□ 기술해설 □

하드웨어 의존적 병렬처리

VLIW 시스템을 위한 컴파일러 최적화 기법

고려대학교 박명순*·이진호**·양정일**
대신대학교 박성순***

목 차

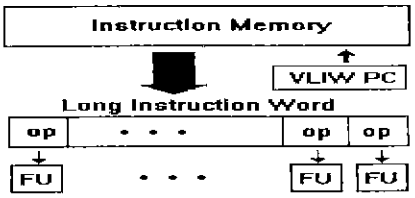
1. 서 론

강력한 계산 능력 요구와 하드웨어 기술 향상 및 비용 하락의 영향으로 병렬 컴퓨터에 대한 연구개발이 활발히 진행되면서 벡터 프로세서(vector processor), 다중프로세서(multiprocessor), 데이타플로우(dataflow) 등 다양한 형태의 병렬 컴퓨터가 제시되었다[18]. 이들 병렬 컴퓨터의 특징을 간단히 살펴보면, 벡터 프로세서는 하나의 명령어로 여러 데이타에 대해 동일한 연산을 수행한다. 따라서, 대량의 데이타를 사용하는 과학 계산용으로는 적합하지만 범용 컴퓨터로 사용하기에는 적합하지 않다. 다중프로세서는 기존의 단일프로세서를 확장한 개념으로 비교적 구현이 용이하면서, 과학 계산은 물론 범용 컴퓨터로 사용할 수 있다는 장점이 있다. 그러나, 프로세서들에 대한 균등한 작업 분배의 어려움, 프로세서간의 동기화 및 데이타 교환을 위한 오버헤드 등의 문제들로 인해 프로그램 전체 또는 부 프로그램 수준인 큰 단위의 병렬성(large grain parallelism)을 이용하는 것이 일반적이다. 데이타플로우의 경우 폰 노이만 방식을 벗어나 명령어 수준인 세부 단위의 병렬성(fine grain parallelism)을 이용하므로 이론적으로는 매우 좋은 효율을 보이지만, 실제 구현시 여러 문제로 인하여 그와 같은 효율을 얻을 수 없으며 범용 컴퓨터로 활용하기에는 적합하지 못하다. 예로든 이들 세 가지 대표적인 병렬 컴퓨터의 특징들을 종합해볼 때, 이상적인 병렬 컴퓨터의 조건을 간단히 정리하여 보면 구현이 용이하고, 가격 대 성능비가 높고, 높은 병렬성을 제공하고, 범용 컴퓨터로 사용될 수 있고, 프로그래밍이 쉬워야 한다는 점 등을 꼽을 수 있다. 현재까지 제안된 여러 병렬 컴퓨터들 중, 비교적 이들 조건을 잘 만족시키는 병렬 컴퓨터로는 VLIW(Very Long Instruction Word) 시스템을 들 수 있다.

VLIW 시스템이라는 명칭은 한 번에 처리되는 수행 단위인 명령 워드(instruction word)의 길이가 수 백~수 천 비트로 일반 프로세서의 명령어보다 매우 길다는 점에서 유래한다[10,12,14]. 전형적인 VLIW 시스템의 구성은 그림 1과

* 종신회원
** 준회원
*** 정회원

같다. 하나의 명령 워드는 수~수 십개 이상의 부속명령어(sub-instruction)들로 이루어진다. 예를 들면, op_1, op_2, op_3, op_4, … 등의 여러 명령어들이 있을 때, 이들을 프로세서[1] 수 만큼의 일정 단위로 묶어서 하나의 명령 워드 {op_1, op_2, op_3, op_4, … op_n}을 구성하며 이 명령 워드내의 각 부속 명령어는 여러 프로세서에서 분산되어 동시에 수행된다. VLIW에서의 한 명령 워드는 단일프로세서에의 한 명령어와 동일한 개념이므로 하나의 PC(Program Counter)에 의해 전체 프로그램 수행의 흐름이 제어된다. 이와 같은 VLIW 시스템의 작동 방식은 최근 프로세서에 널리 적용되는 슈퍼스칼라(superscalar)[15] 기법에서 동시에 두 개 이상의 명령어들을 수행하는 개념과 유사하다.



(FU: Functional Unit, OP: op-code, VLIW PC:Program Counter)

그림 1 전형적인 VLIW 시스템

표 1에서는 다중프로세서, 벡터 프로세서와 VLIW 시스템과의 비교를 통해 VLIW 시스템의 전반적인 특징을 간단히 설명하였다. 표 1에서 알 수 있듯이, VLIW 시스템에서의 가장 두드러진 특징은 전반적으로 하드웨어의 복잡도가 매우 줄어드는 반면 컴파일러의 역할이 매우 증가된다는 점이다. 특히, VLIW 시스템에서 프로그래밍 요구사항 최소화란 일반적인 순차 프로그래밍 기법과 언어를 그대로 사용할 수 있다는 것을 의미하며, 이에 대한 병렬화 작업은 전적으로 컴파일러가 수행한다. 고급 언어로 작성된 순차 프로그램이 주어지면 VLIW 시스템의 컴파일러는 먼저 단일프로세서를 위한 실행 프로그램을 생성한다. 그 후 실행 프로그램의 명령어들을 적절히 묶어 명령 워드를 구성하는 작업(compaction)을 수행한다. 단, 명령 워드 내의 모든 명령어는 동시에 수행되므로 명령 워드 {op_1, op_2, op_3, op_4, …, op_n}에서 각 명령어 op들 상호간에 종속성(dependence)이 없어야만 목적하는 수행 결과를 얻을 수 있다. VLIW 시스템은 정적 스케쥴링(static scheduling)을 사용하므로, 컴파일러는 상호간에 종속성이 없는 명령어들만을 하나의 명령 워드로 묶을 수 있다. 만약, n(>=2)개의 프로세서가 있다면 각 명령 워드는 n개의 명령어로 구성될 것이다. 그러나, 대부분의 경우 단일프로세서를 위한 실행 프로그램에서 연속적으로 존재하는 n개의 명령어들 상호간에 종속성이 없을 것으로 기대하기는 어려우므로, 단순히 명령어들을 순차적으로 묶는 방법으로는 각 명령 워드에 n개의 명령어를 채울 수 없다. 따라서, 각 명령 워드에 채우지 못한 명령어 수 만큼의 nop(no operation)들을 대신 채우게 된다. 이러한 경우 상당 부분의 명령 워드를 nop들이 차지하게 되어 결과적으로 프로세서 이용율이 매우 낮아지게 된다. 이러한 문제의 해결을 위해 별도의 컴파일러 최적화 기법을 이용하여 보다 효율적인 명령 워드를 구성해야만 한다.

VLIW 시스템의 명령 워드상에서 프로세서 자원을 최대한 활용할 수 있는 효율적인 명령 워드 구성에 대한 문제는 전체 VLIW 시스템의 성능을 결정하는 중요한 요소이므로, VLIW 시스템과 관련한 연구들은 비교적 구현이 용이한 하드웨어 측면보다는 주로 컴파일러상에서의 최적화 기법들에 대한 연구들이 대부분을 차지한다. 효율적인 최적화를 위해 VLIW 시스템의 컴파일러는 트레이스 스케쥴링(trace scheduling) 등의 전역 스케쥴링(global scheduling)[5,4,10, 11,21,22] 또는 소프트웨어 파이프라이닝(software pipelining)[8,20,24,27,29,30] 등의 최적화 기법을 사용하여 nop가 들어갈 명령 워드 부분에 다른 유용한 명령어들을 이동시키거나 복사하는 작업을 수행한다. 전역 스케쥴링이란 기본 블록(basic block)의 경계를 넘어서고 루프가 없는 코드를 대상으로 하여 병렬화를 수행하는 기법이다. 소프트웨어 파이프라이닝이란 루프를 대상으로 수행되는 병렬화 기법으로서, 어떤 반복

1) VLIW 시스템에서의 프로세서는 문헌에 따라 functional unit과 동일한 의미로 사용되거나 또는 여러 functional unit들의 집합을 의미하기도 한다. 본 고에서는 전자를 따르기로 한다.

