목적 코드 레벨에서의 벡터화 기법

이 동 호† · 김 기 창††

요 약

명령어 재배치는 ILP(Instruction Level Parallelism) 프로세서의 병렬성을 활용하는 주요한 코드 최적화 기법이다. 명령어 재배치 알고리즘을 루프(loop)에 적용하면 서로 다른 반복(iteration) 사이의 동시 수행 가능한 명령어들이 인접한 위치로 모여지는 소프트웨어 파이프라인(software pipeline)된 루프가 얻어진다. 그러나 루프로부터 병렬성을 추출하는 소프트웨어 파이프라인 방법은 주로 명령어사이의 자료 종속성에 근거하여 스케줄링을 수행하므로 그 자체에 무한한 병렬성을 가지고 있는 벡터 루프의 경우 그 병렬성을 충분히 드러내지 못한다는 문제점을 안고 있다. 본 논문에서는 이러한 벡터루프에 대해 프로그램의 목적 코드 레벨에서 행해질 수 있는 새로운 벡터 스케줄링 방법을 제안한다. 벡터 스케줄링 방법은 프로그램의 목적 코드 레벨에서 루프의 구조나 반복 조건, 그리고 자료 종속성 등에 대한 전체적인 정보에 기반하여 스케줄링을 수행함으로써 소프트웨어 파이프라인 방법보다 프로그램의 수행속도를 향상시킬 수 있다. 본 논문에서는 벡터 스케줄링을 수행한 결과를 전통적인 소프트웨어 파이프라인 방법에 의해 생산된 병렬 루프의 결과와 수행속도 측면에서 비교한다.

A Vectorization Technique at Object Code Level

Dong-Ho Lee† · Ki-Chang Kim††

ABSTRACT

ILP(Instruction Level Parallelism) processors use code reordering algorithms to expose parallelism in a given sequential program. When applied to a loop, this algorithm produces a software-pipelined loop. In a software-pipelined loop, each iteration contains a sequence of parallel instructions that are composed of data-independent instructions collected across from several iterations. For vector loops, however the software pipelining technique can not expose the maximum parallelism because it schedules the program based only on data-dependencies. This paper proposes to schedule differently for vector loops. We develop an algorithm to detect vector loops at object code level and suggest a new vector scheduling algorithm for them. Our vector scheduling improves the performance because it can schedule not only based on data-dependencies but on loop structure or iteration conditions at the object code level. We compare the resulting schedules with those by software-pipelining techniques in the aspect of performance.

1. 서 론

명령어 재배치는 superscalar나 VLIW(Very Long Instruction Word)와 같은 ILP(Instruction level Parallelism) 프로세서에서 병렬성을 추출하는 주요한 기법이다[7,20]. 명령어 재배치는 서로 자료 종속성이 없는 명령어들을 프로그램 흐름 그래프 상에서 인접한 위치로 이동시킴으로써 동시에 수행될 수 있도록 한다. 명령어 재배치 알고리즘을 루프에 적용하면 서로 다른 반복(iteration) 사이의 동시 수행 가능한 명령어들이 인접한 위치로 모아진 소프트웨어 파이프라

※ 본 연구는 한국과학재단의 '97핵심전문연구과제 연구비에 의해 수행 되었음.
†준 회 원 : 인하대학교 전자계산공학과
††정 회 원 : 인하대학교 전자계산공학과
논문접수 : 1997년 11월 28일, 심사완료 : 1998년 3월 2일

이닝(software pipelining) 된 루프가 얻어진다. 루프는 CPU가 대부분의 시간을 소비하는 프로그램 구문으로서 루프로부터 병렬성을 추출하는 소프트웨어 파이프라이닝 방법은 프로그램의 병렬화에 중요한 역할을 하고 있다[1,3,10,16,19].

