□ 기술해설 □

고성능 마이크로 프로세서를 위한 최적화 컴파일러 동향

서울대학교 정회목* · 문수묵**

● 목 차 ●

1. 서 론

컴파일러는 하나의 언어로 쓰인 코드를 입력으로 하여 같은 기능을 하는 다른 언어로 변환하는 프로그램이다. 일반적으로 컴파일러는 어휘 분석기(lexer), 구문 분석기(parser), 중간 코드 생성기, 중간 코드 최적화기, 코드 생성기, 하위 최적화기를 거쳐 어셈블리 코드를 생성하며 이 과정은 그림 1에 잘 나타나 있다. 이 중에서 코드 생성기 이전의 단계는 마이크로프로세서 하드웨어에 무관하게 설계를 할 수 있지만 코드 생성기와 하위 최적화기는 하드웨어에 의존적인 분야이다.

현대의 마이크로프로세서는 컴파일러가 선택할 수 있는 여러 가지 최적화를 지원하기 위한 하드웨어의 구조를 가지고 있다. 이러한 프로세서의 성능은 프로세서 자체의 클럭 싸이클 시간과 더불어 컴파일러가 최적화 과정에서 하드웨어의 특징을 얼마나 잘 이용하느냐에 달려 있다. 특히 최근의 고성능 마이크로프로세서들은 "명령어 수준의 병렬 처리"(ILP : Instruction Level Parallelism), 캐쉬 선반입(cache prefetching), 강력한 명령어 세트 등의 컴파일러에 의해 제어될 수 있는 구조를 지원하기 때문에 마이크로프로세서 기술은 단순한 칩 제작의 기술이 아니라 하드웨어와 소프트웨어가 결합된 복합된 기술의 형태를 갖게 되었다.

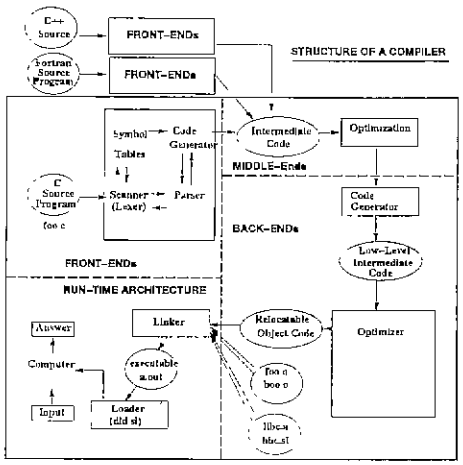

그림 1 컴파일러의 구조도

*비 회 원
**종신회원

이와 같이 최근에 그 중요성이 커진 컴파일러의 최적화 과정을 본 논문에서는 전통적인 최적화 과정과 최근에 많은 연구가 되고 있는 ILP를 위한 최적화 과정으로 나누어 소개한다. 먼저 2장에서는 컴파일러 기술이 프로세서 성능에 미치는 영향을 살펴보고 3장에서는 컴파일러 최적화의 개념을 알아본다. 그리고 4장에서 전통적인 최적화 기법에 대해서 설명하고 5장에서 ILP 프로세서를 위한 최적화 기법을 설명한다. 마지막으로 6장에서 결론을 맺는다.

2. 컴파일러와 마이크로프로세서

최근의 마이크로프로세서의 성능을 나타내는 지표로써는 과거의 MIPS(million instructions per second)와 FLOPS(floating-point operations per second)와 같은 단위보다는 SPEC 벤치마크의 수행 속도를 나타내는 SPECmark가 표준으로 사용되고 있다. 그런데 각 기업들이 발표하는 SPECmark를 살펴보면 재미있는 현상을 찾아볼 수 있다. 즉 같은 CPU 클럭 주파수와 캐쉬 크기를 가진 마이크로프로세서의 SPECmark가 처음 발표할 때보다 시간이 흐름에 따라 서서히 혹은 급속히 증가하는 것을 볼 수 있다. 이러한 성능 증가의 대부분은 최적화 컴파일러의 성능 향상에 의한 것이며 이로 인한 마이크로프로세서의 경쟁력 증가는 많은 이윤을 줄 수 있기 때문에 각 기업들은 최적화 컴파일러의 성능 향상에 많은 투자를 하고 있다.