표 1 다중프로세서, 벡터처리기, VLIW 시스템의 비교

	다중프로세서 (multiprocessor)	벡터 프로세서 (vector processor)	VLIW
제어흐름	- 각 프로세서는 각자의 제어흐름을 가짐 - 프로세서들은 동기적으로 작동	- 모든 프로세서들이 한 번에 하나의 동일한 명령을 수행	- 모든 프로세서는 같은 주소에서 명령들을 읽지만 명령은 각 프로세서마다 다름 - 매우 긴 명령을 사용한 폰노이만식 제어흐름 사용
프로세서간 통 신	- 하드웨어에 의한 스케쥴링	- 상호연결망과 알고리즘의 규칙성에 의거하여 sub-vector들이 이동함 그렇지 않을 경우 단일처리기와 동일	- 각각의 데이터에 대한 이동은 컴파일시 확정됨 - 실행시 자원 할당작업은 없음
메모리 참 조	- 가능하면 지역(local) 메모리를 참조, 아니면 공유메모리(저속도, 고가격) 참조	- 벡터화된 참조는 모든 메모리 bank 사용가능 - 나머지 참조는 단일처리기에서와 동일	- 모든 메모리 참조는 컴파일시 예측되어 최적화함
프로그래밍 요구 사항	- 상호간의 통신과 동기화 오버헤드의 최소화를 위해 비교적 독립적으로 분리 수행됨	- 프로그램이 정형화되어 하드웨어 사양과 조화를 이루어야 함	- 컴파일러가 실행시의 상황을 예측하여 자원할당 - 프로그래밍 요구사항 최소화

의 수행이 종결되기 전에 다음 반복을 수행하도록 함으로써 반복들을 상호 중첩시키고 이 결과 수행 속도를 최대한 증진시키고자 하는데 그 목표가 있다. 이와 같은 VLIW 시스템에서의 최적화 기법들은 최근에 널리 보급되기 시작한 슈퍼스칼라 프로세서 등 여러 분야에 활용될 수 있으므로 현재 이러한 관련 연구들이 활발하게 진행되고 있다.

본 고에서는 VLIW 시스템의 컴파일러에서 사용되는 컴파일 최적화 기법들을 소개하고자 한다. 본 고의 2장에서는 트레이스 스케쥴링 등을 포함한 전역 스케쥴링 기법을 설명한다. 3장에서는 소프트웨어 파이프라이닝을 설명하며, 끝으로 4장에서 결론을 맺는다.

2. 전역 스케쥴링

전역 스케쥴링이란 기본 블록(basic block)의 경계를 넘어서고 루프가 없는 코드를 대상으로 하여 병렬화를 수행하는 기법이다. 전역 스케쥴링 기법은 주어진 제어 흐름 그래프(control flow graph)에서 상호 독립적인 명령어 집단(group)[2]을 생성해야 한다는 동일한 문제를 안고 있다. 대부분의 전역 스케쥴링 기법들은 이러한 문제를 서로 유사한 방법을 이용하여 해결하고 있는데, 먼저 어떤 집단에 포함시킬 수 있는 모든 명령어들의 가용 집합(availability set)을 계산한 다음, 경험적 방법을 기초로 하여 가용 집합으로부터 최선의 명령어를 선택한 후 선택된 명령어를 그 집단에 실제로 포함시키는 작업을 수행한다. 본 장에서는 전역 스케쥴링 기법들 중 트레이스 스케쥴링, 슈퍼블록(superblock) 스케쥴링, 하이퍼블록(hyperblock) 스케쥴링, 센티넬(sentinel) 스케쥴링 기법들을 기술한다.

2) 각 집단내의 모든 명령어들은 명시된 자원 제약 조건하에서 동시에 수행될 수 있다.

2.1 트레이스 스케줄링(trace scheduling)

트레이스 스케줄링[11] 기법은 프로그램상에서 자주 수행되는 트레이스(trace)를 인식하여 스케줄링하는 방법으로 여기서의 트레이스란 그림 2에서 볼 수 있듯이 프로그램의 흐름 그래프상에서 수행될 프로그램 경로이다.

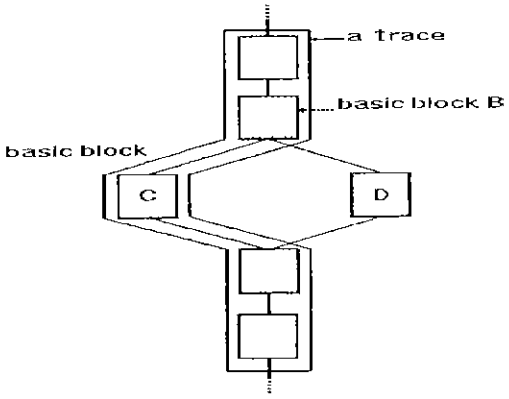


그림 2 트레이스의 예

그림 3에 트레이스 스케줄링의 예를 제시하였다. 가장 먼저 선택된 트레이스인 1st trace에 존재하는 명령어들간의 종속성을 고려하여 최대한 가능한 수의 명령어들을 각 명령 워드에 채운다.

그림 3의 예에서, 'LOAD A'와 'LOAD B'는 병렬 수행이 가능하므로 첫번째 명령 워드에 채워 넣는다. 그러나 이들 명령어들과 'ADD A B'는 A, B의 자료 종속 관계 때문에 병렬 수행이 가능하지 않으므로 다른 명령 워드에 채워진다. 이러한 과정을 통하여 같은 명령 워드에 채워진 명령어들은 VLIW 시스템상에서 서로 다른 프로세서들에 의해 독립적으로 수행된다. 일단 1st trace에 대한 명령 워드 구성이 종료되면 그 다음 트레이스인 2nd trace에 대한 명령 워드를 구성하되 1st trace가 구성한 명령 워드들 중 채워지지 못한 부분들을 최대한 활용할 수 있도록 구성한다. Ellis는 이러한 트레이스 스케줄링 기법을 체계화하여 실제로 Bulldog이라는 컴파일러를 구현하였고[10], 그외 많은 연구들이 진행되었다[5,4,21,22].

Ellis에 의해 체계화되고, 구현된 Bulldog 컴

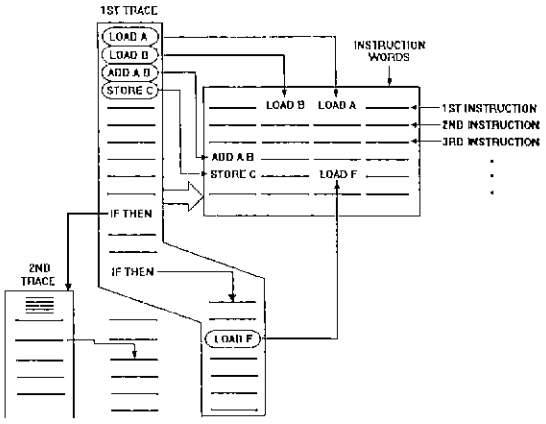


그림 3 트레이스 스케줄링에 의한 분기 명령어의 병렬처리

파일러의 구성도는 그림 4와 같다. Bulldog에서는 원시 언어(source language)를 전형적인 중간 코드(intermediate code) 형태로 구문 분석하고 표준 포트란 컴파일러의 최적화 과정을 통하여 최적화된 중간 코드를 생성한다. 그 후에 메모리 뱅크 참조 모호성 제거기(memory-bank disambiguator)는 벡터 형태로 참조(reference)되는 모든 뱅크들을 결정하고, 그 결과를 트레이스 스케줄러와 모호성 제거기로 보낸다. 여기서 트레이스 스케줄러는 프로그램의 흐름 그래프로부터 반복적으로 트레이스를 찾아내어 코드 생성기(code gernerator)에 그 결과를 보내고, 트레이스는 생성된 기계 코드로 대치된다. 여기서 코드를 생성하는 동안 코드 생성기는 모호성 제거기에게 트레이스에서 어떤 벡터 참조가 같은 메모리 위치를 참조할 수 있는가를 알려주도록 요구한다. Bulldog에서는 기존의 전형적인 컴파일러와는 다르게 구성요소 사이의 인터페이스가 매우 간단하고 그 수행이 절차적으로 진행된다.

Ellis의 Bulldog 컴파일러에서의 트레이스 스케줄링 과정은 다음과 같이 단계별로 진행된다.

(단계 1) 수행할 가능성이 가장 높은 트레이스를 인식하여, 그 트레이스에서 명령 워드를 컴팩션(compaction)[3]한다. 이러한 경로를 인식하고 그 트레이스하에서 병렬 수

3) 명령어들을 명령 워드에 가능한 빈 공간이 없도록 채우는 작업

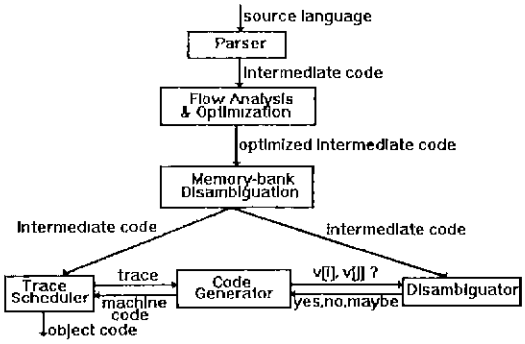


그림 4 Bulldog 컴파일러 구성도

행이 가능한 문장들을 명령 워드에 컴팩션한다. 이 과정에서 트레이스를 어떻게 추출, 선택할 것인가 하는 부분이 주요 연구대상이 되고 있다. 이를 위해 고려되는 접근방안들을 제시하면 다음과 같다.