그러나 소프트웨어 파이프라이닝 방법은 벡터 루프의 경우 그 병렬성을 충분히 드러내지 못한다는 문제점을 안고 있다. (그림 4)는 벡터 루프의 한 예를 보여주고 있다. 원시 코드 수준에서 이 루프는 상수 값 2를 배열 a[i]에 저장하는 명확한 벡터 루프이다. 하지만, 목적 코드 수준에서는 명령어 "INC i"에 자료 종속성 사이클이 존재하므로 그 자체로는 벡터 루프로 볼 수 없다. 따라서, 소프트웨어 파이프라인 기술은 (그림 5)와 같은 병렬 루프를 생산하게 된다. (그림 5)에서 원래의 루프는 매 반복마다 5개의 명령어들을 순차로 실행하고 있으므로 총 50개의 명령어들을 순차로 실행해야 하는 데 반해, 소프트웨어 파이프라이닝 방법으로 스케줄된 경우에는 루프를 시작하기 전에 1개의 병렬 명령어를 실행하고, 매 반복마다 1개의 병렬 명령어를 실행하므로 총 11개의 병렬 명령어들을 실행한다는 것을 알 수 있다. 병렬 명령어 하나의 수행에 걸리는 시간이 순차 명령어 하나를 수행하는 시간과 같다고 가정하면 소프트웨어 파이프라인에 의해 약 5배의 프로그램 속도향상을 얻을 수 있다. 하지만, 이 루프는 원시 코드 수준에서 벡터 루프이었기 때문에 자원이 충분히 주어진다면, 목적코드 수준에서도 루프 내부에 속하는 각 명령어들이 모든 반복에 대해 동시에 수행될 수 있어야 하고, 이로 인해 더 높은 속도향상이 가능 해야 한다. (그림 10)은 같은 벡터 루프에 대해 본 논문에서 제안된 벡터 스케줄링을 수행한 결과로서 총 12개의 병렬 명령어를 수행한다는 것을 알 수 있다. 이것은 단지 4개의 functional unit을 가정하여 스케줄링을 수행한 결과로서 (그림 4)의 소프트웨어 파이프라인 방법으로 스케줄된 프로그램의 속도 향상 정도와 비슷한 결과를 얻는다. 벡터 루프는 그 자체에 무한한 병렬성을 지니고 있으므로 레지스터나 functional unit 등의 하드웨어가 허용하는 한 더 높은 속도향상을 얻을 수 있어야 한다. 본 논문이 제시하는 벡터 스케줄링은 루프 구조나 반복 조건(반복 횟수, exit 조건 등)에 대한 전체적인 정보에 기반 하여 스케줄링을 수행하기 때문에 자료 종속성만으로 스케줄하는 소프트웨어 파이프라인보다 더 높은 병렬성을 추출한다.

벡터 루프는 과학적인 계산을 수행하는 프로그램에서 빈번히 발생되는 루프로써 계산 시간의 많은 부분을 소모하는 루프이다[4,21]. SIMD 계열의 컴퓨터들은 이러한 벡터 루프들을 프로그램의 원시 코드 수준에서 검출하여 벡터 명령어로 변환시킴으로써 프로그램의 수행속도를 향상시키고 있다. 원시 코드 수준에서 벡터 루프를 검출하고 벡터화 시키는 기법들은 많은 연구가 진행되어 왔다[4,18,21]. 그러나 본 논문에서와 같이 목적 코드 수준에서 벡터 루프를 검출하고 벡터화 시키는 기법은 원시 코드 수준에서와는 다른 몇 가지 문제점들을 안고 있다. 첫째로, 목적 코드 수준에서는 루프 구조(structure)와 반복 조건들이 코드 상에 명확히 드러나 있지 않다. 따라서 목적 코드 수준에서 벡터 루프를 찾기 위해서는 먼저 프로그램 흐름 그래프 상에서 루프가 검출되어야 하고, 그 루프를 제어하는 변수(루프 control variable :이하 LCV)가 검출되어야 하며 그 LCV의 초기값, 증감 폭, 최종 값 등이 분석되어야 한다. 둘째로, 벡터 루프로 판명된 루프를 병렬화 시키는 작업에서 루프에 의해 반복되는 동일한 명령어를 하나의 벡터 명령어로 만들기 위해서는 많은 상대 레지스터가 필요하게 되고, 이 레지스터들은 그것을 사용하는 명령어로 정확하게 이식되어야 한다. 많은 상대 레지스터들을 한꺼번에 사용해야 하기 때문에 레지스터 압력을 제어하기 위한 방법 또한 고려되어야 한다.

본 논문은 목적 코드 수준에서 벡터 스케줄링을 수행함에 있어서 제기된 상기 문제점들을 해결하기 위하여 벡터 루프를 벡터화 시키는 알고리즘을 제시하고, 제시한 알고리즘에 의해 생산된 병렬 루프의 성능을 전통적인 소프트웨어 파이프라인에 의해 생산된 병렬 루프의 그것과 비교한다.