3. 컴파일러 최적화란?

컴파일러 최적화란 주어진 연산(computation)을 변환하여 같은 결과를 내지만 더 좋은 성능을 갖도록 하는 과정을 말한다. 컴파일러 최적화는 여러 수준에서 기술될 수 있지만 여기서는 마이크로프로세서 성능 향상을 위한 하위 수준의 최적화를 다룬다.

주어진 프로그램의 수행 속도는 다음의 세 가지에 의해 결정된다.

① 명령어 수(instruction count)
② CPI(cycle per instruction)
③ 클럭 싸이클 시간(clock cycle time)

이중 최적화에 의해 제어될 수 있는 부분은 ①과 ②이다. 기존의 "전통적인" 최적화 기법은 ①을 줄이는 데 역점을 두었다면 최근의 최적화 기법들은 ②를 줄이는데 힘을 기울이고 있다고 말할 수 있다.

4. 전통적인 최적화 기법(Conventional Optimization Techniques)

각각의 최적화 기법을 소개하기에 앞서 최적화의 역할을 보여주는 간단한 예를 소개한다. 이 예는 간단한 C 코드와 이를 PA-RISC 컴파일러를 이용하여 생성한 최적화 되지 않은 어셈블리 코드("cc-S test.c")와 최적화 과정을 통해 얻은 어셈블리 코드("cc-S-O test.c")를 보여 주고 있다. 최적화 과정을 통해 루프 안의 명령어 수를 10개에서 단 2개로 줄였음을 알 수 있다.

```
C code
    #define N 100
    main( )
    {
      int A[N], i;
      for (i=0; i<N; i++) A[i]=0;
    }
Unoptimized Assembly
    LDO       148(%r30),%r30
    STW       %r0,-40(%r30)      ; i=0
    LDW       -40(%r30),%r1
    LDI       100,%r31
    COMBF,<,N %r1,%r31,$002      ; i<N
  $003
    LDW       -40(%r30),%r19
    LDO       -440(%r30),%r20
    SH2ADD    %r19,%r20,%r21
    STWS      %r0,0(%r21)        ; A[i]=0
    LDW       -40(%r30),%r22
    LDO       1(%r22),%r1
    STW       %r1,-40(%r30)      ; i++
    LDW       -40(%r30),%r31
    LDI       100,%r19
    COMB,<    %r31,%r19,$ $003 ; i<N
    NOP
  $002
Optimized Assembly
    LDO       -440(%r30),%r31
    LDI       -100,%r23
  $003
    ADDIB,<   1,%r23,$003        ; i++, i<N
    STWS,MA   %r0,4(%r31)        ; A[i]=0
  $002
```

이렇게 최적화된 코드는 여러 가지의 다양한 최적화 기법들을 차례로 적용함으로써 얻을 수 있다. 그런데 이러한 기법들은 그 적용 순서에 따라 그 성능과 비용에 많은 차이가 있을 수 있으므로 그 순서를 결정하는 것(phase ordering) 이 최적화 컴파일러의 중요한 문제 중의 하나이다. 그림 2는 많이 사용되는 최적화 기법들과 그들의 적용 순서를 보여 준다.

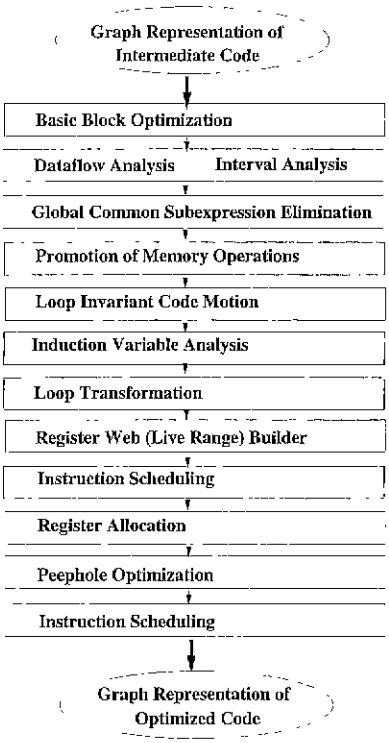

그림 2 최적화 순서도

4.1 기본 자료 구조

최적화 과정의 입력은 어셈블리 수준의 명령어를 하나의 노드로 하는 컨트롤 흐름 그래프(control flow graph)이다. 따라서 최적화의 모든 과정은 그래프 문제로 귀결된다.

