된다. 그림 1의 (b)같은 경우는 다른 관점에서 LICM을 적용하지 않는 경우이다. 값이 상수로서 불변 코드 이지만 루프 안에 정의가 2개 이상 있어서 변수의 이름을 바꾸어서 움직일 수 있지만 실제 사용되는 B6 부분에서 두 개의 값 중에 하나를 선택하는 복사 명령이 필요하게 되므로 전혀 수행시간이 줄지 않는 경우가 된다. 그래서 3.1장에서 말한 알고리즘대로 어떤 변수의 정의가 2개 이상 있는 경우는 LICM의 대상에서 제외 한다.

3.3 기존의 적시 컴파일러에서 루프 최적화

본 논문에 많은 영향을 미친 공개 소스 JVM인 LaTTe JIT 컴파일러에서는 제한적으로 LICM을 하고 있다[3]. Last use를 이용하여 reaching definition을 구하지 않고 분석을 대신하고 있고, 부분적 중복코드와 early exit basic block 문제의 복잡성을 해결하기 위해



(a) 고전적인 알고리즘에서 하지 않는 경우



(b) 절대 움직일 수 없는 경우

그림 1 LICM이 되지 않는 예

서 루프 안에서 경로가 한 개뿐인 간단한 루프를 찾아 LICM을 적용하고 있다. 그렇기 때문에 불변 코드를 찾는 문제는 반복 데이타 흐름 방정식(iterative data flow equation)을 풀지 않고 사용-정의 체인을 이용하여 간단히 구하고 있고, 코드 이동은 정의-사용 체인을 이용해 변수 이름을 임시 변수로 바꾸어 적용하고 있다. 실제 LaTTe의 LICM 분석 시간은 전체 컴파일 시간에 비해 1% 정도로, 전체 수행시간에 비하면 매우 적어 가볍게 구현이 되어 있다.

좀 더 구체적으로 보면, 불변코드를 찾는 문제는 사용-정의 체인을 이용하여 루프 밖에서 정의가 되고, 루프 안에서는 정의 되지 않는 변수로 제한하고 있다. 한 불변 코드가 루프 밖으로 나간다면 사용-정의 체인의 정보를 수정하여 그 def 변수를 사용하는 루프 안의 다른 코드들도 불변 코드가 될 수 있게 만든다. 코드 이동은 정의-사용 체인을 이용해 하고 있다. 움직인 명령어의 def 변수를 임시 변수로 바꾸어 주고, 루프 안의 원래

5) 어떤 변수를 사용할 때 그 변수의 정의가 어디에 몇 개가 있는지 나타내는 자료구조.

6) 어떤 변수가 정의 될 때 그 변수를 어디서 사용하고 있는가를 나타내는 자료구조.

위치에는 임시 변수로부터 원래 def 변수로의 복사(copy) 명령어를 남긴다. 예를 들어 루프 안에 "a = b + 1"이 있을 때 b가 불변이라면, "t = b + 1"이란 명령어를 루프 프리헤더에 넣고 원래 위치에는 "a = t"를 만들어 넣는 것이다. 그리고 def 변수가 루프 밖에서는 절대 사용되지 않는다면 복사 명령어를 없애고 모든 use 변수를 그 임시 변수로 바꾸어 준다. 다시 말해 "a = t"는 없애고 a 대신 t를 사용하도록 a를 모두 교체한다는 것이다. Early exit basic block과의 지배 관계 문제는 루프의 경로가 한 개이기 때문에 자연히 해결된다. 그리고 프리헤더에서 정의된 임시변수는 루프의 모든 exit basic block 다음에 그 임시 변수가 더 이상 사용되지 않는다는 표시를 한다. 즉 변수에 last use를 표시한다. 그렇게 함으로 정확성을 보장 받는다. 위의 예를 계속 이어서 들면 모든 루프를 빠져 나가는 exit basic block 다음에 "t'"을 만들어 주는 것이다.("'" 표시는 last use임을 표시한 것이다. 이 IR 표시는 실제 아무런 코드를 만들지 않고, 정보만 전달하게 된다.)

다른 적시 컴파일러에서는 LICM을 하지 않고 다른 방법으로 루프 최적화를 하기도 한다. SSA(Single Static Assignment)[9,10]를 기반으로 해서 PRE(Partial Redundancy Elimination)[9,11]을 적용한 연구[12]가 있다. PRE는 CSE(Common Sub expression Elimination)[9]와 LICM의 효과를 동시에 볼 수 있는 강력한 최적이지만 SSA와 PRE모두 많은 분석을 요구하는 것이기 때문에 쉽게 접근하기는 어렵다.

본 논문이 구현되어 있는 Intel의 공개 소스 가상 머신인 ORP(Open Runtime Platform)에는 32bit 용 O3 JIT[6] 컴파일러가 있다. 이 경우에도 루프의 경로가 변하지 않는 간단한 경우에 대해서 LICM을 적용하고 있다. 최근에는 LICM 대신 효과적인 방법으로 PRE를 구현한 연구[13]가 있다. 하지만 컴파일 시간은 수십% 증가하였고, 그 시간은 전체 수행 시간에 비해서 본다면 약 1% 이상이나 되어 가벼운 최적화는 아니다. 서버 응용 프로그램 같은 곳에서 효과적으로 사용 될 수 있을 것이다.