본 논문의 구성은 다음과 같다. 2장에서는 명령어 재배치 알고리즘에 대한 기본 개념과 전통적인 스케줄링 기법인 소프트웨어 파이프라이닝 알고리즘에 대해 설명하고 있다. 3장에서는 본 논문이 제안하는 벡터 스케줄링 알고리즘에 대해 설명하고 있다. 4장에서는 제안된 시스템의 스케줄링 결과 및 성능 평가에 대해 설명하고 마지막 5장에서는 결론 및 향후 연구 방향에 대해 제시한다.

2. 기존 연구의 고찰

2.1 명령어 재배치

서로 자료 종속성이 없는 명령어들은 VLIW나 수퍼스칼라와 같은 병렬 아키텍츄어(architecture)에서 자원이 허용되는 한 동시에 실행될 수 있다. 따라서 VLIW 컴파일러등은 동시에 실행될 수 있는 명령어들이 근접해 지도록 프로그램 구조를 변경하고 그러한 명령어들을 단일 VLIW 명령어로 mark를 해준다.
예를 들어,

1) ADD r1 2 r2 ; r2 <— r1 + 2
2) ADD r2 4 r5 ; r5 <— r2 + 4
3) ADD r1 4 r3 ; r3 <— r1 + 4

의 경우 명령어 1)과 3)은 서로 자료 종속성이 없으므로 동시에 수행될 수 있다. 따라서 프로그램 구조를

1) ADD r1 2 r2
3) ADD r1 4 r3
2) ADD r2 4 r5

와 같이 명령어 1) 과 3)을 근접시킨 후

V1: 1) ADD r1 2 r2
3) ADD r1 4 r3
V2: 2) ADD r2 4 r5

와 같이 mark를 하여 1)과 3)이 단일 VLIW 명령어에 속한다는 것을 표시해 준다. 명령어 재배치는 컴파일러의 코드 최적화 과정에서 중요하게 사용되는 기법[2]으로 기본 블록(basic block) 안에서의 코드 재배치나 루프 불변 명령어(loop-invariant instruction)를 루프 밖으로 빼내는 등의 작업에 사용되어 왔었다. Fisher는 기본 블록 밖으로도 명령어를 이동시킬 수 있는 기법을 개발하고 이에 근거하여 트레이스 스케줄링(Trace Scheduling)을 발표하였다[12]. Nicolau는 모든 프로그램 경로를 통해 명령어 이동이 일어날 수 있도록 트레이스 스케줄링을 개선한 퍼콜레이션 스케쥴링(Percolation Scheduling)을 발표하였다[17]. 명령어 재배치를 통하여 순차 명령어들의 나열을 VLIW 명령어들의 나열로 스케줄 하는 알고리즘을 정리해 보면 다음과 같다.

[알고리즘 1] 명령어 재배치
입력 : 명령어로 구성된 제어 흐름 그래프
결과 : VLIW 프로세서를 위해 재배치된 코드를 가진 제어 흐름 그래프
방법 :
단계 1. 스케줄링 지점을 선택한다.
단계 1.1 선택된 지점이 CFG의 외부 명령어라면, 스케줄링을 종료한다.
단계 2. 현 스케줄링 지점에 스케줄 될 수 있는 명령어들을 모두 수집
단계 3. 수집된 명령어 집합이 공집합이 아니라면
  단계 3.1 그 중 가장 먼저 스케줄 되어야 할 명령어를 선택
  단계 3.2 그 명령어를 현 스케줄링 지점으로 이동
  단계 3.3 goto 단계1
else
  단계 3.4 다음 스케줄링 지점을 계산하고, 현 스케줄링 지점을 계산된 스케줄링 지점으로 변경
  단계 3.5 goto 단계1

2.2 소프트웨어 파이프라인 기법

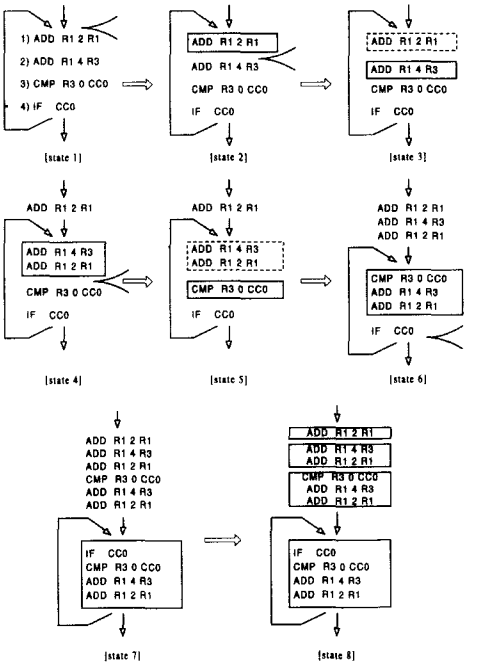
2.1 절의 명령어 재배치 알고리즘에 근거하여 루프를 스케줄하는 소프트웨어 파이프라인 기법이 많은 연구자들에 의해 발표 되었다[9,10,13,16]. 소프트웨어 파이프 라인 기법은 2.1절의 [알고리즘 1]을 루프에 적용하되 다음과 같은 제한 조건을 가지고 적용한 것으로 정리될 수 있다[9].