4.2 기본 블럭(Basic Block) 최적화

먼저 컨트롤 흐름 그래프를 입력으로 받으면 그래프를 기본 블럭이라는 단위로 분할(partition)한다. 기본 블럭이란 컨트롤 흐름이 블럭의 시작(entry)으로 들어오면 그 블럭의 끝까지 수행되는 연속적인 명령어의 집합이다. 이러한 기본 블럭 내에서 수행되는 최적화를 기본 블럭 최적화라 하고 이에는 다음과 같은 기법이 있다[1].

- 반복되는 같은 계산의 제거(common subexpression elimination) : 위의 예에서 LDW -40(%r30) 은 여러 번 반복되므로 이를 한번만 수행하도록 한다.
- 불필요한 코드의 제거(dead code elimination) : 어떤 명령어의 결과를 아무 도 사용하지 않는다면 그 명령어를 제거 할 수 있다.
- 상수와 복사자의 제거(constant/copy propagation) : COPY %r1, %r2 혹은 LDI 100, %r3 등의 명령어는 계산을 하는 명령어가 아니므로 제거할 수 있다.

여기에 기술한 기법들에 대해 오해하지 않아야 할 것은 이들이 프로그래머가 프로그램을 잘 작성하여 반복되는 계산을 쓰지 않고 불필요한 구문을 쓰지 않는다고 발생하지 않는 것이 아니라는 점이다. 즉 어셈블리 명령어를 생성하게 되면 주소 계산, 같은 주소에 대한 load/store 등으로 인해 반복되는 표현과 불필요한 명령어의 생성을 피할 수 없게 되어 위의 최적화 기법이 필요하게 된다.

실제의 많은 최적화 컴파일러에서는 이러한 기본 블럭 최적화를 확장 기본 블럭(extended basic block)이라는 단위에서 적용한다. 확장 기본 블럭이란 기본 블럭과 같이 컨트롤의 시작은 하나이지만 여러 출구를 가진 블럭을 의미한다. 따라서 기본 블럭에서의 최적화보다는 더 많은 최적화 기회가 주어진다.

4.3 전역 데이터 흐름 분석 (Global Data Flow Analysis)

데이터 흐름 분석은 이 이후에 이루어지는 많은 최적화 기법의 적용을 위해 선행되어야 하는 기법이다. 데이터 흐름 분석을 통해 어떠한 변수(레지스터 혹은 메모리 주소)에 대한 정의(definition)가 있을 때에 이 정의를 사용(use)하는 곳을 바로 찾을 수 있고 반대로 어떠한 변수에 대한 사용이 있을 때 그것을 정의하는 곳을 빨리 찾을 수 있다. 또한 프로그램

의 어떤 위치에서 그 사용이 드러나 있는 모든 변수를 알아 낼 수 있다. 전역 데이터 흐름 분석은 이러한 분석 과정이 기본 블럭을 넘어 전체 프로그램에 대하여 이루어지는 것을 의미하는데 반복(iterative)에 의한 방법과 제거(elimination)에 의한 방법이 있다. 다음은 기본적인 분석 방법인 도달 정의 분석과 라이브 변수 분석에 대한 간단한 설명이다[1].

4.3.1 도달 정의 분석 (Reaching Definition Analysis)

도달 정의 분석은 각 노드에 도달하는 변수의 정의가 어디에서 생성된 것인가에 대한 정보를 분석한다. 여기서 어떤 변수(variable)에 값(value)이 할당되는 명령어를 변수의 정의라 한다. 그리고 변수의 정의가 어떤 지점에 도달한다는 것은 그 정의로부터 어떤 지점까지 경로(path)가 존재하고 그 경로에서 정의가 재정의(kill)되지 않는다는 것을 의미한다[1].

4.3.2 라이브 변수 분석 (Live Variable Analysis)

변수가 어떤 지점에서 라이브하다는 것은 그 지점으로부터 경로가 존재하는 어떤 지점에서 사용된다는 의미이다. 그러므로 라이브 변수 분석에서는 각 노드에서 어떤 변수들이 라이브한가를 결정하게 된다.