4. 개선된 LICM 방법

앞에서 설명한 것처럼 LICM을 적시 컴파일러에 효과적으로 적용하기 위해서는 빠르게 분석해야 하고, 코드 이동(code motion)을 많이 해야 한다. LICM을 적용할 루프는 빠른 분석을 위해서 가장 안쪽 루프(inner most loop)에 대해서 적용할 것이고, 많은 코드 이동을 위해서 자연 루프(natural loop)에 대해 적용할 것이다. 그렇기 때문에 루프 안에 경로가 여러 개가 있을 수 있어 새로운 문제들이 발생하게 된다. 그래서 LaTTe처럼 쉽게 불변 코드를 찾을 수 없고, 고전적인 방법인 반복 전진 비트-벡터 문제(iterative forward bit-vector problem)를 풀어 불변 코드를 찾아야 한다. 그렇기 때문에 일반적인 방법을 적용하지 않고, Java의 특성을 최대한 활용하여 빠르게 문제를 해결할 것이다. 그러한 방법으로 고전적인 LICM에서 만드는 자료들 중에서 만들지 않아도 되는 것을 찾아 알고리즘을 수정할 것이다.

4.1 만들지 않아도 되는 자료들

본 논문에서는 reaching definition, 사용-정의 체인을 제거하고 정의-사용 체인을 필요한 경우만 만들 것이다. 먼저 *reaching definition*은 이전에 설명한 last use를 사용하면 같은 정보를 대신 얻을 수 있기 때문에 제거할 수 있다. Reaching definition은 LaTTe에서도 같은 방법으로 만들지 않고 있지만, 자연 루프의 경우에는 경로가 여러 개가 있어 새로운 문제가 발생한다. 어떤 def 변수가 한 basic block을 넘어서서 사용된다면(이런 것을 exposed def 라고 한다.) 여러 basic block에 last use 가 있게 되므로, 이런 부분을 고려하여 알고리즘을 수정해야 된다.

다음으로 정의-사용 체인은 필요한 경우만 만들 수가 있다. 정의-사용 체인의 필요성은 코드가 움직인 후에 그 def 변수에 대한 모든 use 변수를 임시 변수로 바꾸기 위해서 필요하다. 그렇다면 이것은 불변 코드에 대해서만 필요하므로 모든 코드에 대해서 만들지 않고 불변 코드에 대해서만 필요할 때만 만들 수 있다.

마지막으로 사용-정의 체인은 만들지 않아도 된다. 이것의 루프 안에 정의가 몇 개 있는지 알기 위해 필요했다. 그러나 자바 언어가 엄격한 프로그래밍 언어라는 성질을 이용하여 그것을 만들지 않고 해결할 수 있다. 어떤 exposed def 변수가 있을 때 그것이 불변이라면 실제 그 정의-사용 체인을 이용해서 def 변수를 사용하는 모든 use 변수가 def 변수에 의해서 지배되는지 검사하는 것이다. 여러 경로가 만나는 지점에 어떤 use 변수가 있다면 자바에서는 반드시 각 경로 마다 그 변수의 정의가 존재해야 한다. 만일 정의-사용 체인을 만들 때 어느 use 변수가 있는 지점에서 자신의 정의에 의한 경로가 아닌 다른 경로가 있다면, 반드시 그것은 다른 정의가 존재한다는 것이므로 그 use 변수가 있는 명령어를 "*움직일 수 없음*"이라고 표시 한다. 그리고 이것을 코드 이동(code motion)에서 처리하도록 알고리즘을 수정하면 된다. 그렇게 되면 불변 코드를 찾는 알고리즘은 더욱 간단하게 수정될 수 있다.

4.2 수정된 LICM 알고리즘

비트-벡터 데이타 흐름 방정식(bit-vector data flow equation)의 시작 입력 값은 루프 헤더까지 살아 들어오는 변수들이다. 이 변수들은 reaching definition을 이용

하지 않고 구할 수 있다. 루프 안에서 정의되지 않는 지역 변수와 정의되지 않는 스택 변수 그리고 basic block을 넘어서 존재 할 수 있는 임시 변수 중에서 루프 안에서 정의되지 않는 것들로 한다(임시 변수는 한 basic block을 넘어서 유효하지 않지만, 최적화를 위해서 쓰인 것은 그렇지 않다). 이 시작 비트-벡터 값을 가지고 불변 코드를 찾는 과정은 고전적인 방법과 같지만 위에서 설명한 것처럼 루프 안에서 정의가 몇 개인지는 고려하지 않는다. 대신에 불변 코드이고 exposed def 변수인 경우에만 def 변수와 use 변수 사이에 지배관계를 따져서, 지배되지 않는다면 use 변수가 있는 명령어에 *"움직일 수 없음"*을 표시한다. 이 방법이 효과적인 이유는 스택 머신의 특징 때문에 exposed def 변수의 개수가 적고, 또 exposed def 변수인 경우에 그 값이 불변인 경우는 더욱 더 드물기 때문이다. 즉 *"움직일 수 없음"*이라고 표시되는 것이 매우 적다. 그럼에도 고전적인 알고리즘은 이런 경우를 찾기 위해서 모든 use 변수에 대해서 정의가 몇 개인가 분석하기 때문에 비효율적이다.

코드 이동의 알고리즘은 기존의 방법과 조금 다르다.

• 코드 이동(code motion) 하기

1. 불변코드이고 "움직일 수 없음" 표시가 없다면 정의-사용 체인을 만든다.
   1) 정의-사용 체인에 지배되지 않는 use 변수가 있다면 def 변수가 있는 명령어와 use 변수가 있는 명령어에 "움직일 수 없음"을 표시한다.
2. "움직일 수 없음" 표시 되어 있지 않으면 프리헤더로 움직인다.
   1) def 변수가 있는 명령어의 좌변을 def 변수 대신 임시 변수로 교체 한다. 그리고 프리헤더로 옮긴다.
   2) 지역 변수에 대한 def 변수라면 또는 정의-사용 체인에서 모든 경로에서 last use를 찾지 못했다면, 루프 안의 원래 위치에 좌변은 원래 변수이고 우변은 그 임시 변수인 복사 명령어를 남긴다.
   3) 그 def 변수를 사용하는 모든 use 변수를 그 임시 변수로 바꾸어 준다.
3. 모든 exit basic block에 그 임시 변수에 대한 last use를 명시적으로 넣는다.
4. 루프 안의 모든 불변 코드에 대해서 위의 1번부터 3번 항목을 반복한다.