(1) 사용자가 제공하는 컴파일러 지시자(compiler directive)를 이용하는 방안: if-then-else 구조와 같은 조건 분기문을 이용할 경우 실행 확률이 높은 부분을 then 이후나 else 이후에 넣는 것으로 약속하거나 확률 정보를 제공하는 컴파일러 지시자를 이용하여 요구하는 정보를 넣는 방식

(2) 경험(heuristic)을 이용하여 처리하는 방안: 구조화된 프로그램(structured program)에서는 조건 분기문이 수행될 경우, 역방향 분기(backward branch) 명령문이 존재하면 그 방향으로 수행이 진행될 확률이 거의 모든 경우이고, 순방향 분기(forward branch) 명령문이 존재하면 그 방향으로 수행이 진행될 확률이 50%라는 프로그램 작성의 특성을 이용하는 방안

(3) 프로그램의 실제 수행을 통하여 얻어진 통계적 자료를 이용한 방안: 실제 프로그램을 반복적으로 실행하여 수행시간 비교 및 분기 명령어의 분기결과 값을 이용하는 방안 이러한 방안들을 고려하여 트레이스를 추출하고 그 트레이스내에서 병렬 수행이 가능한 명령어들을 가능한 적은 수의 명령 워드들로 컴팩션한다.

(단계 2) 만약에 선택되지 않은 다른 트레이스들이 실제로 수행될 경우를 위해 원래 프로그램의 의미를 복구할 수 있도록 보상코드(compensation code)를 프로그램의 진입(entry)부분과 출구(exit) 부분에 추가한다. 여기서는 보상 코드를 적절하게 삽입하는 방안이 주요 연구대상이 되고 있는데, 고려되는 보상 코드의 내용은 다음 두가지가 있다.

(1) 한 트레이스내에서 다른 트레이스로 조건분기하는 문장들에 대한 보상코드: 그림 5는 조건 분기문을 포함하는 트레이스의 예이다. 여기서 1, 2, 3 문장은 트레이스 A내에 있기 때문에 분기의 발생여부에 관계없이 트레이스 B보다 먼저 수행된다. 그런데 분기가 발생될 경우, 5번 문장은 트레이스 A의 3번 문장에서 새로이 정의한 d를 이용하는데, 이렇게 되면 잘못된 값을 생성하게 된다. 따라서 이 경우 트레이스 A의 3번 문장을 무효화하는 작업이 필요한데, 이러한 작업을 수행하는 보상 코드(예. d:=a+3)를 다음과 같이 트레이스 B의 4번 문장 이전에 삽입한다.

보상코드 d:=a+3
4 f:=a*3
5 g:=d+2

(2) 다른 트레이스로부터 분기 유입으로 인한 보상코드 : 그림 6의 a와 같은 경우는 한 트레이스내로 분기가 유입되는 경우이다. 여기서 3번 문장은 분기의 유입에 관계없이 1번 문장에서 정의한 값을 사용하므로 옳은 결과를 갖지 못한다. 따라서 그림 6의 b와 같이 보상코드를 삽입할 수 있다. 이렇게 하여 트레이스의 결정으로 인한 부작용(side effect)을 제거할 수 있다. 이러한 보상 코드가 삽입되면, 예상하였던 트레이스대로 수행되지 않았을 때, 보상 코드의 내용을 수행하여 원하는 트레이스로 수행을 복구할 수 있다.

(단계 3) 모든 트레이스들이 스케쥴링될 때까지 앞의 두 단계를 반복한다. 여기서 스케쥴링의 목적은 보다 많은 명령어들을 병렬로 실행 가능하게 하는 것이다. 따라서 한

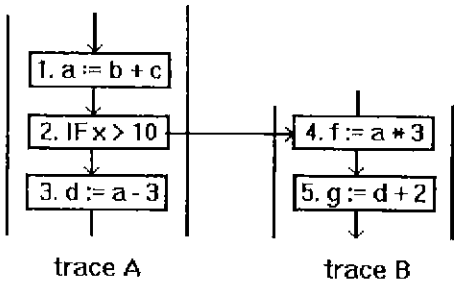


그림 5 트레이스의 예

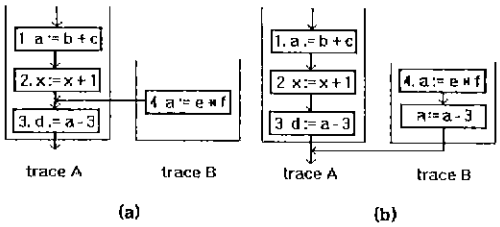


그림 6 트레이스의 예

트레이스내의 명령어들간에 종속성이 존재하지 않는다면 병렬로 수행 가능하기 때문에 메모리의 한 명령 워드 영역내에 배치 가능하고 이렇게 한 명령 워드내에 배치된 부분 명령어들은 여러 프로세서에 의해 병렬로 수행 가능하다.

이러한 트레이스 스케쥴링 기법에서는 전체 트레이스를 위한 모든 연산들이 연관성 있게 스케쥴링되며 모든 코드들에 대해 수행의미(execution semeantics)의 변경없이 정당한 코드이동(code motion)이 허용된다. 따라서 트레이스내의 모든 연산들에 대해 공통부분 수식제거(common subexpression elimination)같은 코드 최적화 형태를 적용할 수 있다. 그러나, 자료종속(data dependent)적인 조건문들에 대해서는 매우 복잡한 분석을 필요로 하며, 보상 코드가 매우 커질 수 있다.

2.2 슈퍼블록 스케쥴링 (superblock scheduling)

Chang에 의해 제시된 슈퍼블록 스케쥴링[4] 기법은 여러 개의 트레이스들로 구성된 슈퍼블록을 인식하여 스케쥴링하는 방법으로서, 트레이스내 기본 블록의 수가 일반적으로 적다는 단점을 보완하여, 프로그램으로부터 더 많은 병렬성을 추출·활용할 수 있도록 해주는 기법이다. 각각 한 개의 진입점과 출구를 갖고 있는 트레이스와는 달리 슈퍼블록은 한 개의 진입점과 여러개의 출구를 갖고 있는 것으로서, 하나의 슈퍼블록이 수행되면 이 슈퍼블록내에 있는 모든 기본 블록들은 수행될 가능성이 매우 높다.

슈퍼블록은 다음과 같은 두 단계를 거쳐 형성된다. 첫번째는 프로그램의 수행분석 도구(profiler)를 이용하여 트레이스를 구하는 단계이다. 두번째는 테일 중복(tail duplication)이라는 기법을 적용하는 단계로서, 기본 블록을 중복시켜 트레이스의 중간으로 유입되는 분기 흐름을 제거한다. 예를 들어 그림 7을 살펴보자. (a)는 수행 분석 도구를 이용하여 선택된 트레이스를 보여 주고 있다. 각 기본 블럭내의 숫자는 수행 빈도수를 표시하고 있다. (a)에서 B는 D로 분기하기 보다는 C로 분기되어 수행될 확률이 크므로 B, C, E를 하나의 트레이스로 형성한다. 이런 식으로 트레이스를 선택한 후에는 테일 중복 기법을 적용한다. (a)에서는 D가 트레이스 (B,C,E)의 중간으로 분기하고 있는데, 이런 분기를 제거하기 위해 (b)에서 보이는 것처럼 E와 동일한 E′를 생성한 후 E대신 E′로 분기방향을 조정한다. 이 결과로서 슈퍼블록 (B,C,E)와 (D,E′)가 생성되었다.

2.3 하이퍼블록 스케쥴링 (hyperblock scheduling)

하이퍼블록 스케쥴링[21] 기법은 여러개의 술어화된 기본 블록(predicated basic block)들로 구성된 하이퍼블록[4]을 인식하여 스케쥴링하는 방법이다. 여기서 술어화된 기본 블록은 제어종속되는 명령어를 자료 종속되는 명령어로 변환하여 여러 기본 블록들을 한 개의 기본 블록으로 형성한 것이다. 제어 종속되는 명령어는 If 변환(If-conversion)[16,23] 기법을 통해 술어화된 명령어(predicated instruction)로 변환되고,

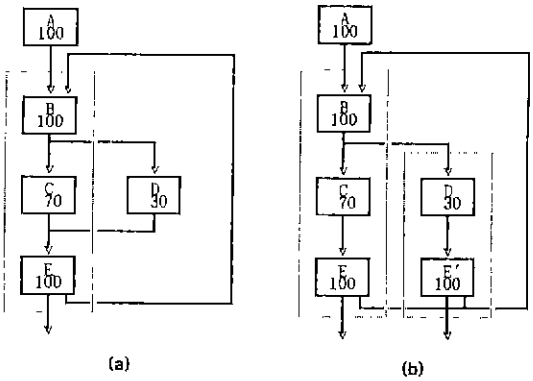


그림 7 슈퍼블록의 형성 단계

이 변환된 명령어는 조건 레지스터(condition register)에 저장된 값을 피연산자로 사용하여 수행된다. 이러한 하이퍼블록은 한 개의 진입점과 여러 개의 출구들을 갖고 있다는 점에서 슈퍼블록과 비슷하나, 기본 블록을 사용하느냐 아니면 술어화된 기본 블록을 사용하느냐에 따라 구분된다.

