濟大論文集(自然). 35. 185-192. 1992
Cheju Univ. Jour. (Natural Sci.),
35. 185-192. 1992

컴파일러 최적화를 위한 코드 스케쥴링

곽호영*, 한정현**

The Instruction Scheduling for Compiler Optimization

*Ho-Young Kwak**, *Jung-Hyun Han***

Summary

This paper deals with instruction scheduling, one area of compiler optimization. This assures the correctness of program execution during occuring the interlock and reduces the execution time in pipelined processor. DAG(Directed Acyclic Graph) is used to represent the problem and is proven to be pretty convienent to represent the various Kinds of serialization constraints.

Gibbon's scheduling algorithm with $O(n^2)$ is introdued to schedule the instruction from DAG. Three heuristics and some assumptions are explained to prevent the problem to be $O(n^4)$. Furthermore, the impact on another phase in compiler is discussed. Actually this scheduling algorithm assumes to perform the instruction scheduling after register allocation. More work relating to this area will be the relationship beween two phases-register allocation and code schduling.

서 론

컴퓨터 구조에서 최근의 연구는 두 분야로 집중되어 있다. 하나는 복잡한 기계어 집합을 갖고 고급언어 시스템을 지원하는 시스크(CISC) 구조이며, 또 다른 하나는 파이프라인과 대량의 레지스터를 바탕으로 하며 단순한 구조를 지원하는 리스크(RISC) 구조이다. 리스크는 구조의 단순성으로 기대되는 하드웨어 비용 절감과 규격화된 명령문들의 일관성으로 인해 고급언어 개발과 컴파일러 최적화 측면에서 활발히 연구가 이루어지고 있는데, 그 특징을 보면 다음과 같다(Hennessy 1983).

· 명령문들의 집합이 단순하므로, 각 명령문의 처리 속도가 빠르다.

· 컴파일러 몇몇 고급 레벨 구조에는 적합하지 않는 복잡한 명령문들을 이용할 필요가 없다.

· 리스크 기종은 구조가 단순한 대신 강력한 컴파일러 기법을 요구한다.

이상에서 열거한 특징들을 가지고 있는 RISC

* 공과대학 정보공학과 (Dept. of Information Eng., Cheju Univ., Cheju-do, 690-756, Korea)
** 홍익대학교 전자계산학과

구조의 컴퓨터 시스템에서는 컴파일러와 코드 생성 기법에 새로운 요구 사항들이 제시되고 있으며, 컴퓨터 구조에 대한 일관성 있는 설계 전략은 컴파일러 설계자가 컴퓨터 구조 자체에 영향력을 갖도록 하고 있다. 동시에 더욱더 강력한 컴파일러와 효과적인 수행을 할 수 있도록 하는 최적화를 요구한다.

따라서, 본 연구에서는 파이프라인 인터록(pipeline interlock)을 갖고 있는 파이프라인 프로세서에서 프로그램의 실행 시간을 줄이고 코드 스케줄링 문제를 DAG를 이용하여 제시하였다.

방 법

1. 파이프라이닝 (Pipelining)

리스크 기계에서 각 명령문이 수행하는 작업이 단순하고 간단하다면 각 명령문을 수행하기 위해서 요구되는 시간은 짧아질 수 있으며 주기 회수는 줄어든다. 성능을 높이기 위해서 리스크 기계에서 일반적으로 다음에서 열거되는 기법들이 사용되고 있다(Kane 1987).

- 명령문 파이프라이닝(instruction pipelining)
- Load/Store architecture
- 지연 적재 명령문(delay allocation instruction)
- 지연 분기 명령문(delay branch instruction)

파이프라이닝은 높은 성능을 기대하는 컴퓨터에서 많이 사용되는 구현 기술이다. 명령문들의 실행을 중첩(overlapping) 함으로써 성능을 증진한다. 이상적으로는 파이프라인의 단계(phase)를 세분화(fine-grained)함으로써 세분화 단계만큼 더 좋은 결과를 기대할 수 있으나, 코드들간의 데이타 의존성, 자원 부족 등으로 인해 파이프라인의 성능 증진은 제약을 받는다.