(1) 루프 밖으로 나가는 루프 종료 경로에 대해서는 검색 윈도우를 확장하지않는다.
(2) 검색 윈도우가 루프의 반복 경로를 통해 같은 명령어를 지나서 확장하지 않는다.
(3) 스케줄링은 스케줄링 지점이 루프 내부의 마지막 명령어를 통과 할 때 끝이 난다.

이 세가지 제한조건을 지키며 2.1장의 명령어 재배치 알고리즘을 적용한 예가 (그림 3) 에 있다.

(그림 3)에서 루프의 반복 횟수가 1000번이라고 가정을 해보자. 스케줄링 되기 전인 [state 1]에서는 매 반복마다 4개의 순차 명령어들을 실행하므로 총 4000개의 순차 명령어들을 실행한다. 이 루프에 2.1 장에서 제안된 명령어 재배치 알고리즘을 단계별로 적용해 보자. [state 1]은 현재 스케줄링 지점을 보여주고 있다. 이때, 검색 윈도우는 루프의 외부로 확장되지 않고, 경로 4)-1)을 따라 같은 명령어를 지나서 확장되지 않으므로 검색 윈도우는 1)-2)-3)-4)이다. [state 2]는 [state 1]의 스케줄링 지점에서 스케줄하기 위해 선택된 명령어를 보여준다. 여기서, 명령어(1)과 (2), (2)와 (3), 그리고, (3)과 (4) 사이에는 flow 종속성이 존

재하므로 현재 스케줄링 지점에서 명령어 (1)만이 스케줄 될 수 있다. [state 3]은 선택된 명령어를 현재의 스케줄링 지점으로 이동시키는 과정을 보여주고 있다. [state 4]는 이동되는 명령어가 joint point를 통과하므로 복사본을 남겨두고 다음 번 스케줄링 명령어가 선택된 상태이다. 이때 남겨진 명령어 (1)은 현재의 반복이 아닌 다음 번 반복에서 사용되므로 명령어 (1)과 (2)는 동시에 수행할 수 있다. 이와 같은 과정을 반복하면 [state 8]에서 볼 수 있듯이 [state 1]의 순차적인 루프는 루프 시작 전에 수행되는 3개의 병렬 명령어와 매 반복마다 수행되는 1개의 병렬 명령어로 변환된다.



(그림 1) 소프트웨어 파이프라인 스케줄링
(Fig. 1) Software Pipeline scheduling

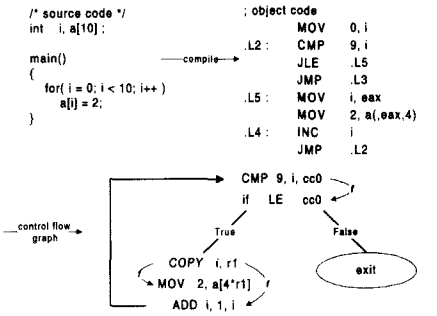
결과적으로 스케줄링 전의 4000개의 순차적인 명령어들이 1003개의 병렬 명령어로 변환이 되므로 스케줄링 결과 약 4배 정도의 속도향상을 얻을 수 있다.

3. 벡터 스케줄링

3.1 벡터 스케줄링

벡터 루프는 LCV로 인한 자료 종속성을 제외하고 모든 자료 종속성이 하향 종속성인 루프로 정의된다 [4,18]. 벡터 루프의 주요 특징은 루프 내부의 모든 종속성이 하향이기 때문에 루프 내부의 명령어들을 각 반복에 대해 병렬로 수행할 수 있다는 것이다.