지역 변수에 대해서 정의-사용 체인을 만들 때는 바이트코드로부터 얻은 last use 정보가 없기 때문에 일단 루프 안에서 사용되는 부분에 한하여 만들게 된다. 그러나 이 문제는 루프 안의 원래 def 변수가 있는 위치에 임시 변수로부터 지역변수로의 복사(copy) 명령어를 남겨서 해결할 수 있다. 만일 지역변수의 모든 use 변수가 루프 안에만 있다면 일반 변수와 같이 처리하면된다.

수정된 알고리즘을 적용한 예는 그림 2를 보면 쉽게 알 수 있다. 그림 2의 (a)는 자바 바이트코드의 IR (intermediate representation) 형태 의사 코드를 나타낸다.

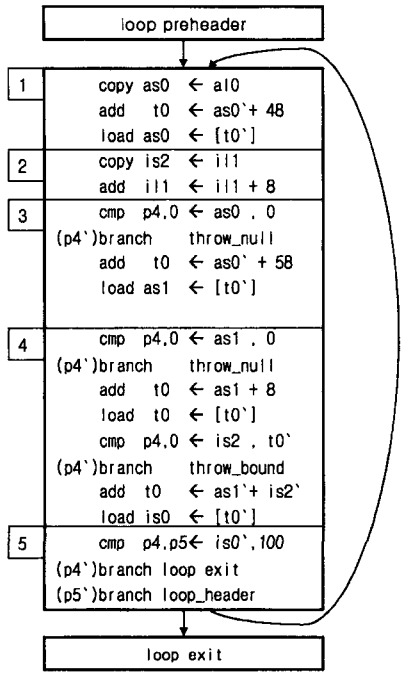
여기서 잠깐 IR의 표기 방법에 대해서 말하면 다음과 같다. 변수에서 첫째 문자는 자바 변수 형을, 둘째 문자는 변수 종류를, 셋째 문자 "m"은 코드 이동이 생성한 임시 변수를 뜻한다. 그리고 마지막 숫자는 그 종류의 몇 번째 변수인지를 나타내는 숫자이다. 자바 변수형 "a"는 객체, "i"는 정수, "p"는 조건 변수(predication)를 뜻한다. 변수 종류 중에서 "l"은 지역 변수, "s"는 스택 변수, "t"는 임시 변수를 뜻한다. 예를 들어 al0은 지역 변수 0번이고, asm1은 스택 변수인 것이 LICM에 의해 움직이면서 생성된 1번의 임시 변수 이다. "cmp"명령어는 비교 명령어로 예를 들어 "cmp p4, 0 ← as0`,0" 코드는 스택 0 에 있는 객체 변수가 널(null)인지 확인하여 널이면 조건 수행 레지스터 4번을 참으로 할당하는 것이다. 그리고 "as1`" 는 더 이상 사용되지 않는다고 "`" 표시가 되어 있다. 그림 2의 (b)에서 루프를 빠져나가는 basic block에 "kill"이라는 내용은 그 변수에 last use만 표시하여 더 이상 사용되지 않는다는 내용을 의사 코드로 나타낸 것이다.

그림 2의 (b)는 위의 수정된 알고리즘을 적용한 예다. 이는 기존의 고전적인 방법을 적용해도 같은 결과를 얻을 수 있다. 다만 만드는 자료 구조의 양이 본 논문이 제안한 양보다 많을 것이다. 그리고 (b) 그림은 자바의 정확한 예외 모델(precise exception model)을 정확히 지키는 범위 내에서 이루어졌다. 실제 루프 바디 안에 아직 많은 불변 코드가 있다. 이를 올리는 방법은 5장에서 다룰 것이다.

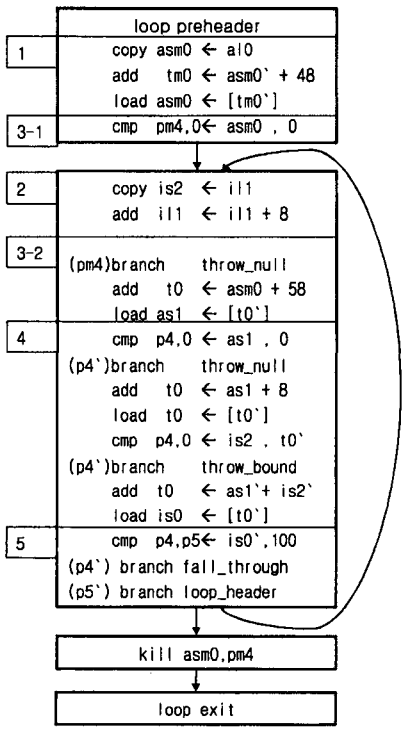
5. 적극적인 LICM 방법

5.1 부분 중복 코드 올리기

본 논문이 구현되어 있는 VLaTTe 적시 컴파일러는 tree region을 기본 단위로 하여 스케줄과 레지스터 할당을 빠르게 동시에 한다. 그래서 그림 1의 (a)와 같이 부분적으로 실행되는 코드도 IPC(Instructions Per Cycle)를 높이기 위해서 다른 경로의 명령어들과 함께 스케줄이 된다. 그 이유는 수행 시간(runtime)에 수집한 분석표(profile) 정보가 없는 한 어느 경로가 자주 수행되는지는 알 수 없기 때문이다. 그러므로 본 논문에서는 루프 부분을 스케줄 했을 때 불변 코드에 의해서 그 수행 시간이 길어지는 것을 방지하기 위해, 이런 부분적 중복 코드들을 Itanium의 풍부한 자원을 적당히 활용하는 한도 내에서 LICM의 대상으로 삼는다. 고전적인 알고리즘에서는 early exit basic block을 지배하지 못하는 basic block을 코드 이동의 대상에서 제외하였지만,