전형적인 파이프라인 구조에서 각 명령문의 단계는 서로 중첩된다. 한 명령문의 결과가 명령문이 종료되기도 전에 다른 명령문에 의해 요구되거나 또는 자원(resource)이 서로 다른 명령문들에 의해 요구되면, 두번째 명령문은 첫번째 명령문이 종료될 때까지 기다려야 한다. 이 경우 파이프라인 해저드(hazerd)가 발생했다고 말한다. 다행히 연속되는 모든 명령문들 전부가 해저드를 발생하지 않고 레지스터-메모리 이동을 하는 연산에서 해저드가 발생한다. 이 파이프라인 해저드는 파이프라인의 단계를 명령문의 데이타나 자원이 가용될 때까지 저지(stall) 또는 인터록(interlock)하는 원인이 된다. Fig. 1은 3단계-$PS_a \rightarrow PS_b \rightarrow PS_c$로 이루어진 전형적인 파이프라인을 보여준다. 같은 시간에 세가지 명령문-(1 PS_c), (2 PS_b), (3 PS_a)-이 동시에 실행되며, (2 PS_a) 실행은 (1 PS_c) 단계에 앞서 실행된다. 만약 명령문 2가 파이프로 읽혀들어 올 때 페이지 부재(page fault)가 발생하면 명령문 1은 종료되고 페이지 부재는 처리된다. 즉, (2 PS_a)가 수행되면 그 실행 결과는 정확하지 못하다.

1	PS_a	PS_b	PS_c		
2		PS_a	PS_b	PS_c	
3			PS_a	PS_b	PS_c
4				PS_a	PS_b	PS_c

Fig. 1. Example of pipeline.

파이프라인 인터록이라고 불리워지는 하드웨어 메카니즘은 명령문이 요구하는 데이타 값이 가능할 때까지 파이프라인을 저지한다. 파이프라인 구조에서 인터록 메카니즘의 설계는 매우 복잡하며, 높은 성능을 추구할 때 하드웨어 추가 비용을 요구한다. 즉, 명령문들에 인터록이 발생하거나 하지 않거나 모든 명령문들은 인터록 유무가 조사되어야 한다. 이 기능이 제거될 수 있으면 하드웨어 칩의 설계는 한결 단순화될 수 있으며 구조가 단순해질수록 규격화될 수 있고, 궁극적으로 더 작고 빠른 칩이 허용된다.

파이프라인 인터록이 제공되지 않은 컴퓨터 구조에서 컴파일러의 코드 생성기에 의해 제기될 수 있는 문제점은 프로그램의 정확한 실행의 확신 유무이다. 즉, 프로그램의 입출력 기능은 프로그램이 한 단계 형태로 진행되거나 또는 파이프라인 인터록이 없이 진행되거나 관계없이 항상 동일해

야한다.

프로그램 문과 대응되는 정확한 명령문들의 순서와 부정확한 명령문들의 순서를 Fig. 2에서 보인다. 파이프에서 Add 명령문을 실행할 때 Load 명령문은 대부분의 컴퓨터 기종에서는 한 주기 이상을 요구하므로 레지스터 R1의 값이 사용될 수 없다.

따라서 Add 명령문을 실행하기 전에 파이프 단계를 저지 또는 인터록하는 것과 동일한 효과를 갖는 NOP(no-operation)이 추가된다.

Statement	Incorrect code	Correct code
A := B+C	Load B, R1 Add C, R1 Store R1, A	Load B, R1 NOP Add C, R1 Store R1, A

Fig. 2. Incorrect instruction sequence and correct instruction sequence.

프로그램 실행 과정에서 발생하는 인터록의 수를 줄일 수 있는 세 방법을 보면 다음과 같다 (Gibbons 1986).