2.4 센티널 스케쥴링(sentinel scheduling)

하이퍼블록 스케쥴링과 마찬가지로 Mahlke에 의해 제시된 센티널 스케쥴링[22] 기법은 사변 명령어(speculative instruction)를 수행할 때 발생할 가능성이 있는 예외(exception) 경우를 정확히 탐지하고 이를 보고할 수 있는 방안을 제시하고 있다. 즉, 예외를 야기할 수 있는 각 사변 명령어에 대해서 센티널이라는 명령어를 원래 기본 블록내에 두고, 센티널로 하여금 사변 명령어에 의해 야기된 예외 경우를 보고하도록 한다. 이러한 센티널은 원래 기본 블록내에 존재하는 명령어일 수도 있고 아니면 새로이 생성된 명령어일 수도 있다. 이를 위해 센티널 스케쥴링 기법은 효율적인 구조로 알려져 있는 슈퍼블록을 기초로 하여 스케쥴링을 수행한다.

3. 소프트웨어 파이프라이닝

모든 응용 프로그램은 루프(loop)를 수행하는데 그 수행 시간의 대부분을 차지하므로, 최적화 컴파일러를 설계하는데 있어서 주요 관심은 루프내에 존재하는 명령어 수준의 병렬성(Instruction Level Parallelism: 이하 ILP와 혼용)을 어떻게 잘 활용할 것인가에 있다. 루프내 ILP를 활용하기 위한 기존의 접근 방법으로서 루프 전개(loop unrolling) 기법이 있는데, 루프를 여러번 전개하고 전개된 명령어들을 대상으로 스케쥴링을 수행한다. 그러나 이 기법은 서로 다른 반복(iteration)에 속해 있는 명령어들 사이의 병렬성을 충분히 활용하지 못하기 때문에[9], 각 반복의 마지막과 다음 반복의 시작 사이에는 약간의 지연시간이 존재하게 된다. 이러한 지연 문제를 해결하기 위해 소프트웨어 파이프라이닝 기법이 도입되었다[30].

소프트웨어 파이프라이닝 기법은 강력한 스케쥴링 기법으로서, 명령어 수준에서 루프를 효율적으로 최적화한다. 이 기법은 자원(resource)과 종속성(dependence)이라는 제약 조건상에서 각 반복이 수행 완료되기 전에 다음 반복을 실행하도록 루프 몸체(loop body)를 재구성한다. 이 결과 각 반복은 다른 반복과 중첩 실행되어 루프의 실행 속도가 크게 증진된다[27].

소프트웨어 파이프라이닝 기법의 예를 살펴보면 다음과 같다. 그림 8(a)는 연산 f를 위해 r0:=f(r0)를 반복적으로 계산하고 r0이 영(zero)보다 작을 때, r0를 출력하는 예제 코드이다. 그림 8(b)는 그림 8(a)를 병렬화한 코드를 보여주고 있다. 여기서 각 그룹은 각 주기(cycle)에서 독립적으로 수행될 수 있는 명령어들을 나타낸다. 첫번째 주기에서는 r0:=f(r0)이 수행된다. 두번째 주기에서는 r0가 0과 비교되는 동안, 두번째 반복의 f(r0) 값이 임시적으로 r1에 저장된다. 세번째 주기에서는 조건 분기문인 IF cc0이 먼저 수행되는데, 분기가 수행되면 두번째 반복을 위해 r1이 r0에 복사되고, 0과 비교된다. 이와 동시에 세번째 반복의 f(r0) 값을 임시적으로 저장하는 r1:=f(r0)가 수행된다. 그리고 제어는 네번째 주기에서 세번째 그룹으로 되돌아가고, 동일한 수행이 반복된다. 세번째 주기와 같은 상황이 n번째 주기까지 계속적으로 이어지는데, 이런 식으로 일정한 유형을 보이는 부분을 커널(kernel)이라 한다. 이 예에서는 각 반복이 매 1주기마다

```
L1 : r0  = f(r0)
     cc0 := r0 < 0
     IF cc0 GOTO L2
L2   printf(r0)
```

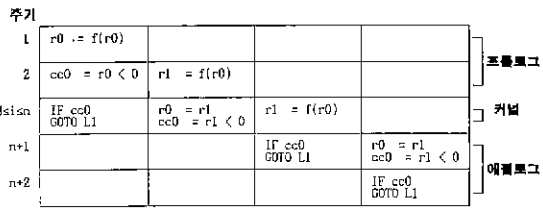
(a) 예제 프로그램 부분

```
     r0  := f(r0)

     cc0 := r0 < 0
     r1  := f(r0)

L1   IF cc0 GOTO L2
     r0  := r1
     cc0 := r1 <0
     r1  = f(r1)
     GOTO L1
L2   printf(r0)
```

(b) 병렬화된 코드 형태



(c) 소프트웨어 파이프라인화된 코드 형태

그림 8 프로그램의 소프트웨어 파이프라이닝 예

초기화되는데, 이 값을 초기화 간격(Initiation Interval: 이하 II와 혼용)이라 한다. 커널에서는 각 반복이 매 II주기마다 수행 완료된다. 또한 커널 전에 있는 부분은 2주기의 수행 시간을 보이는데, 이 부분은 프롤로그(prologue)라 불리운다. 마찬가지로 커널 후에 있는 부분을 에필로그(epilogue)라 하는데, 나머지 반복들을 수행 완료하는데 필요하다[30].

소프트웨어 파이프라이닝 기법을 적용함으로써 얻어지는 이점을 보이기 위해 이 기법을 적용하기 전과 후의 실행 시간을 비교해 보자. 100번의 반복을 수행한다고 할 때, 중첩 수행을 하지 않았을 경우에는 각 반복마다 3주기가 필요하므로 총 300주기의 수행 시간이 필요하다. 중첩 수행하는 경우에는 프롤로그와 에필로그를 수행하는데에 각각 2주기, 커널을 수행하는데에 98주기가 필요하므로 총 102주기의 수행 시간이 필요하다. 중첩 수행을 적용하지 않았을 경우와 비교해 볼 때, 2.94배의 수행 시간이 향상되었음을 알 수 있다.

이러한 소프트웨어 파이프라이닝 기법은 지역 스케줄링 기법을 사용하느냐 또는 전역 스케줄링 기법을 사용하느냐에 따라 크게 두 가지로 구분된다[30]. 조건 분기가 없는 루프를 대상으로 스케줄링하는 소프트웨어 파이프라닝을 지역 소프트웨어 파이프라이닝(local software pipelining)이라 하고, 조건 분기가 있는 루프를 대상으로 스케줄링하는 소프트웨어 파이프라이닝을 전역 소프트웨어 파이프라이닝(global software pipelining)이라 한다[27].

3.1 지역 소프트웨어 파이프라이닝

소프트웨어 파이프라이닝에 관한 초기 연구들은 조건 분기가 없는 루프만을 대상으로 하였다. 이들은 자원과 자료 종속성을 기초로 하여 최소 II를 구한 후 루프를 소프트웨어 파이프라인화 하였다. 본 절에서는 이러한 연구 결과중 모듈로 스케줄링(modulo scheduling) 기법과 분해된 소프트웨어 파이프라이닝(decomposed software pipelining) 기법에 대해서 기술한다.

3.1.1 모듈로 스케줄링(modulo scheduling)

모듈로 스케줄링[4][8,20,24,29] 기법은 조건 분기가 없는 루프만을 대상으로 적용되는 것으로서, 조건 분기와 종속성 싸이클이 없는 루프에 대해서는 최적의 스케줄을 생성한다. 이 기법의 첫 단계는 자료 종속성 그래프(data dependence graph)를 구성한 후 자원(resource) 및 종속성 사이클(dependence cycle)을 근거로 하여 II의 하한값을 결정한다. 다음 단계는 자원의 사용상태와 타이밍을 표현하고 있는 MRT[6]를 이용하여 스케줄링 기법을 적용함으로써 대상 루프를 소프트웨어 파이프라인화한다.