(a) LICM 적용 전의 의사 코드 수준의 예



(b) 기본적인 알고리즘을 적용한 경우

그림 2 LICM 적용의 예

본 논문에서는 그렇지 않다. 루프 안에서 그 정의와 사용이 모두 끝난다면 정확성에 문제가 없게 되므로, 적극적으로 코드 이동을 할 수 있다. 이렇게 되면 레지스터 사용량이 증가하겠지만 "load"와 같은 중요한 명령어들 위주로 LICM을 적용하고, 레지스터 사용량을 확인하는 간단한 heuristic을 적용하면, 빠른 분석을 통해서 더 좋은 성능을 얻을 수 있다. 예를 들어 "store" 명령과 같은 메모리 주소를 가졌기 때문에 읽어 온 값이 가변이라 생각되는 "load" 명령어가 있을 때, 그 주소를 계산하는 산술 연산은 중요하다고 판단되지 않아서 레지스터 사용량을 위해 LICM의 대상으로 삼지 않는다.

현재 적용된 레지스터 사용량을 체크하는 간단한 heuristic은 다음과 같다. 레지스터를 할당 받을 수 있는 유효한 변수들의 총 개수를 짐작하여 그 한계 값을 넘지 않는 범위에서 LICM을 적용하는 것이다. 즉 복사 명령에 의한 변수와 한 basic block을 넘지 않는 유효 범위가 매우 작은 임시변수는 그 개수에 포함되지 않았다.

위와 같은 방법을 적용하여 생성된 임시 변수는 반드시 루프 안에서 그 사용이 끝나게 되므로 모든 루프 exit basic block에 그 임시 변수들의 last use를 명시적으로 넣어 정확성을 보장 받는다. 이 명시적인 last use는 실제 아무런 코드도 만들지 않고 단지 스케줄러와 레지스터 할당 시에 정보만 넘겨준다. LaTTe에서도 경로가 하나인 루프에 대해서 이와 비슷한 방법을 사용하였다. 하지만 경로가 여러 개인 지금 같은 경우는 정의-사용 체인을 통해서 def 변수가 루프 안에서만 유효한지 확인을 하고 적용해야 한다. 그래야만 중간에 루프를 빠져나가는 early exit basic block이 있거나 또는 부분적으로 중복되는 코드에 대해서도 last use를 사용하는 방법이 문제없이 정확하게 확장되어 사용될 수 있다.

5.2 널 포인터 체크 코드 위로 올리기

어떤 객체 변수가 널(null)이 아니라고 보장을 받아야만 그 다음 연산을 수행할 수 있기 때문에, 이 널 포인터 확인 코드(null pointer check code, 이하 줄여서 null check 코드)[7]는 코드 이동에 있어서 커다란 장벽이 된다. 대부분의 경우 메모리로부터 읽어와 새로 정의되는 객체가 루프 안에 있다면, 반드시 null check 코드가 있어야 한다. 만일 이 값이 불변이어도 그 null check 코드를 자바의 정확한 예외 모델(Precise Exception Model)때문에 프리헤더로 움직일 수 없다. 물론 그 객체를 이용하는 그 하위의 명령어들도 함께 움직이지 못하게 된다. 그림 2의 (b)와 그림 3를 비교해 보면 그림 2에서 많은 불변 코드가 움직이지 못함을 알

7) 일반적으로 hardware trap을 이용하거나 또는 비교(compare)와 분기(branch) 명령어를 사용한다. Intel의 O3 적시 컴파일러와 본 논문의 VLaTTe도 후자의 방법을 사용하고 있다.

수 있다. null check 코드 때문에 움직이지 못한 코드들은 불변 코드만 있는 것이 아니라 null check 코드 자체도 있다. 그림 2에서는 4개의 코드가 있지만 null check 코드가 문제가 되지 않는다면 그림 3과 같이 기존의 4개보다 7개의 코드가 더 움직일 수 있다.

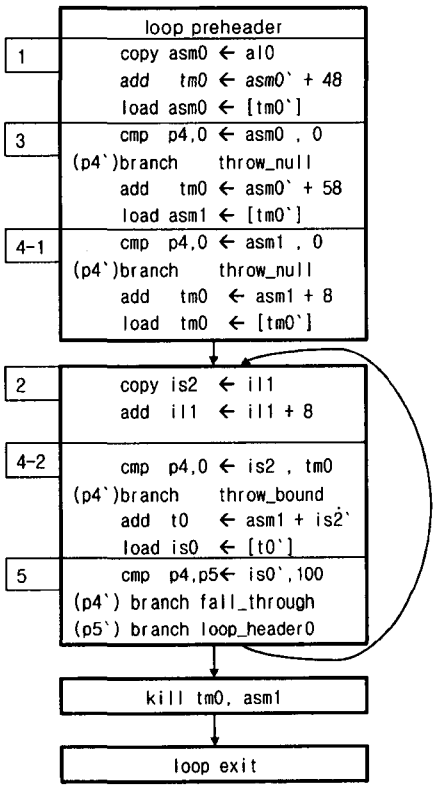


그림 3 널 포인터 확인 코드가 프리헤더로 올라간 예 (그림 2와 같은 의사 코드)