(1) 하드웨어 기법

하드웨어 인터록이라고 불리워지며 Control Data 6600, IBM 360/91에서 구현, 사용되고 있다. 효과적이나 비용이 많이든다.

(2) 에셈블리 프로그래머가 감지

Rymarczyk가 제안하였으며, 시간 소모가 크며 오류가 쉽게 발생한다.

(3) 컴파일러 감지-코드 스케줄링

많이 사용되는 방법으로서 본 논문에서도 이 방법을 사용하여 효율을 얻는다.

2. 코드 스케줄링 (Instruction Scheduling)

부정확한 명령문들의 순서를 막기 위해 삽입되는 NOP의 수를 줄이기 위해 컴파일러는 코드를 재배열한다. 명령문들의 재배열 -코드 스케줄링-은 실행시간 지연을 줄이기 위해 컴파일 시간에 코드 순서를 재 배열하는 소프트웨어 기법으로서 파이프라인된 프로세서의 프로그램 실행 시간을 줄이기 위해 사용한다(Hennesy 1983, Gibbons 1986).

코드 스케줄링의 장점은 잠정적으로 실행 속도를 빠르게 한다. 만약 코드가 재배열될 수 있고, 그 재배열된 코드들의 실행 결과가 정확하며, 인터록 해결을 위해 추가적인 시간 소모가 크지 않으면 파이프라인 인터록을 가진 기계에서 재배열되지 않은 코드의 실행 속도보다 재배열된 코드 실행 속도는 훨씬 우수하게 된다.

2.1 인터록

파이프라인 구조에서 명령문들의 중첩 실행으로 인해 명령문 i의 결과는 i+k번째 명령문이 실행될 때까지 가용할 수 없는 경우가 있다. 만약 i+k' (k'<k)번째 명령문이 i번째 명령문의 결과를 참조할 경우 파이프라인 지연을 일으킨다. 하드웨어인터록은 이런 경우를 감지하고 데이타가 가능할 때까지 i+k'번째 명령문의 실행을 저지한다. 이 경우를 목적-원시 파이프라인 상충(destination-source pipeline conflict)이라고 한다. 원시-목적 파이프라인 상충(source-destination pipeline conflict)은 목적-원시 상충과 유사하나 참조하려는 데이타가 결과값이 아니고 원시값인데 후에 결과를 계산한다. 목적-원시 상충은 파이프라인 구조에서 가장 흔하게 발생되는 인터록의 형태이다. 반면 원시-목적 상충은 인터럽트(interrupt)나 또는 페이지 부재로 발생한다. 목적-목적 상충도 두개의 명령문이 같은 파이프 단계에서 같은 레지스터에 쓰려고 할 때 발생한다. 그러나 레지스터의 상태는 결정될 수 없으므로 이 상충은 죽은 코드 제거(dead-code removal)또는 코드 재배열 등으로 제거될 수 있다.

2.2 접근 방법

파이프라인 인터록이 제공되지 않는 기계에서 코드 스케줄링 문제를 해결하는 일반적인 두가지 접근 방법은 다음과 같다.

1) 코드 생성 시점에서 수행되는 명령문 스

케줄링 문제로 취급된다.

인터록을 피하기 위한 명령문 스케줄링 문제는 표준 DAG를 기반으로 하는 코드 생성 문제와 합성될 수 있다. 이런 형태에서 인터록을 가진 DAG에서 코드생성 문제는 최소한 레지스터가 기반이 되어있는 코드생성 문제의 최적화 만큼이나 어려우며, 이것은 일반적으로 NP-complete라고 알려져 있다. 더구나 DAG의 경험적 코드 생성 알고리즘은 매우 복잡하다. 이 문제들의 일부분은 트리를 사용하거나 또는 부표현의 공통성을 배제하거나, 또는 한 기본 블럭(basic block)에서 다중 트리의 존재를 부인함으로서 해결될 수 있다. 그러나 단순화된 전략은 실제로 제약점이 너무 많으므로 현실적으로 적용되기 어렵다. 코드생성 최적화로서 명령문 스케줄링은 IBM PL.8 같은 여러 컴파일러에서 구현되어 사용되어지고 있다.