예를 들어, 부동 소수점 처리기라는 자원이 2개 있고, 이 2개의 처리기를 루프내 5개의 명령어들이 사용한다고 가정하면, 각 반복은 이전 반복이 수행된 후 ⌈5/2⌉=3주기 안에는 수행될 수 없다. 즉, 부동 소수점 처리기라는 자원으로 인해 최소 II는 3이 된다. 또한 II는 종속성 사이클에 의해서도 하한값이 결정된다. 종속성 사이클은

4) 하이퍼블록은 블록 선택(block selection), 테일 중복(tail duplication), 루프 필링(loop peeling), 노드 분할 (node splitting), If 변환(If-conversion) 등과 같은 기법들을 적용함으로써 구성된다[21].

5) 순환 스케줄링(cyclic scheduling)[9]이라고도 한다.

6) 모듈로 자원 테이블(Modulo Resource Table: MRT)은 자원의 사용상태를 주기 단위로 표현하기 위해 사용된다.

어떤 반복에서의 명령어 I가 여러 반복 이전의 자기 자신 I로 종속될 때 존재한다. 종속성 사이클 $I_1 \rightarrow I_2 \rightarrow \cdots \rightarrow I_k \rightarrow I_1$($a \rightarrow b$는 b가 a에 종속됨을 표현한다)에서, 첫번째 I_1과 마지막 I_1 사이에는 20만큼의 지연시간과 5만큼의 반복이 존재한다고 가정하자. 이때 하나의 반복은 이전 반복이 수행된 후 $\lceil 20/5 \rceil = 4$주기 안에는 수행될 수 없다. 즉, 종속성 사이클로 인한 최소 II는 4가 된다. 이런 식으로 모든 자원과 모든 종속성 사이클 각각에 대해 최소 II를 구한 후, 이들 값중 가장 최대값을 최종 II로 결정한다.

그림 9은 종속성 싸이클이 없는 루프에 대해 모듈로 스케쥴링을 적용하는 과정을 보여 주고 있다. 그림 9(c)의 자료 종속성 그래프에서 에지는 종속성의 종류(f: 흐름 종속성, a: 반 종속성)와 지연 시간을 보여 주고 있다. 이 예에서는 종속성 싸이클이 존재하지 않으므로 최소 II는 자원 제약 조건에 의해 결정된다. 사용 가능한 자원(FU)의 수가 2개이고 루프내 명령어들이 4개 있으므로, 최소 II는 $\lceil 4/2 \rceil = 2$이다. 이렇게 최소 II를 구한 후 MRT를 이용하여 리스트 스케쥴링(list scheduling)[15] 기법을 적용함으로써 각 명령어들을 스케쥴링한다. LD와 ADD는 각각 주기 0과 4에서 스케쥴링될 수 있고 (0 mod II)=(4 mode II)=0 이므로 MRT의 주기 0에 해당하는 행에 두 명령어를 채운다. 마찬가지로 MUL은 주기 5에서 MRT의 주기 1에 해당하는 행에 이 명령어를 채운다. ST는 주기 6에서 스케쥴링 되어야 하지만 (6 mod II)=0에 해당하는 MRT 행에는 사용 가능한 자원이 없으므로 주기 7에서 스케쥴링되어야 한다. 이런 식으로 지역 스케쥴링을 수행한 후, 주어진 II에 대해서 가능한 스케쥴을 발견하지 못할 경우에는 II의 값을 하나 증가시켜 다시 스케쥴링을 수행한다[9].

주어진 II에 대해 스케쥴링을 수행한 후 소프트웨어 파이프라인화된 루프 코드를 생성한다. 앞서 언급하였듯이 소프트웨어 파이프라인은 프롤로그, 커널, 에필로그로 구성되어 있다. 그림 10(c)에서 볼 수 있듯이 프롤로그 부분은 주기 0부터 3까지이고 나머지는 커널과 에필로그를 구성한다. 그러나 그림 10(c)와 같이 여러 반복이 중첩됨에 따라 레지스터 수명(register lifetime)

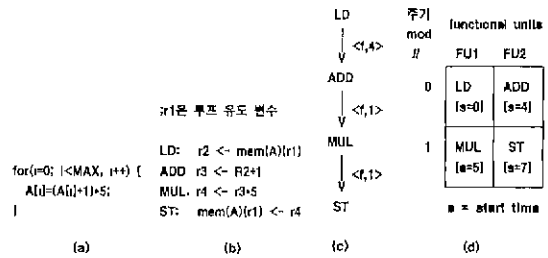


그림 9 자료 종속성이 없는 루프에 대한 모듈로 스케쥴링
(a) C코드 (b) 루프 제어문이 없는 어셈블리 코드
(c) 자료종속성 그래프 (d) MRT

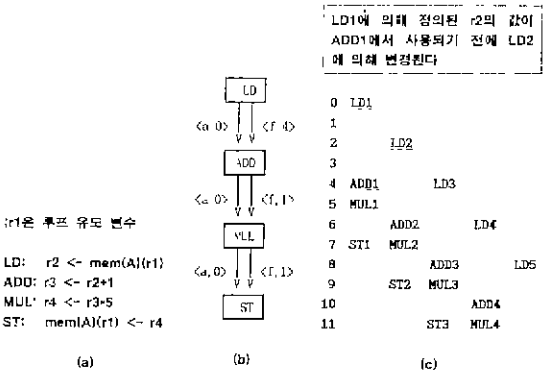


그림 10 소프트웨어 파이프라닝 후 레지스터 재명명의 필요성의 예
(a) 루프 제어문이 없는 어셈블리 코드 (b) 자료종속성 그래프 (c) 레지스터 재명명의 필요성을 보여주는 예

이 겹치기 때문에 원래 프로그램의 의미가 변경되어 버리는 현상이 발생한다[19,29]. 이를 해결하기 위해서는 수명이 겹치는 각각의 레지스터에 대해 재명명을 수행하여야 한다. 이 작업은 소프트웨어나 하드웨어에 의해 수행될 수 있다. 소프트웨어 기법에는 커널을 전개함으로써 재명명을 수행하는 모듈로 변수 확장[19](modulo variavle expansion) 기법이 있고, 하드웨어 기법에는 Cydra 5에서 사용된 것으로서 스택(stack)을 이용한 회전 레지스터 화일(rotating register file)[17,25] 기법이 있다.

3.1.2 분해된 소프트웨어 파이프라이닝(decomposed software pipelining)

분해된 소프트웨어 파이프라이닝(DEcomposed Software Pipelining: 이하 DESP와 혼용)[27] 기법은 모듈로 스케쥴링이 일반적인 경우에

있어서 최적의 스케쥴을 생성하지 못하는 원인을 규명하고, 이를 고려하여 일반적인 경우에 있어서도 최적의 스케쥴을 생성할 수 있는 새로운 소프트웨어 파이프라이닝 기법이다. 예를 들어 그림 11을 살펴보자. 루프내에 A, B, C, D, E와 같은 명령어들이 있다고 할 때, A, C, E는 2만큼의 지연 시간을, B, D는 1만큼의 지연 시간을 갖는다고 가정한다. (a)는 이에 대한 자료 종속성 그래프를 보여 주고 있고, (b)와 (c)는 예약 테이블(reservation table)[9]을 보여 주고 있다. 여기서 예약 테이블은 자원 사용 및 수행 시간을 표현하고 있는 것으로서, 수평 좌표는 주기 단위의 시간을 표시하고 있고, 수직 좌표는 사용가능한 자원을 표시하고 있다. 이때 최소 II의 값을 6이라 가정하고 모듈로 스케쥴링을 수행하면 그림 (b)과 같은 결과를 보이는데, 명령어 E가 MRT에 채워지지 않았으므로 실패한 스케쥴이다. 이런 경우 모듈로 스케쥴링 기법은 II의 값을 하나 증가시켜 다시 스케쥴링을 수행한다. 그러나 명령어 C의 스케쥴을 조정하면 그림 (c)처럼 II=6에서도 가능한 스케쥴이 존재하므로, 모듈로 스케쥴링 기법이 항상 최적의 스케쥴을 생성하는 것이 아님을 알 수 있다[9].