4.4 구간 분석(Interval Analysis)

이 과정에서 파악되는 것은 프로그램의 컨트롤 구조이다. 즉 루프와 그들의 계층 구조 그리고 switch, if 등의 구조가 파악된다. 루프의 경우 먼저 각 노드를 깊이 우선 순위로 방문하여 루프의 선두(header)를 찾고 그 루프에 속하는 명령어들을 찾아 루프 계층을 분석한다[1].

4.5 전역 공통 부표현 제거(Global Common Subexpression Elimination)

공통 부표현이란 이전에 이미 계산되었고 그 표현(expression) 내의 변수가 그 계산 후에 변하지 않은 표현을 의미한다. 이 경우에는 이 계산을 반복할 필요가 없게 되고 이전의 결과를 다시 이용하면 된다. 기본 블럭 내에서는 주로 값 순서기(value numbering)를 이용하고 전역에서는 유용(available) 표현 분석을 통해서 불필요한 계산을 제거한다[1].

4.6 메모리 명령어의 승진(Promotion of Memory Operations)

메모리의 라이브 구간(live range)은 같은 주소에 접근하고 각 load에 대해서 그 load에 도달하는 store는 모두 같은 집합에 속하는 store와 load의 집합이다. 메모리 명령어가 승진한다는 것은 store가 복사 명령어로 대치되고 load를 제거하는 것이다. 이를 이용하여 메모리의 접근을 최소화할 수 있다[2].

4.7 루프 변환(Loop Transformation)

루프 변환 기법은 유도 변수 제거(induction variable elimination), 루프 불변이 코드 이동(loop invariant code motion), 계산 세기 낮추기(strength reduction) 등이 있다[3].

4.8 레지스터 라이브 구간 분석 (Register Live Range Analysis)

레지스터 라이브 구간(웹 : Web)은 실제 하나의 레지스터로 할당될 수 있는 변수의 집합이다. 라이브 변수 분석과 도달 정의 분석의 결과를 이용해서 def-use 연결(chain)과 use-def 연결을 생성하고 같은 기계 레지스터로 할당될 수 있는 변수의 정의(def)와 사용(use)을 서로 연결하여 하나의 웹으로 정의한다. 이러한 레지스터 웹을 이용해서 레지스터 할당을 효율적으로 수행케 한다[4].

4.9 명령어 스케줄링 (Instruction Scheduling)

명령어 스케줄링기는 기본 블럭 내에서 하드웨어 인터락(interlock)이나 무익 대기(idle wait) 시간을 줄이기 위해 명령어를 재배치하는 것이다. 물론 이 때에 명령어 순서의 의미(semantics)를 유지해야 한다. 하드웨어 인터락은 파이프라인 내의 명령어간에 같은 하드웨어 자원의 할당 경쟁에서 발생한다. 또한 수치계산과 같이 대기 시간(latency)이 긴 명령어

의 결과를 기다리는 동안 아무 일도 하지 못하고 기다리는 것을 방지한다[5]. 명령어 스케쥴링은 다음에 기술할 ILP 최적화 기법의 핵심이 된다.

4.10 레지스터 할당 (Register Allocation)

레지스터 할당은 주로 그래프 컬러링 알고리듬을 이용해서 수행한다. 먼저 어느 변수(또는 라이브 구간)들이 동시에 라이브 한가를 계산한 간섭 그래프(interference graph)를 작성하고 각 라이브 구간의 코스트를 계산을 한다. 이 코스트를 이용해서 우선 순위를 두고 할당되고 레지스터가 부족하거나 메모리에 저장하는 것이 효율적일 때에 코스트가 작은 레지스터를 메모리에 저장한다(spill)[4].

4.11 핍홀 최적화 (Peephole Optimization)

핍홀 최적화는 어떤 코드 패턴을 동등한 기능을 수행하는 좀 더 짧은 명령어의 집합으로 대체하는 것이다. 핍홀 최적화는 단지 코드의 작은 범위 안에서 수행되지만 간단하면서 효과적이다[1].