이 방법의 장점은 레지스터 할당전에 명령문 스케줄링을 수행하는 데 있다. 같은 레지스터에 대한 관계없는 사용으로 발생되는 인터록을 최소화할 수 있다는 것이지만 반면에 많은 단점을 가지고 있다.

이 경우 코드 스케줄링은 레지스터의 생존 기간을 확장시킴으로써 레지스터 할당자가 좋은 레지스터 할당을 하는데에 저해 요인이 된다. 그 결과로 발생하는 레지스터 대피(spilling)는 인터록 비용 혹은 추가된 NOP 비용을 초과하는 단점이 있다. 또한 스케줄링은 레지스터 할당전과 마지막 명령문 선택전에 수행하기가 어려운데, 이는 다중 주소 모드나 명령문 형태를 갖고 있는 기종에서 특수한 기능을 구현하기 위해서 사용되는 정확한 명령문과 그 명령문의 인터록은 레지스터 할당 후에나 알 수 있기 때문이다. 그리고, 컴파일링 시스템은 코드 생성 과정에서 여러 단계를 갖고 있다. 이런 경우 코드 최적화가 모든 컴파일 과정에서 부분을 차지한다면 이러한 반복들은 시간을 많이 소모하며 비현실적이다. 따라서 이 방법은 생성 코드의 질이 매우 중요한 요인으로 작용되는 경우에만 적합하다.

그 외에도 명령문 스케줄링 최적화가 코드 생성 과정에 포함되므로 각 컴파일러는 최적화를 반드시 구현해야 하고, 코드 생성 전략은 쉽게 어셈블리 언어로 쓰여진 프로그램에 적용될 수 없다는 단점도 있다.

(2)생성된 코드에 수행될 코드 재배열 문제로 취급된다.

인터록을 제거하기 위해서 코드 생성 후에 기계 코드를 사용하면서 코드를 재배열한다. 이 방법은 Cray-1에서 구현 사용된다. 이 방법의 장점은 다음과 같다.

컴파일러가 생성한 코드와 어셈블리 언어로 작성한 프로그램 코드 양편에 다 적용될 수 있다는 것이다. 하드웨어 인터록이 없으면 어셈블리언어에서 정확한 프로그램을 산출하는 것이 무척 어렵다. 따라서 이러한 프로그래밍을 지원하기 위해서는 소프트웨어 도구가 필요하다. 또한 하나의 문제를 더 작은 문제들로 세분화할 수 있다는 장점도 있다.

본 논문에서는 표준 프로그램 최적화나 표현식 자체를 다시 구성하는 재배열 알고리즘을 다루지 않고 최적화 과정은 이미 코드 재배열 이전에 수행되는 것을 전제로 한다. 따라서 코드 재배열을 수행한 후 결과는 코드 재배열을 수행하기 전과 같은 연산자를 갖는다.

2.3 문제 표현

파이프라인의 제약점을 갖고 있는 코드 스케줄링 알고리즘은 일반적으로 프로그램 실행 시간을 향상시키기 위해서 코드의 순서를 재배열한다. 그러나 재배열된 코드의 순서들은 원래의 연산 순서들을 지켜야한다. 프로그램에서 연산들의 우선 순서를 나타내기 위해서 대그(DAG)가 사용된다. 대그는 한 기본 블럭내의 연산들에 대한 적당한 순서를 정의하는 것이 용이하도록 되어있는데, 대그의 노드는 명령문을 나타내며 아크(arc)는 명령문들간의 순서를 보여준다. 즉, 명령문 노드 1에서 명령문 노드 2로 아크가 존재하면 명령문 1이 실행된 다음에야 실행될 수 있다는 표현이다. 명령문 스케줄링에서 제시되어야할 두가지 고려사항은 다음과 같다.