(그림 5)는 (그림 4)에서 제시된 벡터 루프에 대해 소프트웨어 파이프라인에 의해 루프 스케줄링을 적용한 결과이며, (그림 10)은 같은 벡터 루프에 대해 본 논문에서 제안된 벡터 스케줄링 방법을 사용하여 스케줄링을 수행한 결과이다. 벡터 스케줄링은 기존의 소프트웨어 파이프라인 스케줄링 방법보다 스케줄링 시간이 빠르고 성능도 우수한 스케줄 결과를 생산할 수 있다.

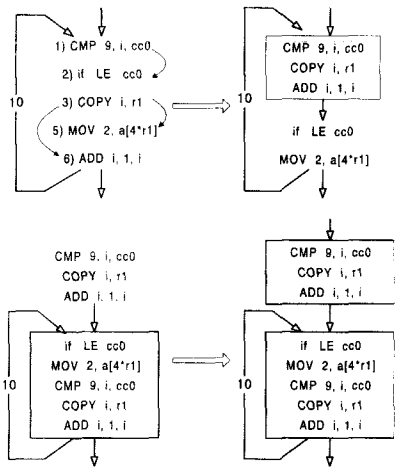


(그림 2) 벡터 루프
(Fig. 2) Vector Loop

어떤 루프가 벡터 루프임이 판명되었을 때[1] 벡터 스케줄링을 얻기 위한 알고리즘은 다음과 같다.

[알고리즘 2] 벡터 스케줄링
입력 : 명령어로 구성된 제어 흐름 그래프
결과 : 벡터 스케줄 된 제어 흐름 그래프
방법 :
단계 1. 루프 control variable (LCV)을 찾는다
LCV를 초기화하는 명령어 A와 LCV의 초기값(I)을 찾는다
LCV를 변경하는 명령어 B와 LCV의 증감폭(S), 종료 명령어 C를 찾는다.
단계 2. 루프의 unroll size "x"를 계산하여, 루프를 unroll.

1) 주어진 루프가 벡터루프인지를 판명하는 것은 원시코드 레벨에서 행해지는 벡터 판명기법에 기초하여 자료 종속성 사이클이 있는 지 여부를 판단함으로써 행해진다[22].



(그림 3) 소프트웨어 파이프라인 스케줄링
(Fig. 3) Software Pipeline Scheduling

단계 3. 레지스터 재명명을 사용하여, 모든 output 종속성을 제거한다.

단계 3.1. 이때, LCV에 의한 output 종속성은 무시한다.

단계 3.2. 만약 레지스터 재명명을 위해 사용할 수 있는 레지스터의 수가 부족하다면, 루프의 입구에서 레지스터를 저장하고, 루프의 출구에서 다시 복구한다.

단계 4. 루프 body의 모든 복사본에 대해 LCV를 서로 다른 레지스터로 교환한다.
만약 레지스터 수가 부족하다면, 단계 3.1~3.2의 과정을 반복한다.

단계 5. 명령어 A를 다음과 같이 교환한다.
COPY i, LCV to
COPY i, r1; COPY i+S, r2; ; COPY i+(x-1)S, rx

단계 6. 명령어 B의 각 복사본을 다음과 같이 교환한다.
updateop ri, S, ri to
updateop ri, x*S, ri

단계 7. 루프의 i번째 복사본의 각 exit 명령어 C의 exit branch에
COPY ri, LCV를 삽입한다.

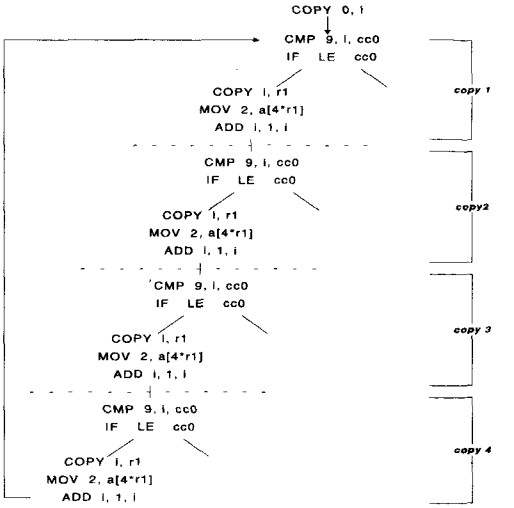
단계 8. 루프 body의 첫번째 복사본에서의 각 명령어에 대해 중복된 명령어를 수집한다.