5. ILP 마이크로프로세서를 위한 최적화 기법

최근의 상용 마이크로프로세서는 성능 향상을 위해 '명령어 수준의 병렬 처리' 기법을 이용한다. 즉 주어진 순차 코드(sequential code)에서 각 클럭 싸이클마다 상호 의존성(dependency)이 없는 여러 개의 독립적인 명령어를 병렬 수행하여 단일 프로세서의 성능을 크게 향상시키고 있다. 예를 들면 Pentium, PA7100, PowerPC 603, Alpha는 동시에 두 개의 명령어를 수행할 수 있고(2 way) Power, PowerPC 604, Supersparc 등은 세 개의 명령어를 수행시킬 수 있으며(3 way) 4 개에서 6 개까지 동시에 수행시킬 수 있는 Power2, Ultrasparc, PowerPC 620, PA8000 등이 개발되었다.

이러한 병렬 수행을 위해서는 순차 코드에서 독립적인 명령어를 찾아내는 '스케쥴링'이 필요하다. 위에서 언급한 프로세서들은 수행 시간(run-time)에 하드웨어가 스케쥴링을 담당하고 이를 수퍼스칼라(Superscalar) 방식이라 한다. 이러한 스케쥴링을 컴파일 시에 수행하는 것을 VLIW(Very Long Instruction Word) 방식이라 한다. VLIW 시스템에서는 컴파일러가 순차 코드에서 병렬 수행 가능한 명령어들을 찾아내고 이 명령어(operation)들로 하나의 긴 명령어(instruction word : VLIW)를 만든다. 이 VLIW의 하나의 단위 명령어는 보통의 RISC 명령어이며 한 클럭 싸이클에 하나의 VLIW를 수행하면 동시에 여러 개의 명령어를 병렬 수행하게 된다[5,6,7]. 상용화된 VLIW 프로세서로는 Trace, Cydra 5 등과 최근에 발표된 멀티미디어용 프로세서인 Mpact, TriMedia-1, M.f.a.s.t 등이 있다. 그리고 최근에 Intel과 HP의 합작 사업의 일환으로 주목받고 있다. 수퍼스칼라와 VLIW 프로세서를 비교하면 그림 3과 같다.

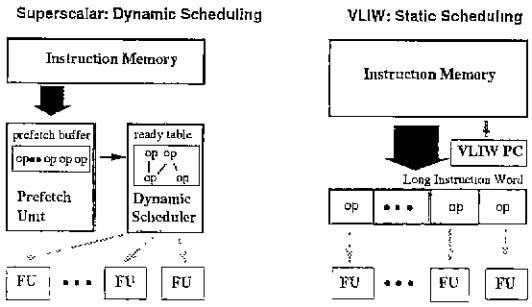

그림 3 수퍼스칼라와 VLIW[8]

수퍼스칼라 방식은 병렬 수행 명령어 수가 증가함에 따라 하드웨어 로직이 기하급수적으로 복잡해지며 이로 인해 싸이클 시간이 증가한다. 또한 그 복잡성으로 개발 과정에서 많은 오류를 발생시킨다. 이러한 이유로 8 way이상의 수퍼스칼라 프로세서에 대해서는 회의적인 전망이 지배적이며 따라서 VLIW의 가능성에 많은 관심을 갖고 있다.

수퍼스칼라 프로세서에서도 성능 향상을 위해서는 컴파일러가 스케쥴링을 해주면 하드웨어가 독립적인 명령어를 더 잘 찾을 수가 있고 VLIW 방식에서는 컴파일러의 역할이 더욱 중

요하다. 이와 같이 최근에 마이크로프로세서의 발전과 더불어 컴파일러의 중요성이 더욱 커져 가고 있으며 최근의 모든 상용 수퍼스칼라 컴파일러에서도 ILP 스케쥴링을 수행한다. 스케쥴링의 목적은 명령어들을 그들의 피연산자(operand)가 가용한 곳으로 최대한 가까이 이동시켜 전체 프로그램의 수행을 빨리 끝내는 것이다.

ILP 스케쥴링은 크게 조건 분기가 많은 범용 정수 프로그램에서 DAG(Directed Acyclic Graph)의 코드를 대상으로 하는 전역 스케쥴링(global scheduling)과 루프(loop)를 대상으로 하는 소프트웨어 파이프라인화 기법(software pipelining)이 있다[9].