첫째, 코드 재배열이 원래의 순서를 실행했을 때와 같은 결과를 제공하기 위해서 제약 사항을 표현할 수 있어야 한다.

둘째, 위의 제약사항에 따라 명령문들의 순서를 부여하는 방법이다. 프로그램에서 명령문들은 임의의 순서를 가질 수 없는데, 어떤 명령문들은 반드시 지정된 명령문 다음에 위치해야만 정확한 프로그램의 결과를 얻을 수 있기 때문이다.

문제 표현 방법은 첫번째에서 말하고 있는 제약사항을 표현해야할 뿐만아니라, 명령문들 순서를 표현하는데 최대한 융통성을 나타낼 수 있어야 한다. 일단 재배열의 제약 사항이 결정되면 합리적인 실행시간이 유지되는한 여러 재배열들 중에서 하나를 선택한다. 여기서 스케줄링의 문제는 크게 세 단계로 구분해서 볼 수 있다.

(1) 각 프로시저를 기본 블럭으로 나누다.

(2) 각 블럭의 제약사항을 나타내기 위해 대그를 사용한다.

(3) 그래프에서 lookahead를 사용하지 않고, 경험적 방법을 사용하여 명령문들을 스케줄링한다.

코드 순서와 그에 대응되는 대그를 Fig. 3과 Fig. 4에서 보였다. 레지스터 같은 특수한 자원에서 대그는 명령문들이 정의-정의(명령문 5번과 명령문 9번), 정의-목적(명령문 3번과 명령문 8번), 사용-정의(명령문 4번과 명령문 6번) 관계가 있으면 명령문들의 실행 순서에 제약을 주기위해 아크가 생성된다.

```
1 add    #1, r1, r2
2 add    #12, sp, sp
3 store  r0, A
4 load   -4(sp), r3
5 load   -8(sp), r4
6 add    #8, sp, sp
7 store  r2, 0(sp)
8 load   A, r5
9 add    #1, r0, r4
```

Fig. 3. Basic block instructions.

대그를 구축하기 위해 우선 기본 블럭을 역으로 검조(scan)한다. 검조 과정에서 자원의 정의, 사용에 기초하여 아크를 만든다. 즉, 2번 명령문의 스택 포인터 sp를 정의하고, 4번 명령문은 sp를 사용하거나 혹은 정의하므로, 2번 명령문과 4번 명령문 사이에 제약 순서를 보여주기 위해 아크가 만들어진다. 위의 제약 요건을 만족하여 구축된 대그는 Fig. 4와 같다.

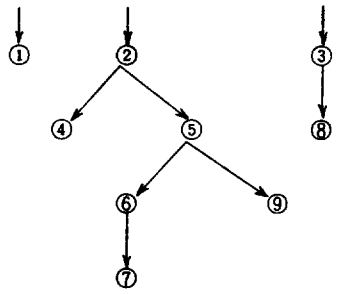

Fig. 4. DAG (Directed Acyclic Graph).

대그는 레지스터 의존 관계 뿐만아니라, 메모리 의존 관계 등의 모든 명령문들의 순서를 나타내는 모든 제약 사항을 표현한다. 대그가 표현해야할 순서에 대한 제약 사항은 다음과 같다.

① 레지스터 의존도(register dependency)

② 메모리 의존도(memory dependency)

③ 프로세서 상태 변경 명령문(processor state -modifying instructions)

④ 올림수/내림수 의존도(carry/borrow dependency)

명령문 재배열을 위해 기본 블럭을 대그로 표현하는 방법은 여러 학자에 의해 연구되어졌다. 그 중에서도 Hennessy와 Gross, 그리고 Gibbons의 대그가 가장 보편적으로 사용된다. Hennessy와 Gross의 방법이 서로 다른점은 전자는 그렇지 않은 반면, 후자는 정의-정의 관계도 순서 제약사항을 부여했다. 이 정의-정의 순서 제약사항의 할당은 스케줄링 알고리즘에서 교착상태(deadlock) 현상을 제거한다.