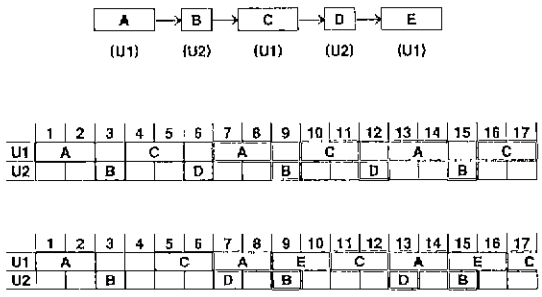


그림 11 모듈로 스케쥴링의 문제점을 보여 주는 예
(a) 자료 종속성 그래프
(b) 모듈로 스케쥴링후 실패한 스케쥴 (II=6)
(c) II=6일 때 가능한 스케쥴

이러한 문제점을 해결하기 위해 Eisenbeis[9]는 비순환 종속성 그래프(acyclic dependence graph)를 갖는 루프만을 대상으로 하여, 루프 스케쥴링 문제를 두 단계로 분리하고 있다. 첫번째 단계로 명령어 간의 종속성은 고려하지 않은 채 루프 유형(loop pattern)을 구성하였고, 두번째 단계로 모든 종속성이 만족되도록 반복 인덱스(iteration index)라 불리우는 정수값을 각 명령어에 결부시켰다. 또한 Gasperoni[31]는 자원이 무한하다는 가정하에 주어진 루프를 전처리하고, 전처리로부터 얻은 정보를 이용하여 자원 종속성 그래프가 비순환 그래프가 되도록 에지를 삭제한 후, 마지막으로 자원 제약 조건하에 리스트 스케쥴링 기법을 적용하여 루프 몸체를 재구성하였다. 이러한 두가지 접근 방법을 기초로 하고 있는 DESP[27] 기법은 소프트웨어 파이프라이닝을 일차원 벡터에서 이차원 행렬로의 변환으로 간주하여, 소프트웨어 파이프라이닝 문제를 두 개의 부문제로 분해하였다. 첫번째 문제는 행렬에 있는 명령어들의 행 번호(row number)를 결정하는 문제로서, 리스트 스케쥴링 기법을 적용함으로써 효과적으로 해결된다. 두번째 문제는 열 번호(column number)를 결정하는 문제로서, 기존의 그래프 알고리즘을 적용함으로써 쉽게 해결된다.

행 번호와 열 번호에 대한 개념을 설명하기 위한 예로서 그림 12를 살펴보자. (a)의 루프내에 있는 명령어들은 일차원 백터(1,2,3,4)로 간주할 수 있고, 새로 구성된 (c)의 루프 몸체는 다음과 같은 이차원 행렬로 간주할 수 있다.

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

이 행렬에서 각 행은 주기를 표현하고 각 열은 반복을 표현한다. 동일한 행 번호를 갖는 두 명령어는 동일한 주기에서 수행될 수는 있으나, 동일한 자원을 동시에 사용할 수 없음을 의미한다. 동일한 열 번호를 갖는 두 명령어는 동일한 반복에 속함을 의미하고, 이런 경우 두 명령어 사이에 종속성이 존재하는지를 고려해야 한다.

3.2 전역 소프트웨어 파이프라이닝

본 절에서는 조건 분기가 있는 루프를 대상으로 소프트웨어 파이프라이닝을 수행하는 여러가지 기법들을 기술한다. 이러한 기법들은 앞서 살펴본 지역 소프트웨어 파이프라이닝 기법과

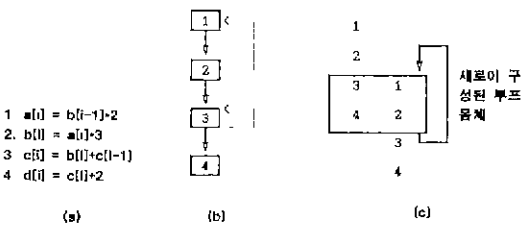


그림 12 DESP를 위한 예
(a) 루프 몸체 (b) 자료 종속성 그래프(c) 소프트웨어 파이프라인화된 루프

코드 컴팩션 기법을 이용하여 수행된다.

3.2.1 완전 파이프라이닝(perfect pipelining)

Aiken에 의해 제시된 완전 파이프라이닝 기법[1]은 반복적인 유형(repetitive pattern)이 발견될 때까지 루프를 컴팩션 전개시켜 나가는 기법이다. 이 기법의 첫번째 단계는 삼투 스케줄링(percolation scheduling) 기법의 코어 변환(core transformation)과 같은 컴팩션 연산들을 사용하여 루프내의 명령어들을 이동시킨다. 이 과정에서 불필요한 경쟁 상태(race condition)를 피하기 위해 자원 제약 조건을 적용시키지 않는다. 여기서 경쟁 상태란 하나의 명령어를 이동시킴으로써 다른 명령어를 이동시킬 수 없는 상태를 의미한다. 두번째 단계는 컴팩션된 루프를 전개하고 명령어들을 스케줄링한다. 즉, 반복 n에 있는 명령어들을 1과 n-1 사이에 있는 반복내로 최대한 끌어 올린다. 물론 이 과정에서 반복내(intraiteration) 또는 반복간(interiteration) 종속성을 만족시켜야 하며 자원 제약 조건을 침해해서는 안된다.

그림 13은 반복적인 유형을 발견하는 내용을 보여주고 있다. 파이프라인화된 메모리 접근 유닛과 가산기가 각각 한 개씩 존재하고, 파이프라인 단계가 각각 3과 1이라 가정하자. 이때 (b)는 (a)에 대한 예약 테이블로서, 주기 3과 8 사이에서 반복적인 유형을 발견할 수 있으며, 6주기당 2개의 반복을 수행시킬 수 있는 최적의 스케줄을 보이고 있다.

이러한 완전 파이프라이닝 기법의 장점은 거의 최적의 스케줄을 구할 수 있다는데 있다. 그러나 반복적인 유형을 발견하기 위해 최악의 경우 지수 증가적인 수행 시간이 필요하고, 또한 다중 통로 분기(multiway branch)[23]를 위한 부가적인 하드웨어 메카니즘이 필요하다는 단점을 갖고 있다.

3.2.1 완전 파이프라이닝(perfect pipelining)

Aiken에 의해 제시된 완전 파이프라이닝 기법[1]은 반복적인 유형(repetitive pattern)이 발견

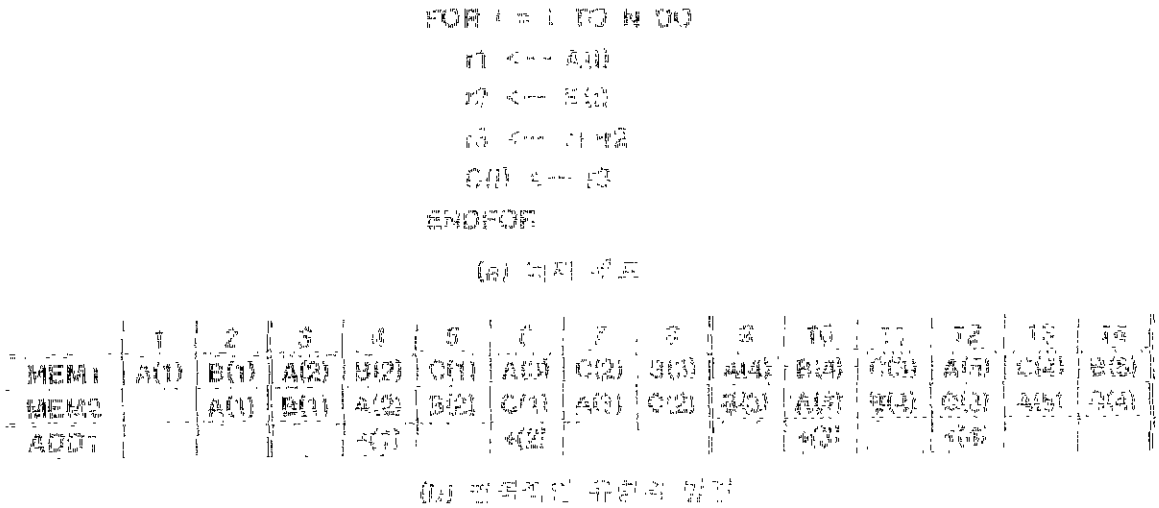


그림 13 완전 파이프라이닝을 위한 예

될 때까지 루프를 컴팩션 전개시켜 나가는 기법이다. 이 기법의 첫번째 단계는 삼투 스케쥴링(percolation scheduling) 기법의 코어 변환(core transformation)과 같은 컴팩션 연산들을 사용하여 루프내의 명령어들을 이동시킨다. 이 과정에서 불필요한 경쟁 상태(race condition)를 피하기 위해 자원 제약 조건을 적용시키지 않는다. 여기서 경쟁 상태란 하나의 명령어를 이동시킴으로써 다른 명령어를 이동시킬 수 없는 상태를 의미한다. 두번째 단계는 컴팩션된 루프를 전개하고 명령어들을 스케쥴링한다. 즉, 반복 n에 있는 명령어들을 1과 n−1 사이에 있는 반복내로 최대한 끌어 올린다. 물론 이 과정에서 반복내(intraiteration) 또는 반복간(interiteration) 종속성을 만족시켜야 하며 자원 제약 조건을 침해해서는 안된다.