5.1 전역 스케쥴링(Global Scheduling)

보통의 비수치 계산용 프로그램에서 기본 블럭은 크기가 보통 5~20 명령어로 작다. 그러므로 충분한 병렬 수행 명령어를 찾아낼 수가 없다. 그러므로 기본 블럭 경계를 넘어서는 전역에서 코드 이동을 통해 스케쥴링 한다. 코드 이동에는 추측 코드 이동(speculative code motion)과 컨트롤 합류점(control join point) 이전으로의 코드 이동을 포함한다. 이러한 코드 이동을 통한 전역 스케쥴링 기법은 트레이스 기반과 DAG 기반의 두 가지로 나눌 수 있다.

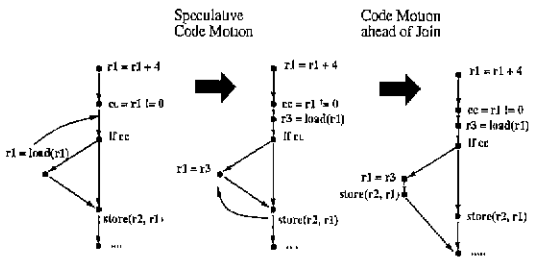

그림 4 추측 및 컨트롤 합류점 이전으로의 코드 이동

5.1.1 트레이스 기반 스케쥴링 (Trace Based Scheduling)

트레이스(trace)는 컨트롤 흐름 그래프 상에 있는 기본 블럭들의 비싸이클 열(acyclic sequence)이며 그 수행 확률에 따라 선택된다. 이 트레이스는 마치 기본 블럭과 같이 취급되며 이 안에서의 코드 이동을 수행 시에 트레이스 밖으로의 분기나 안으로의 분기에는 부기 코드(bookkeeping code)를 생성한다. 하나의 트레이스가 스케쥴이 되면 다시 다음 트레이스를 선택하여 스케쥴을 만든다. 이는 더 이상의 트레이스가 없을 때까지 반복되며 이를 트레이스 스케쥴링이라 한다[6].

수퍼블럭(superblock)은 트레이스를 선택한 후에 트레이스 내로의 분기를 후미 중복(tail duplication) 기법을 이용해서 제거하였다. 이로 인해 코드 이동시에 부기 코드를 생성하는 복잡성을 제거하였으며 이를 수퍼블럭 스케쥴링이라 한다[10].

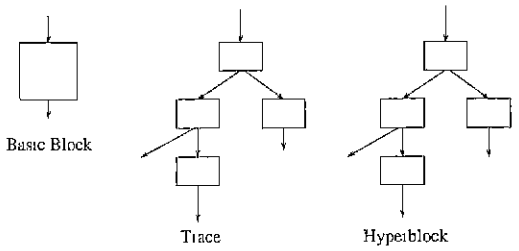

그림 5 트레이스 선택의 비교

하이퍼블럭(hyperblock) 스케쥴링은 트레이스를 선택할 때에 수행 확률 등을 고려하여 어떤 영역을 잡고 그 내부의 분기를 제거하여 하나의 입구와 여러 출구를 가지는 수퍼블럭과 같이 만든 후에 스케쥴링을 수행한다[11].

5.1.2 DAG 기반 스케쥴링 (DAG Based Scheduling)

트레이스를 기반으로 하는 스케쥴링에서는 현재 스케쥴 하는 영역 이외의 부분은 코드 이동의 고려 대상에서 제외된다. 이를 개선한 것이 전체 프로그램 DAG 상에서의 코드 이동을 통한 스케쥴링이다.

삼투 스케쥴링(percolation scheduling)은 모든 수행 경로를 통한 코드 이동을 수행한다. 코드 이동의 경로에 들어오는 분기에는 부기 코드를 생성하고 같은 명령어가 분기의 두 목표에서 코드 이동하였다면 이를 하나로 합친다는 원칙 하에서 계속적인 코드 이동을 수행한

다[12].

컨트롤과 데이터 등가를 이용한 전역 스케쥴링 기법은 먼저 코드 이동의 유용 집합(available instruction set)에서 그 유용성에 의해 코드 이동 대상 명령어를 찾아 나간다[13].

선택 스케쥴링(selective scheduling)은 코드 이동 전에 그 지점으로 코드 이동이 가능한 명령어의 집합을 구하고 이 중에서 코스트가 가장 작은 명령어를 골라서 코드 이동을 수행한다[5,8].