2.4 정적 평가기(static evaluator)

기본 블럭에서 명령문들이 대그를 위상 정렬 방식으로 스케줄링하면 블럭의 성능 효과는 원래 명령문들의 순서에서 실행과 분리될 수 없다. 알고리즘은 루트에서 시작해서 대그를 따라 내려가면서 스케줄된 명령문을 선택한다. 대그에서 바로 위의 조상 노드들이 없는 노드는 후보자로 뽑히

고, 후보자 그룹에서 가장 조건이 좋은 명령문 노드가 스케줄될 노드로 선정된다.

기본 블럭에서 아직 후보자 그룹에 속하지 못한 명령문들을 미리 고려할 수 있으면 확실히 스케줄링의 성능은 향상된다. 그러나 이 lookahead는 최악의 경우 실행시간을 많이 증가시킨다. 대신 아래에서 보여지는 세 가지 경험적 방법이 사용된다(Gibbons 1986).

① 대그에서 명령문이 다음 자손 노드들과 인터록이 있는가? 인터록이 있는 노드가 선정된다. 즉 아직 선택 노드가 많을 때 인터록을 만드는 노드를 우선적으로 선택하여 스케줄한다.

② 바로 다음 자손 노드들의 수가 많은 노드가 우선적으로 선정된다. 가장 잠재력을 갖고 있는 자손들을 먼저 노출한다. 그러므로 나중에 선택의 폭을 넓게 가질 수 있다.

③ 명령문으로 부터 대그의 단말 노드(leaf)까지 가장 긴 경로를 가진 노드를 선정한다. 대그의 여러 단말 노드들에 대한 경로를 가지고 균형을 맞춘다.

이 휴리스틱의 중요성 정도는 기술된 순서에 의존한다. 대그에서 노드를 선택할때 ①번 조건을 먼저 조사하고 조건을 만족하는 노드가 있으면 그 노드를 선택하고 없으면, 다음 ②번 조건을 검사하고, 다시 없으면 ③번 조건에 기초해 노드를 선택한다.

2.5 스케줄링 알고리즘

실제 명령문을 스케줄링하는 알고리즘은 다음과 같다.

(1) 기본 블럭을 역으로 검조하면서 스케줄링 대그를 만든다. 각 명령문은 그 시점까지 구축된 대그의 노드들과 서로 연관성이 고려되어야 한다.

(2) 대그의 루트를 후보자 그룹에 추가한다. 조상이 없는 노드를 루트로 정의한다.

(3) 후보자 그룹으로 부터 앞의 기본 블럭을 종료시키고, 정적 평가기를 적용시켜 얻은 스케줄될 첫 명령문을 선택한다.

(4) 후보자 그룹이 empty가 아닌 한 다음 ①, ②, ③을 실행한다.

① 마지막으로 스케줄된 명령문에 기초해서 후보자들에 정적 평가기를 적용하여 평가한 다음 가장 유망한 노드를 선택한다.

② 선택된 명령문을 스케줄한다.

③ 새 명령문을 후보자 그룹에서 제거하고, 명령문의 자손들을 후보자 그룹에 넣는다.

스케줄링 알고리즘을 원래의 코드 순서에 적용하면 다음의 Fig. 5와 같은 순서가 바뀐 코드 순서를 얻는다. Fig. 3의 원래 코드에는 네 개의 인터록(3-4, 5-6, 7-8, 8-9)이 있었으나 스케줄된 코드는 1개(8-1) 인터록만 존재한다.

```
3 store   r0, A
2 add     #12, sp, sp
4 load    -4(sp), r3
5 load    -8(sp), r4
8 load    A, r5
1 add     #1, r1, r2
6 add     #8, sp, sp
7 store   r2, 0(sp)
9 add     #1, r0, r4
```

Fig. 5. Scheduled codes.

결과 및 고찰

앞에서 언급된 컴퓨터 구조와 코드 재배열 기법은 컴파일러의 다른 부분—코드 생성기, 최적화, 디버거-에 영향을 미친다.