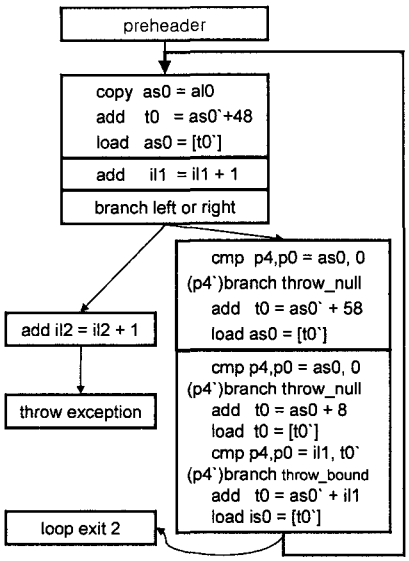
LaTTe에서는 이 문제를 다음과 같이 해결하고 있다. 경로가 한 개뿐인 루프이므로 그 루프의 몸통부분을 한 번 벗겨내어 루프에 들어오기 전에 미리 수행한 후, 안으로 들어오게끔 만드는 것이다. 불변의 값이므로 한번만 null check 코드가 수행되면 되므로, 루프 안에서는 이 null check 코드를 지우면 된다.

하지만 전체 루프가 복사되었기 때문에 수행 경로(control flow)가 복잡해지고 코드의 양도 늘어나게 된다. 특히 이와 같은 방법은 VLaTTe에서 안 좋은 결과를 낼 수도 있다. 그 이유는 VLaTTe는 tree region 단위로 스케줄을 하는데, 복잡해진 수행 경로에 의해서 더 많은 tree region이 생겨서 스케줄의 단위가 작게 나누어지게 되기 때문이다. 그리고 또 이 방법은 경로가 2개 이상인 루프에 적용할 수 없다. 부분적으로 수행되는 null check 코드는 위와 같이 미리 한번 수행해도 다른 경로를 통해 수행되지 않고 루프 안으로 들어 올 수 있기 때문이다. 그래서 이 경우 기존과 달리 프리헤더로 옮기는 두 가지 방법을 제시한다.

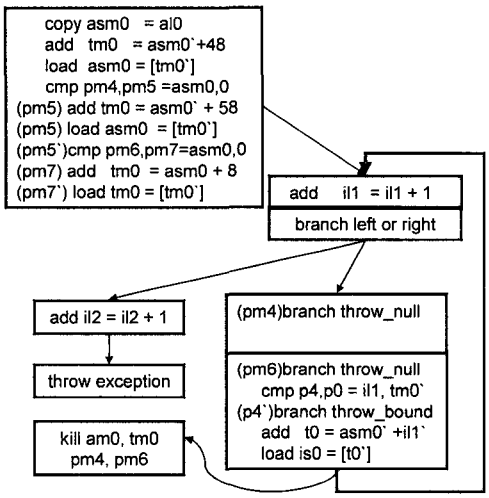
한 가지 방법은 null check 코드가 프리헤더에서 시작해 반드시 거쳐 가는 경로에 있고 (즉 완전히 반복적이어서 항상 수행되는 경우), 또 프리헤더와 null check 코드가 있는 basic block사이에서 다른 예외(exception)가 발생할 가능성이 없어서 자바의 정확한 예외 모델(Precise Exception Model)을 어기지 않는 경우, 그 null check 코드를 프리헤더로 옮기는 것이다. 그림 3이 그 예다. 의사코드는 그림 2와 같은 것을 사용했다. 그림 3을 보면 3과 4-1 부분의 코드가 그림 2 보다 더 많이 옮겨진 것을 확인 할 수 있다.

하지만 그림 4의 예와 같이 early exit block이 중간에 있어서 null check 코드에 도달하기 전에 다른 경로로 빠질 수 있다면, 그 다른 경로에서 먼저 예외가 발생할 수 있기 때문에 프리헤더까지 null check 코드를 옮길 수 없다. 이렇게 부분 반복적인 null check 코드는 미리 수행한다 하더라도 수행이 안 되고 루프에 들어올 수 있으므로 정확성에 문제가 생긴다. Early exit block의 경우나 또는 부분 반복적인 null check 코드를 해결하기 위한 다른 한 가지 방법은 바로 Itanium에 있는 조건 수행을 이용하는 것이다. Null check 코드가 비교(compare) 명령과 분기(branch) 명령으로 이루어져 있는 경우, 비교되는 객체가 루프 안에서 불변일 때, 비교 명령만 프리헤더로 옮기고 분기 명령은 남겨 둔다. 분기 명령은 객체가 널(null)인 경우에만 실행되게끔 조건 레지스터 비트를 설정하고, null check 코드 아래의 그 객체를 사용하는 다른 불변 코드들은 분기 명령이 가진 조건 레지스터 비트의 반대 값을 가지게 하여 프리헤더로 옮긴다. 만일 실제 예외가 발생한다면 분기 명령어의 조건 레지스터 값이 "참"이라고 설정될 것이고, 다른 명령어들의 조건 레지스터 값은 "거짓"이 되기 때문에 수행이 되지 않을 것이다. 이 상태로 계속 수행이 되어 분기 명령어까지 도달 한다면 그때서야 예외가 발생하여 코드를 옮기기 전과 같은 결과를 나타내게 될 것이다. 만일 다른 경로로 중간에 빠졌다면 예외가 발생하지 않아 원래와 같은 결과를 낼 것이다. 이런 방법을 적용한 예가 바로 그림 4의 (b)이다. 이 결과를 보면 대부분의 불변 코드들이 null check 코드의 분기 명령 위로 움직인 것을 알 수 있다. Null check 코드의 비교 명령과 그 아래의 조건 레지스터가 붙은 6개의 코드를 확인 할 수 있다.