그림 13은 반복적인 유형을 발견하는 내용을 보여주고 있다. 파이프라인화된 데모리 접근 유닛과 가산기가 각각 한 개씩 존재하고, 파이프라인 단계가 각각 3과 1이라 가정하자. 이때 (b)는 (a)에 대한 예약 테이블로서, 주기 3과 8 사이에서 반복적인 유형을 발견할 수 있으며, 6주기당 2개의 반복을 수행시킬 수 있는 최적의 스케쥴을 보이고 있다.

이러한 완전 파이프라이닝 기법의 장점은 거의 최적의 스케쥴을 구할 수 있다는데 있다. 그러나 반복적인 유형을 발견하기 위해 최악의 경우 지수 증가적인 수행 시간이 필요하고, 또한 다중 통로 분기(multiway branch)[23]를 위한 부가적인 하드웨어 메카니즘이 필요하다는 단점을 갖고 있다.

3.2.2 GURPR*

초기 소프트웨어 파이프라이닝 기법들은 하나의 기본 블록으로 구성된 루프를 대상으로 적용되었다. 이러한 기법중에 URCR(UnRolling, Compaction, and Rerolling) 기법은 트레이스 스케쥴링 기법을 개선시키기 위해 사용된 것으로서, 루프를 두 번 전개하여 명령어들을 컴팩션한 후 II를 구함으로써 새로운 루프 몸체를 구성한다. 보다 복잡한 루프가 소프트웨어 파이프라인화됨에 따라 개선된 알고리즘이 등장하게 되었다. 즉, 종속성 싸이클 문제를 해결하기 위해 URPR(UnRolling, Pipelining, and Rerolling) 기법이 사용되었고, 이의 개선된 형태로서 조건 분기를 해결하기 위한 GURPR(Global URPR) 기법이 등장하였다. 또한 보다 나은 수행 속도를 달성하기 위해 GURPR의 분기 처리 능력을 수정한 GURPR* [3,26]기법이 발표되었다.

GURPR*의 첫번째 단계에서는 완전 파이프라이닝에서 처럼 루프 몸체를 컴팩션하고, 다음 단계에서는 반복간 최대 종속성 거리(dependence distance)를 기준으로 하여 II를 구한다. 그런 후 연속된 반복들이 매 II주기마다 수행할 수 있도록 루프를 파이프라인화 한다. 명령어들은 컴팩션된 원래 루프에서의 순서와 동일하도록 유지되고, 또한 II는 종속성 싸이클을 배려하고 있기 때문에 파이프라이닝 단계에서는 종속성 문제를 고려할 필요가 없다. 만약 파이프라이닝 수행시 자원 충돌이 발생하면 그 명령어를 한 주기만큼 지연시키고, 이 결과 자원 충돌 현상이 여전히 존재하면 비어 있는 주기를 하나 삽입하도록 한다. 이러한 식으로 파이프라이닝 단계가 종결된 후에는 최소 II를 다시 구하기 위한 작업을 수행한다. 최종적으로 최소 II를 구한 후에는 불필요한 명령어들을 삭제하고 프롤로그, 커널, 에필로그를 구성한다.

이 기법의 장점은 첫째 부가적인 하드웨어 지원을 필요로 하지 않고, 둘째 파이프라이닝시 종속성을 고려할 필요가 없고, 세째 자원 제약 조건을 명시적으로 처리한다는데 있다. 또한 완전 파이프라이닝 기법처럼 반복적인 유형을 발견하기 위해 복잡한 작업을 수행하지 않으므로 시간 복잡도가 낮다는 장점이 있다. 이 기법의 단점으로는 여러가지 이유로 인해 병렬성이 희생된다는데 있다. 첫째 불필요한 명령어들 각각에 대해서도 자원이 사용되기 때문에 최적의 커널을 구할 수 없고, 둘째 파이프라이닝 단계에서는 종속성 분석을 하지 않기 때문에 종속성에 대해서는 최악의 가정을 해야 한다.

3.2.3 계층적 축소(hierarchical reduction)

계층적 축소[19] 기법은 루프내 각각의 조건 구조를 하나의 명령어 형태(reduct-op)로 표현

함으로써 조건 분기를 갖는 코드를 직선 코드(straight-line code)로 변환시킨다. 계층적 축소를 한 후 각 reduct_op 명령어는 루프내 다른 명령어들과 함께 스케쥴링된다. 계층적 축소 기법은 별도의 하드웨어를 사용하지 않으므로 모듈로 스케쥴링 후에는 조건 분기문을 다시 삽입함으로써 코드를 재생성해야 한다.

3.2.4 강화된 모듈로 스케쥴링(enhanced modulo scheduling)

강화된 모듈로 스케쥴링(Enhanced Modulo Scheduling: 이하 EMS와 혼용)[28] 기법은 조건 분기를 갖고 있는 루프를 대상으로 하여 최소 II를 구하는 기법이다. 이를 위한 첫번째 단계로서 If 변환[16,23] 기법을 루프 몸체에 적용하여 술어화된 코드(predicated code)를 생성한다. 계층적 축소와는 달리 If 변환을 적용하면 조건문 안의 명령어와 조건문 밖의 명령어를 함께 고려하여 스케쥴링할 수 있기 때문에, 소프트웨어 파이프라이닝 기법보다 더 나은 II의 하한값을 구할 수 있게 된다. 이런 식으로 술어화된 코드를 대상으로 자료 종속성 그래프를 구성한 후 앞서 언급한 모듈로 스케쥴링 기법을 적용한다. 마지막 단계로서 EMS는 조건부 실행(conditional execution)을 지원하지 않는 하드웨어를 대상으로 하고 있기 때문에, 계층적 축소 기법과 마찬가지로 모듈로 스케쥴링 후에는 조건 분기문을 다시 삽입함으로써 코드를 재생성해야 한다. 이렇게 조건 분기문을 다시 삽입하는 것을 역 If 변환(reverse If-conversion)[30] 기법이라 한다.

3.2.5 파이프라인 스케쥴링 (pipeline scheduling)

Ebcioglu에 의해 제안된 파이프라인 스케쥴링[6]은 알고리즘의 초기 단계에서 유형(pattern)을 찾을 수 있도록 항상 보장하는 기법이다. 현재 반복에서 조건 분기의 두 실행 경로는 이후 반복에서 두 경로의 초기화율이 구분되거나, 구분되지 않더라도 독립적으로 검사된다. VLIW에서의 한 유형의 연산들은 각각 독립적으로 진행되지 않고 전체가 하나로 진행된다. 이러한 이유 때문에 이 기법에서는 최적의 결과를 얻을 수 없다.

3.2.6 강화된 파이프라인 스케쥴링(enhanced pipeline scheduling)

강화된 파이프라이닝 스케쥴링(Enhanced Pipeline Scheduling: 이하 EPS와 혼용) 기법[7]은 각 명령어를 트리의 에지에 표현한 일반 모델을 확장한 것으로서, 하나의 노드로 표현된 각 이진 트리(binray tree)는 종속성을 침해하지 않고 수행될 수 있는 모든 명령어들을 포함하고 있다. 트리의 어느 경로를 따라서 수행할 것인가에 관한 조건들이 정의되기 때문에 다중통로 분기 뿐만 아니라 조건부 실행과 같은 하드웨어 지원이 필요하다. EPS 기법의 첫 단계는 삼투 스케쥴링(percorlation scheduling)[2], 선택 스케쥴링(selective scheduling)[23], 영역 스케쥴링(region scheduling)[13]과 같은 전역 코드 컴팩션 기법들을 이용하여 루프내 명령어들을 컴팩션한다. 다음 단계는 각 주기마다 새로운 반복이 수행될 수 있도록 알고리즘을 수행함으로써 루프를 재구성한다. 이 기법은 알고리즘의 수행 도중 어느 때 중지시켜도 정당한 코드를 생성할 수 있다는 장점을 갖으며, 알고리즘의 모든 단계가 방향성 비순환 그래프를 이용하므로 전역 스케쥴링 기법과 결합되어 사용될 수 있다는 장점을 갖는다.

4. 결 론

본 고에서는 VLIW 시스템에서의 컴파일러 최적화 기법으로 사용되는 전역 스케쥴링 기법과 소프트웨어 파이프라이닝 기법을 소개하였다. 전역 스케쥴링 기법은 루프가 없는 코드를 대상으로 하여 명령어 수준에서의 병렬성을 추출 활용하는 병렬화 기법이며, 소프트웨어 파이프라이닝 기법은 루프를 대상으로 하여 여러 개의 반복들이 중첩 수행될 수 있도록 루프를 재구성하는 루프 변환 기법이다.