1. 코드 생성과 지역 레지스터 할당

재배열 알고리즘은 코드 생성기의 생성 코드를 고정된 것으로 가정한다. 발생될 수 있는 문제점은 재배열 단계와 앞서 수행되는 다른 컴파일러 단계와의 연관성이다. 예를 들어, 레지스터 할당이 재배열 이후 이루어진다면 어떤 일이 발생하겠는가? 이때, 지역 할당자의 종류가 달라지면 재배열은 영향을 받게 된다.

x[i]=k+x[i] 문에서 생성되는 명령문을 보기로 한다. portable C 컴파일러가 생성하는 명령문

들의 순서는 Fig.6에서 보여진다. 레지스터들은 가능한 미리 재사용되어진다. 재배열 되지 않는다면 프로그램의 정확한 실행을 위해서는 Fig.6에서 여섯개의 NOP이 추가되어야만 한다.

```
Load J, R0
Load X(R0), R1
Load K, R2
Add  R2, R1
Store R1, X(R3)
```

Fig. 6. Genderdated Instructions after register allocation.

위의 방법과 다른 방법으로는 레지스터를 가능한 재사용하지 말고, 사용되지 않은 레지스터들을 더 많이 사용하는 것이다. 즉, 다른 종류의 레지스터 그룹(홀수/짝수)을 갖거나 또는 가용될 수 있는 레지스터들을 차례로 회전하면서 할당한다.

여기서 두번째 방법을 살펴 보면 네개의 NOP이 코드 재배열된 후 제거되는데, 이때 재배열되어진 코드는 Fig.7과 같다.

```
Load J, R0
Load K, R2
Load I, R3
Load X(R0), R1
NOP
NOP
Add  R2, R1
Store R1, X(R3)
```

Fig. 7. correct instruction sequence.

재배열 과정은 레지스터들의 생존 기간을 확장시킨다. 즉, 레지스터 할당을 하기 전에 스케줄링을 하는 것과 같은 효과를 갖는다. 주된 차이점은 레지스터 할당을 먼저하고 재배열을 하는 방법은 기본 블럭을 계산하기 위해서 필요한 레지스터수를 증가시킴으로써 생존 기간을 확장하지 않는다는 것이다.

2. 전역 레지스터 할당

전역 레지스터 할당도 역시 재배열 과정에 영향을 미친다. 레지스터가 전역적으로 할당될 때, 한 레지스터는 한 기본 블럭의 첫 명령문에서 활성화된다. 이 경우는 레지스터가 전 블럭의 끝 부분에 위치할 때 발생한다. 인터록 기간이 길면 스케줄러는 어떤 레지스터가 블럭 시작에서 인터록에 의해 영향을 받는지를 알기 위해 전 블럭을 찾아야 한다. 대부분 이런 경우 비현실적이다. 한 기본 블럭이 jump 명령문으로 끝날때 pc의 변경을 위해 걸리는 시간은 항상 긴 레지스터 인터록의 길이를 초과한다. 따라서 스케줄러는 한 기본 블럭 내에 순차적 제어 흐름에서 발생하는 인터록만을 고려해도 된다. 이 인터록은 앞의 기본 블럭이 수행될 때 쉽게 계산된다.

3. 디버거

코드 재배열 과정은 계산의 중간 단계가 변경될 때 최적화와 같다. 최적화된 코드를 디버깅할 때 같은 문제가 발생한다. 디버깅 과정에서 코드 재배열의 중요한 효과는 다른 store에 관해서 이것을 옮기는 것이다. 이 상황은 대그로부터 코드를 생성할 때 발생될 수 있는 재배열과 일치한다.

적 요

요즈음 향상된 프로세서 아키텍춰와 VLSI 설계는 컴파일러와 코드 생성 기법에서 새로운 요구사항을 만들어 내고 있다. 컴퓨터 구조를 일관성있게 설계함으로써 컴파일러 설계자는 구조에도 영향을 미치는 것이 가능하며, 동시에 하드웨어/소프트웨어 장단점 여부가 논의의 대상이 된다. 결과적으로 컴퓨터 구조는 더 강력한 컴파일러와 효과적인 수행을 하는 최적화를 요구한다.