이렇게 전자와 후자의 방법을 함께 사용한다면 자바의 정확한 예외 모델(Precise Exception Model)을 깨지



(a) 의사 코드 수준의 예



(b) 조건 수행을 이용해서 널 포인터 확인 코드 넘어서 이동한 경우

그림 4 루프 중간에 빠져 나가는 경로가 있는 경우

않고도 null check 코드를 옮길 수 있고 또 null check 코드의 분기 명령 위로 불변 코드들을 거의 모두 올릴 수 있게 된다. 그렇게 되면 더 좋은 결과를 얻을 수 있다.

6. 실험 결과

6.1 실험 환경

실험에 사용한 시스템은 Intel Itanium 733Mhz, 1GB RAM, EPIC 6-issue architecture이고, OS는 Windows XP 64-bit version 2003 이며, 본 논문이 구현된 VLaTTe 적시 컴파일러가 이용하는 자바 가상 머신은 Intel의 Open Runtime Platform이다[6]. 그리고 사용한 벤치마크는 SPECjvm98[14] 이다.

크게 단일 경로의 루프와 복수 경로를 포함하는 루프에 대해서 비교를 할 것이다. 먼저 분석 시간의 차이를 비교하고, 다음으로 성능 향상을 비교할 것이다.

6.2 분석 시간의 비교

그림 5,6의 "one path loop"은 단일 경로의 루프에 대해서 LaTTe와 같은 방법으로 null check 코드를 해결했을 때 이고, "natural loop"은 복수 경로의 루프에 대해서 조건 레지스터를 이용해서 null check 코드를 해결한 경우이다. 그림 5의 세로축 기준은 LICM을 하지 않은 경우로 분석에 소모한 시간이 0%를 기준으로 하여, 각각 적용했을 때의 백분율이다. 그림 6의 세로축 기준이 되는 것은 LICM을 적용하지 않는 경우로 100%를 기준으로 하여서, 각각 적용 했을 때 백분율이다. 두 그림 모두 LICM을 적용하지 않은 것을 기본으로 삼아 분석 시간/컴파일 시간 증가를 비율로 나타낸 것이다.

분석시간은 단일 경로의 루프가 당연히 적다. 하지만 전체 컴파일 시간의 증가는 더 많다. 그 이유는 LaTTe의 방법이 코드를 복사하여 루프를 미리 먼저 한번 실행하기 때문에 루프의 크기만큼 스케줄과 레지스터 할당 과정에서 시간을 더 소모했기 때문이다. 그렇기 때문

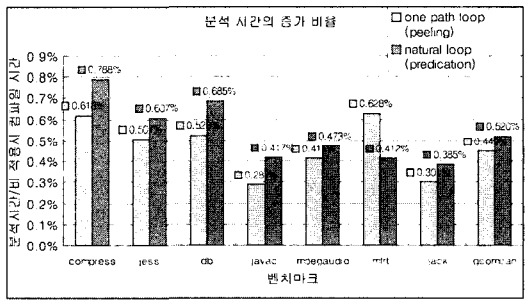


그림 5 LICM 적용시의 분석시간 증가 비율

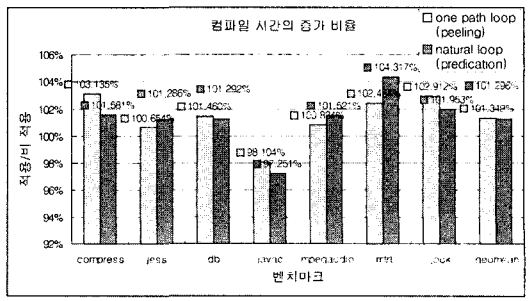


그림 6 LICM 적용시의 전체 컴파일 시간 증가 비율

에 조건 수행을 이용한 경우 비슷한 시간을 들이고도 더 많은 루프에 대해서 LICM을 적용할 수 있었다.

6.3 널 포인터 체크와 LICM의 성능 향상

Null check 확인 문제를 해결하지 않은 경우 많은 성능 향상을 기대하기는 어렵다. 그 이유는 대부분의 경우 루프 안에 null check 코드가 있기 때문이다. 그 실험 결과는 그림 7에 있다. 복수 경로의 루프에 대해서 null check 코드의 문제를 조건 수행을 이용해서 해결한 것과 해결하지 않은 것을 비교하였다. 세로축은 LICM을 적용하지 않는 것을 바탕으로 각각을 적용했을 때 백분을 나타낸다.

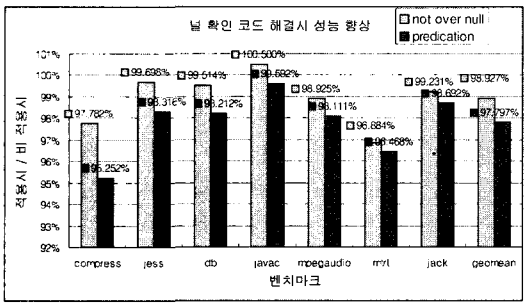


그림 7 null check 코드 문제를 해결한 경우와 그렇지 않은 경우 성능 비교

"not over null"은 불변 코드를 null check코드 넘어서 이동시키지 않은 경우이고, "predication"은 조건 수행으로 해결한 경우 이다. 성능 차이가 상당히 나는 이유는 5장에서 설명한 것과 같이 당연하다고 볼 수 있다.