VLIW 시스템이 여러면에서 많은 장점을 가지고 있다는 점을 다시 강조할 필요는 없겠지만, 90년대를 전후하여 상업용 마이크로프로세서를 기반으로 하는 병렬 컴퓨터의 연구개발이 본격화됨에 따라, 앞으로 VLIW 시스템을 포함한 다른

형태의 컴퓨터들을 우리가 사용하게 될 수 있을지는 불분명하다. 그러나, VLIW 시스템에서의 컴파일러 최적화 기법들은 명령어 수준에서의 병렬성을 최대한 활용하기 위한 기법들로, VLIW 시스템 뿐 아니라 슈퍼스칼라 기능을 가진 마이크로프로세서에의 코드 최적화에도 동일하게 적용될 수 있음을 주목할 필요가 있다[7]. 최상의 병렬성은 명령어 수준에서 얻을 수 있음을 고려한다면 병렬처리 분야에서 이러한 컴파일러 최적화 기법들에 대한 더욱 많은 연구는 물론 보다 실용적인 최적화 컴파일러가 개발되어야 할 것이다.

참고문헌

[1] Aiken, A. and Nicolau, A., "Perfect Pipelining", *Proceedings of the second European Symposium on Programming*, pp. 221~235, June 1988.

[2] Aiken, A. and Nicolau, A., "A Development Environment for Horizontal Microcode", *IEEE Transactions on Software Engineering*, Vol.14, No.5, pp. 584~594, May 1988.

[3] Bockhaus, J. W., *An Implementation of GURPR*: A Software Pipelining Algorithm*, MS thesis, Illinois University, 1992.

[4] Chang, P. P., Mahlke, S. A., Chen, W. Y., Warter, N. J. and Hwu, W. W., "IMPACT: An Architectural Framework for Multiple Instruction-Issue Processors", *The 18th Annual International Symposium on Computer Architecture Conference Proceedings*, pp. 266~275, May 1991.

[5] Colwell. R. P., Nix, R. P., O'Dnnell, J. J., Papworth. D. B. and Rodman, P. K., "A VLIW Architecture for a Trace Scheduling Compiler", *IEEE Transactions on Computers*, Vol. 37, No. 8, pp. 967~979, August 1988.

[6] Ebcioğlu K., "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps", *Proceedings of the 20th Annual Workshop on Microprogramming*(Micro-20), pp. 69~79, 1987.

[7] Ebcioğlu, K. and Nakatani, T., "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW architecture", *Languages and Compilers for Parallel Computing*, pp. 213~229, 1989.

[8] Eisenbeis, C., "Optimization of horizontal microcode generation for loop structures", *International Conference on Supercomputing*, pp. 453~465, July 1988.

[9] Eisenbeis, C. and Windheiser, D., *A New Class of Algorithms for Software Pipelining with Resource Constraints*, Rapport de Recherche INRIA, INRIA, 1992.

[10] Ellis, J. R., *Bulldog : A Compiler for VLIW Architectures*, The MIT press, 1986.

[11] Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, Vol. c-30, pp. 478~490, July 1981.

[12] Fisher, J. A., "Very Long Instruction Word Architectures and The ELI-512", *The 10th Annual International Symposium on Computer Architecture*, IEEE Computer Society and Association for Computing Machinery, pp. 140~150, June 1983.

[13] Gupta, R. and Soffa, M. L., "Region Scheduling: An Approach for Detecting and Redistributing Parallelism", *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, pp. 421~431, April 1990.

[14] Holm, J. G., *Evaluation of Some Superscalar and VLIW Processor Designs*, MS thesis, Illinois University, 1992.

[15] Johnson, W. M., *Superscalar Microprocessor Design*, Englewood Cliff, NJ: Prentice-Hall, 1991.

[16] Hsu, P. Y. T. and Davidson, E. S., "Highly Concurrent Scalar Processing", *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 386~395, Jun 1986.

[17] Huff, R. A., "Lifetime sensitive modulo scheduling", *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, pp. 258~257, June 1993.

[18] Hwang, K., "Advanced Parallel Processing with Supercomputer Architectures", *Proceedi-*

7) 슈퍼스칼라 프로세서는 동적 스케쥴링(dynamic scheduling)을 지원하여 동시에 수행되는 명령어들간의 종속성 문제를 컴파일러가 책임지지 않아도 된다는 점에서 VLIW 시스템과 차이는 있지만, 최적화를 위한 컴파일러 기법상에서는 차이가 없다.

ngs of IEEE, Vol. 75, No. 10, October 1987.

[19] Lam, M. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318~328, June 1988.

[20] Lilja, D. J., "Exploiting the Parallelism Available in Loops", *IEEE Computer*, pp. 13~26, February 1994.

[21] Mahlke, S. A., Lin, D. C., Chen, W. Y., Hank, R. E. and Bringmann, R. A., "Effective Compiler Support for Predicated Execution Using the Hyperblock", *Proceedings of the 25th Annual Workshop on Microprogramming*(Micro-25), pp. 45~54, December 1992.

[22] Mahlke, S. A., Chen, W. Y., Hwu, W. W., Rao, B. R. and Schlansker, M. S., "Sentinel Scheduling for VLIW and Superscalar Processors", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 238~247, October 1992.

[23] Moon, S.-M., *Compile time Parallelization of Non-numerical Code; VLIW and Superscalar*, Technical Report CS-TR-3068, Maryland University, 1993.

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

[25] Rau, B. R., Yen, D. W. L., Yen, W. and Towle, R. A., "The Cydra 5 departmental supercomputer", *IEEE Computer*, pp. 12~35, January 1989.

[26] Su, B. and Wang, J., "GURPR*: A new global software pipelining algorithm", *Proceedings of the 24th Annual Workshop on Microprogramming and Microarchitecture*, pp. 212~216, November 1991.

[27] Wang, J. and Eisenbeis, C., *Decomposed Software Pipelining*, Rapport de Recherche INRIA, INRIA, 1993.

[28] Warter, N. J., Bockhaus, J. W., Haab, G. E. and Subramanian, K. "Enhanced Modulo Scheduling for loops with conditional branches", *Proceedings of the 25th International Symposium on Microarchitectures*, pp. 170~179, November 1992.

[29] Warter, N. J., Lavery, D. M. and Hwu, W. W., "The Benefits of Predicated Execution for Software Pipelining", *HICSS-26 Conference Proceedings*, Vol. 1, pp. 497~506, January 1993.

[30] Warter, N. J., *Modulo Scheduling with Isomorphic Control Transformation*, PhD thesis, Illinois University, 1994.

[31] Gasperoni, F. and Schwiegelshohn, U., *Scheduling Loops on Parallel Processors: A Simple Algorithm with Close to Optimum Performance*, Lecture Note, INRIA, 1992.

박 명 순

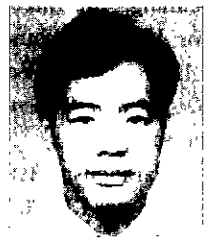
1975 서울대학교 전기공학과 석사
1982 Utah대학교 전기공학과 석사
1985 Iowa대학교 전기전산공학과 박사
1975 ~1980 국방과학연구소 연구원
1985 ~1987 Marquette대학교 조교수
1987 ~1988 포항공과대학 전자전기공학과 조교수
1988 ~현재 고려대학교 전산과학과 부교수
관심 분야 : 컴퓨터 구조, 병렬 컴파일러

이 진 호

1992 고려대학교 전산과학과 학사
1994 고려대학교 전산과학과 석사
1994 ~현재 고려대학교 전산과학과 박사과정
관심 분야 : 병렬처리 컴퓨터 구조, 병렬처리 알고리즘

양 정 일

1992 고려대학교 전산과학과 학사
1993 ~현재 고려대학교 전산과학과 석사과정
관심 분야 : 병렬 컴파일러, 병렬 언어



박 성 순

1984 홍익대학교 전자계산학과 학사
1987 서울대학교 계산통계학과 석사
1988 ~1990 공군사관학교 전산학과 전임강사
1994 고려대학교 전산과학과 박사
1994 ~현재 대신대학교 전자계산학과 전임강사
관심 분야 : 병렬 컴파일러, 병렬 언어

● 제 2회 문자인식 워크숍 ●

- 일 자 : 1994년 9월 1일(목)~2일(금)
- 장 소 : 쉐라톤 워커힐 호텔
- 주 관 : 인공지능연구회
- 문 의 : 중앙대 컴퓨터공학과 권영빈 교수
  Tel : (02) 810-2250
  Fax : (02) 824-2522
  e-mail : ybkwon@ripe.chungang.ac.kr