5.2 소프트웨어 파이프라인화 기법 (Software Pipelining)

소프트웨어 파이프라인화 기법은 루프에서 현재의 반복(iteration)이 끝나기 전에 다음 반복의 수행을 시작하여 서로 다른 반복을 중복 수행하는 기법이다. 소프트웨어 파이프라인된 코드는 반복들이 중복 수행되는 파이프라인 핵심(kernel)과 핵심에 이르기까지 초기화를 위해 수행되는 전단부(prologue), 그리고 핵심에서 끝나기 전에 마무리를 위해 수행되는 후단부(epilogue) 코드로 이루어진다. 소프트웨어 파이프라인 기법은 이중에서 핵심의 스케쥴 길이를 줄이는 것을 목적으로 하며 이는 반복 시작 간격을 줄이는 것과 같다 그림 6의 예에서 a)를 소프트웨어 파이프라인화한 것이 b)이고 b)를 수행하면 c)와 같이된다. 싸이클 1,2가 전단부이고 싸이클 3,4가 핵심 코드를 반복 수행한 것이고 싸이클 5,6이 후단부이다. 여기에서 중요한 점은 반복 수행되는 핵심 코드는 같은 패턴의 코드로서 하나의 VLIW를 만들 수 있으며 한 싸이클에 루프의 모든 명령어들 (다른 반복에서 차출된) 을 수행하는 성능 향상을 갖는다는 점이다.

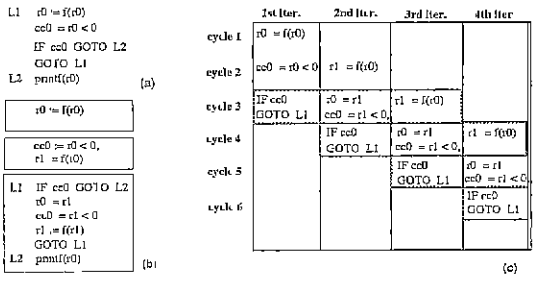

그림 6 소프트웨어 파이프라인화의 예

본 논문에서는 현재 상용화되어 많이 사용되고 있는 모듈로 스케쥴링과 향상 파이프라인 스케쥴링 기법을 간단히 소개한다.

5.2.1 모듈로 스케쥴링(Modulo Scheduling)

이 기법은 먼저 데이터 의존 싸이클(precedence constraints)과 자원 제한 조건(resource constraint)에 바탕을 두어 최소 시작 간격(minimum initiation interval : MII)을 구하고 이를 이용하여 스케쥴링을 시도한다. 만약 이를 실패하면 MII를 하나 늘여서 다시 스케쥴링을 시도한다. 스케쥴링 후에 회전 레지스터(rotating register)나 모듈로 변수 확장(modulo variable expansion)을 이용해서 역 의존성(anti-dependency)을 제거한다.

루프 내에 조건 분기를 포함할 경우에는 먼저 IF 제거(IF conversion) 기법이나 계층 축소(hierarchical reduction)를 통해서 분기를 제거한 후에 모듈로 스케쥴링을 수행해야 하고 이 때의 시작 간격(initiation interval)은 모든 경로에서 최악의 경우로 고정되는 단점이 있다[14,15].

5.2.2 향상 파이프라인 스케쥴링 (Enhanced Pipeline Scheduling)

이 기법은 전역 스케쥴링과 소프트웨어 파이프라인화 기법을 접목하여 정수 코드를 효과적으로 스케쥴링 한다. 즉 싸이클이 있는 그래프에서 어떤 경로(back edge)를 끊어 싸이클을 제거하여 이에 선택 스케쥴링 기법을 적용하여 스케쥴링을 수행한다. 이를 계속적으로 반복 수행하여 하나의 루프를 자동적으로 소프트웨어 파이프라인화 한다[5]

이 기법은 소프트웨어 파이프라이닝의 문제의 문제를 더 단순한 전역 스케쥴링 문제로 변환하기 때문에 모듈로 스케쥴링에 비해서 훨씬 간단하며 동시에 각 경로마다 적절한 시작 간격을 갖는 장점이 있다. 그러나 이 기법은 전역 스케쥴링의 반복을 제어하는 것과 코드 이동의 대상 명령어를 선택하는 방법에 있어 알고리듬적이기 보다는 휴리스틱(heuristic)에 의존하는 단점이 있다.