6.4 단일 경로와 복수 경로의 성능 차이

그림 8은 단일 경로를 가진 루프에 대해서 null check 코드 문제를 해결한 두 가지 방법을 비교하였다. 세로축은 LICM을 적용하지 않는 경우를 바탕으로 각각을 적용하였을 때 백분율을 나타낸다. "peeling"의 방법은 루프를 한번 미리 수행한 한 것이고, "predication"의 방법은 조건 수행을 이용한 것이다. 전체적으로 "predication"의 방법이 더 좋은 결과를 나타내고 있다. "peeling"의 방법은 루프 안에 null check 코드가 사라지지만 루프가 완전히 한 번 복사되었기 때문에 루프를 빠져 나가는 모든 basic block이 새로운 tree region을 만들게 된다. 그러므로 tree region을 단위로 스케줄과 레지스터 할당을 하는 VLaTTe의 경우는 더 낮은 IPC(instruction per cycle)를 가진 코드를 만들게 된다. 그렇기 때문에 조건 수행을 이용한 경우 보다 성능이 덜 나왔다. 조건 수행을 이용하면 null check 코드가 완전히 프리헤더로 옮겨 가거나 또는 루프 안에 분기(branch) 명령어가 남아 있게 된다. 후자의 경우 분기

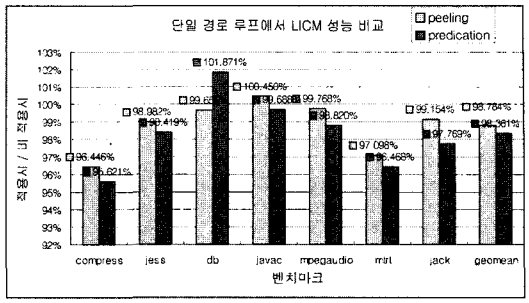


그림 8 단일 경로의 루프에 대해서 null check 코드를 두 가지 방법을 적용해서 해결 했을 때 성능 비교

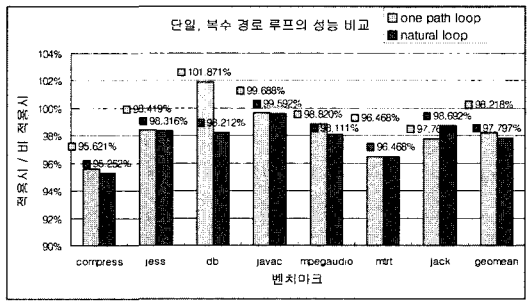


그림 9 단일 경로 루프와 복수 경로 루프에 각각 조건 수행을 이용한 방법을 적용했을 때 성능 비교

명령어는 EPIC의 특성상 다른 중요한 경로의 명령과 함께 대부분 수행되므로 루프 안에서 cycle을 늘리지 않게 되는 경우가 많다. 그런 이유로 "peeling"의 방법보다 좀 더 나은 성능을 보이고 있다. 그러나 벤치마크 db의 경우 "predication"이 더 좋지 않은 성능을 나타내었다. 이 벤치마크의 경우는 단일경로 루프가 크게 성능 향상에 도움이 없었고, 또 조건 수행으로 인해 루프 안에 남은 분기(branch) 명령어가 많은 경우 다른 명령어에 숨어서 실행되지 않았기 때문이다. 그렇기 때문에 프리헤더와 LICM 적용 부담(분석시간과 레지스터 압력) 때문에 더 좋지 않은 성능을 나타내었다.

그림 9는 조건 수행을 이용해 null check 코드 문제를 해결한 경우, 단일 경로만 가진 루프와 단일 또는 복수 경로를 가진 루프에 LICM을 적용했을 때의 성능 비교이다. 세로축은 LICM을 적용하지 않은 경우를 바탕으로 각각을 적용하였을 때 백분율을 나타낸다. "one path loop"은 단일 경로 루프이고, "natural loop"은 단일 또는 복수 경로의 루프이다. 전반적으로 좋아졌고, 특히나 "db"같은 벤치마크의 경우 복수경로의 루프가 매우 자주 실행되는 경우라서 커다란 성능 향상을 보았다. "jack"은 복수 경로까지 적용하였을 때에 이전 보다 더 좋지 않은 성능을 얻었다. 이는 많이 수행되지 않는

루프에 대해서 LICM이 적용되어 부담이 증가되었기 때문이다.

결론적으로 LaTTe의 방법은 많은 코드가 복사되어서 수행되고 또 많아진 tree region 때문에 스케줄 결과도 좋지 않아져 본 논문에서 제시된 방법 보다 낮은 성능 향상을 보였다. 그리고 중요한 것은 복수 경로의 루프에서는 절대 적용될 수 없다. 그렇기 때문에 복수 경로의 루프에서 조건 수행을 이용하여 null check 코드 문제를 해결한 결과가 기하 평균 2.2% 정도j8 가장 좋은 성능을 나타내었다. 또한 이를 위한 분석 시간과 컴파일 시간의 증가는 그림 4,5에서 보듯이 약 0.5%, 1.3%정도 단일 경로에 대해서 LaTTe의 방법을 적용했을 때와 크게 차이가 나지 않았다.