본 논문은 파이프라인된 컴퓨터 구조에서 컴파일러 최적화를 제공하는 명령문 스케줄링 알고리즘을 제시한다. 일반적으로 한 기본 블럭 내에서

실행 속도를 향상시키는 명령문 스케줄링 알고리즘의 시간 복잡도(complexity)는 NP-complete이다. 그러나 성능을 높이는 휴리스틱과 파이프라인 인터록 감지가 하드웨어적으로 지원되며, 파이프라인 해저드를 1로 가정할 때 알고리즘은 $O(n^2)$ 복잡도를 보여준다. 또한 이 알고리즘으로 컴파일러의 다른 단계들 (레지스터 할당 등)과 균형적인 관계를 입증하였다.

명령문 스케줄링에 대하여 이 방법의 장점은 의존 대그가 다른 코드 최적화에도 적용될 수 있다는 것이다. 예를 들어 코드 상승(code hoisting)이나 고정 코드 모션(invariant code motion)의 후보자를 의존 대그를 사용하면서 쉽게 감지할 수 있다. 또한 핍홀(peephole) 최적화에도 매우 유용하게 사용될 수 있으며, 그 효과가 다른 방법을 사용한 것보다 우수하다는 것이다. 코드 스케줄링에 관련된 부분으로서 앞으로 더 연구되어야 할 분야는 기본 블럭을 넘는 명령문 스케줄링, 즉 기본 명령문 스케줄링의 확장이다. 이것은 긴 파이프라인 구조를 가지고 또는 다중 실행 장치나 분기 예측을 가진 컴퓨터 구조에서 특히 관심사가 된다.

참 고 문 헌

Bernstein D., Jaffe J. M. and Michel Rodeh. 1989. Scheduling arithmetic and load operations in parallel with no spilling SIAM *Journal on Computing*, 18(6) : 1098~1127.

Bradlee D. G., Henry R. R., and Eggers S. J., 1990. The Marion system for retargetable in struction scheduling, Tech-rep. 90-11-02, Univ. of Washington.

Bradlee D. G., Eggers S. J. and Henry R. R., 1991. Integrating Register Allocation and Instruction Scheduling for RISCs, *ACM*.

Briggs P., Cooper K. D., Kennedy K. and Torczon L., 1989. Coloring Heuristics for register allocation, ACM SIGPLAN conf. on Programming Language Design and Implementation, July.

Chaitin G. J., 1982. Register allocation and spilling via graph coloring, ACM SIGPLAN Symposium on compiler Construction.

Chow F. C. and Hennessy J. L., 1984. Register Allocation by priority based coloring, ACM SIGPLAN Symposium on compiler Construction.

Gibbons P. B. and Muchnick S. S., 1986. Efficient instruction scheduling for a pipelined architecture, In Proceedings of the SIGPLAN '86 Symposium on compiler construction, pp. 11~16.

Goodman J. R. and Hsu W. C., 1988. Code scheduling and register allocation in large basic blocks, In Intl. conf. on Supercomputing, July.

Hennesy J. L. and Gross T. R., 1983. Postpass code optimizations of pipeline constraints, *ACM Transactions on Programing Languages and Systems*, 5(3) : 422~448, July.

Kane G., 1987. MIPS R2000 RISC Architecture, Prentice Hall.

Patterson D. A. and Hennessy J. L., 1990. Computer Architecture : A Quantitative Approach. Morgan Kaufmann Publishers, Palo Alto, California.

Proebsting T. A. and Fischer C. N., 1991. Linear-tiem, Optimal Code Scheduling for Delayed-Load Architectures, Proceedings of the ACm SIGPLAN '91 Conference on Programming Language Design and Implementation, June.