6. 결 론

본 논문에서는 최적화 컴파일러의 최근 동향에 대해서 전통적인 최적화 기법과 ILP 최적화 기법으로 나누어 살펴보았다. 컴파일러의 최적화 과정은 그 목표 하드웨어의 특성과 밀접한 관계를 맺고 있다. 그러므로 프로세서는 하드웨어와 컴파일러 상호 간에 공조를 잘 이루어야 높은 성능을 얻을 수 있다고 알려져 있다. 특히 VLSI 제작 기술의 한계로 하드웨어 성능 향상의 어려움이 커지고 있으므로 최적화 컴파일러의 중요성은 점점 커져 가고 있다고 할 수 있다.

참고문헌

[1] A.V. Aho, R. Sethi, and J.D. Ullman, 'Compilers : principles, techniques, and Tools", Addison-Wesley, 1986

[2] R.C. Hansen, New Optimizations for PA-RISC Compilers, HP Journal, Jun. 1992

[3] V. Santhanam, Register Reassociation in PA-RISC Compilers, HP Journal, Jun. 1992

[4] P. Briggs, et al., Improvements to Graph Coloring Register Allocation, ACM trans. on Programming Languages and Systems, May 1994

[5] S.-M. Moon and K. Ebcioglu, An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW processors, In Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25), pages 55-71, Dec 1992, also available as IBM Research Report RC 17962.

[6] J.R. Ellis, Bulldog : A Compiler for VLIW Architecture, MIT press, 1986

[7] B.R. Rau, et al., The Cydra 5 departmental supercomputer : Design philosophies, decisions and trade-offs, pages 12-34, Computer, 22, Jan., 1989

[8] S.-M. Moon, Compile-time Parallelization of Non-numerical Code : VLIW and Superscalar, Phd Thesis, U. of Maryland College Park, 1993

[9] B.R. Rau and J.A. Fisher, Instruction Level Parallel Processing : History, Overview, and Perspective, Instruction Level Parallelism, Kluwer Academic Publishers, 1993

[10] W.W. Hwu, et al., The Superblock : An Effective Technique for VLIW and Superscalar Compilation, Instruction Level Parallelism, Kluwer Academic Publishers, 1993

[11] S.A. Mahlke, et al., Effective compiler support for predicated execution using the hyperblock, In Proceedings of the 25th Annual International Symposium on Microarchitecture (Micro-25), pages 45-54, Dec 1992

[12] A. Nicolau, Percolation Scheduling : A Parallel Compilation Technique, Technical Report TR-85-678, Cornell University, 1985.

[13] D. Bernstein and M Rodeh, Global Instruction scheduling for superscalar machines, In Proc. SIGPLAN "91 Conf. on PLDI, pages 241-255, Jun. 1991

[14] B.R. Rau and C. Glaeser, Some Scheduling Techniques and and Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In Proceedings of the 14th Annual Workshop on Microprogramming (Micro-14), pages 183-198, 1981.

[15] M. Lam, Software pipelining : An effective scheduling techniques for VLIW machines, In Proc. SIGPLAN'8 Conf. on PLDI, pages 318-327, Jun. 1988

정 회 목

1993 서울대학교 전자공학과 학사 졸업
1995 서울대학교 전자공학과 석사 졸업
1995~현재 서울대학교 전기공학부 박사 과정 재학중
관심분야 : 최적화 및 병렬화 컴파일러, 컴퓨터 구조

문 수 묵

1987 서울대학교 컴퓨터공학과 학사 졸업
1990 Univ. of Maryland 전산학과 석사 졸업
1993 Univ. of Maryland 전산학과 박사 졸업
1991~93 IBM T. J. Watson 연구소 객원연구원 (VLIW 컴파일러 개발)
1993 ~ 94 Hewlett-Packard Company 소프트웨어 엔지니어 (최적화 컴파일러 개발)
1994~현재 서울대학교 전기공학부 전임강사
관심분야 : 최적화 및 병렬화 컴파일러, 마이크로프로세서 구조, 병렬 처리