7. 결론 및 향후 과제

본 논문에서는 프로그램의 큰 비중을 차지하는 루프에 대한 최적화인 LICM를 기존의 단일 경로만 적용한 경우에서 복수 경로까지 적용할 수 있게끔 확장하였고, 이에 따라 생기는 여러 문제를 자바의 특성에 알맞게 해결하도록 효과적인 방법으로 제시하였다. 그리고 적극적인 코드 이동으로 좋은 성능을 기대하였다. 먼저 빠른 분석을 위해서 기존의 알고리즘에서 만들던 자료 구조들을 줄였고, 이에 알맞게 알고리즘을 수정하였다. 기존 알고리즘으로 하지 않던 부분 중복 코드에 대해서도 EPIC 환경과 스케줄러에 맞게끔 적극적으로 프리헤더로 옮기는 안전한 방법을 제시하였고, 조건 수행이란 기능을 이용하여 null check 코드를 넘어서 코드 이동을 하도록 하여 많은 성능 효과를 얻었다.

최대한 루프 안에서 많은 불변 코드를 움직임으로써 중요한 경로의 길이를 줄이려고 하였지만, EPIC이라는 환경 때문에 데이타 종속 관계상 가장 긴 경로의 코드가 움직이지 않아서 실제 루프의 cycle이 줄지 않는 경우도 있고, 따라서 더 좋지 않은 결과를 얻기도 하였다. 그렇기 때문에 더 좋은 성능을 얻기 위해서 이런 경우를 효과적으로 빠르게 분석하는 새로운 방법이 요구된다.

또 앞으로 개선해야 할 문제점은 레지스터 사용량에 관한 것이다. 현재는 Itanium에 많은 자원이 있기 때문에 사용하는 레지스터의 양을 간단하게 고려하였지만 이 점도 개선하여 더 정확하게 분석해야 할 것이다. Itanium의 레지스터 스택 엔진(register stack engine)[7] 때문에 비록 레지스터 퇴출(spill)은 일어나지 않지만, 실제 heuristic에서 쓰인 한계 값을 높이게 되면 너무 많은 레지스터를 사용하여 느려진다. 이를 위해서 정확한 레지스터 사용량을 검사하는 heuristic과 위에서 말한 불필요한 코드 이동이 어떤 것인지 알아내는 방법이 필요하다.

마지막으로 LICM말고도 다른 루프 최적화를 적용하여(레지스터 프로모션, array index out of bound exception check 코드 제거 등) 기존의 분석된 자료들을 최대한 이용해야 할 것이다. 그렇게 되면 LICM을 하는데 있어서 제한을 가하던 다른 코드들이 사라져서 더 좋은 성능을 나타낼 수 있을 것이다.

참 고 문 헌

[1] F.Yellin and T. Lindholm, "The Java Virtual Machine Specification," Addisin-Wesley, 1996.

[2] S.-M. Moon and K. Ebcioglu "A Just-in-Time Compiler," IEEE Computer, March 2000.

[3] B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioglu, and E. Altman, "LaTTe: A Java VM just-in-time compiler with fast and ecient register allocation," In Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99), pages 128-138, Newport Beach, California, Oct. 1999. IEEE Computer Society Press. http://latte.snu.ac.kr

[4] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parakh, and James M. Stichnoth. "Fast, Effective Code Generation in a Just-In-Time Java Compiler," In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), pp. 280-290, Montreal, Canada, June 1998.

[5] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth, "Practicing JUDO: Java under dynamic optimizations," In Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00), volume 35.5 of ACM Sigplan Notices, pages 13{26, N.Y., June 18{21 2000. ACM Press.

[6] Open Runtime Platform (ORP), Intel Microprocessor Research Laboratory, http://intel.com/research/mrl/orp

[7] V. IA-Application, "Intel IA-64 architecture software developer's manual volume 1 : IA-64 application architecture."

[8] S. Kim, S.-M. Moon, and K. Ebcioglu. vLaTTe: A Java Just-In -Time Compiler for VLIW with Fast Scheduling and Register Allocation, July 2000. submitted for publication.

[9] Steven S. Muchnick, "Advanced Compiler Design Implementation," pp.252~258, pp.397~406, pp.407~415, Morgan Kaufman, 1997.

[10] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," in ACM Transactions on Programming Languages and Systems. ACM, October 1991.

[11] Robert Kennedy, Sun Chan, Shin-Ming Liu, Raymond Lo, Peng Tu, and Fred Chow. "Partial redundancy elimination in SSA form," ACM Transactions on Programming Languages and Systems, 21(3):627--676, 1999.
[12] Nystrom, N., Hosking, A. L., Cutts, Q., and Diwan, A., "Partial redundancy elimination for access path expressions," Unpublished manuscript. Palsberg, J. and Schwartzbach, M. I. 1994. Object-Oriented Type Systems. Wiley.
[13] Qiong Cai and Jingling Xue, "Optimal and Efficient Speculation-Based Partial Redundancy Elimination," In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'03), 2003.
[14] The Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98/, 1998.

유 준 민
2002년 고려대 전기공학과 학사. 2004년 서울대 전기컴퓨터 공학부 석사. 2004년~현재 삼성전자 반도체 사업부 재직중

최 형 규
2000년 서울대학교 전기공학부 학사. 2002년 서울대학교 전기컴퓨터공학부 석사 2002년~현재 서울대학교 전기컴퓨터공학부 박사과정

문 수 묵
1993년 University of Maryland, Computer Science 박사. 1993년~1994년 Hewlett-Packard, Calf. Lang. Lab. 소프트웨어 엔지니어. 1994년~현재 서울대학교 전기컴퓨터공학부 부교수. 1997년 IBM T. J. Watson Research Center. Visiting Scientist. 2002년~2003년 Sun Microsystems. Visiting Professor. 관심분야는 컴파일러 최적화, 자바 가속, 명령어수준의 병렬처리