제안된 벡터 스케줄링 알고리즘의 단계 2 에서 "x"는 동시에 실행될 수 있는 명령어의 수로서, 일반적으로 하드웨어가 제공하는 연산 처리기의 수를 의미한다. 일반적으로 대부분의 하드웨어에서 제공되는 레지스터는 한정되어 있으므로, 단계 3.1 ~ 3.2 와 같이 레지스터를 확보하기 위해 루프의 시작부분과 끝부분에 루프에서 사용되는 레지스터들을 저장하고 반환하는 부분이 삽입된다. 기본적으로, 단계 4 ~ 6에서 보는 것처럼 루프는 x개의 부분루프로 분할되고, 각 부분루프는 단계 2에서 결정된 "x" 값을 참조하여 계산된 증감폭을 갖는 서로 다른 LCV(r1, r2, r3, , rx)를 갖는다. 각 부분루프의 루프 종료 경로에는 LCV를 다시 반환하는 명령어가 삽입된다.

제안된 벡터 스케줄링 알고리즘을 (그림 4)의 벡터 루프에 적용해 보자.

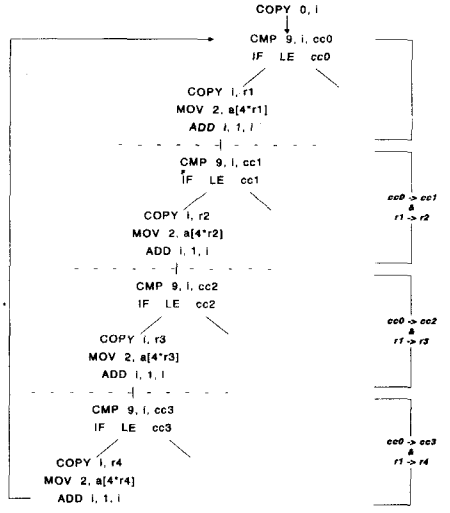
이 루프의 LCV는 i이고, LCV를 초기화하는 명령어 A는 "COPY 0, i"이며 LCV의 초기값 I는 0이다. 또한, LCV를 변경하는 명령어 B는 "ADD i, 1, i"이고 증감 폭 S는 1이다. 마지막으로, 이 루프의 종료 조건을 테스트하는 명령어 C는 "if LE cc0"이다. 이때, 연산 처리기의 수, 즉 "x"를 4라고 가정하자. 루프는 (그림 6)에서 보는 바와 같이 4개의 부분루프로 분할된다.

LCV에 의해 야기되는 output 종속성을 제외한 루



(그림 4) 벡터 스케줄링 (단계 2 적용 후)
(Fig. 4) Vector Scheduling (After Step 2)

프 분할에 의해 발생되는 모든 output 종속성은 제안된 알고리즘의 단계 3에서 보는 것처럼 레지스터 재명명에 의해 제거된다. 즉, 명령어 "CMP 9, i, cc0"의 cc0에 의해 발생되는 output 종속성을 제거하기 위해 2nd, 3rd, and 4th 복사본의 cc0를 각각 cc1, cc2, cc3로 재명명하고, 명령어 "COPY i, r1"의 r1에 의해 발생되는 output 종속성을 제거하기 위해 2nd, 3rd, and 4th 복사본의 r1을 각각 r2, r3, r4로 재명명 한다. 이 때, 재명명 된 레지스터들이 정확하게 이식되기 위해서는 2nd, 3rd, 4th 복사본들 내의 명령어 "IF LE cc0" 와 "MOV 2, a[4*r1]"의 레지스터들이 적절히 변경되어야 한다. (그림 7)은 이를 잘 설명하고 있다.

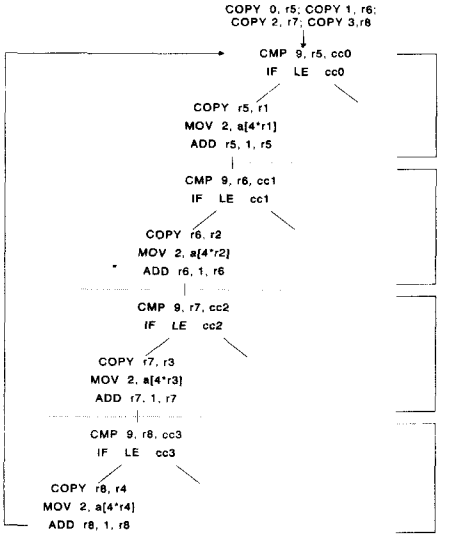


(그림 5) 벡터 스케줄링 (단계 3 적용 후)
(Fig. 5) Vector Scheduling (After Step 3)

각 복사본의 LCV는 (그림 8)에서 보는 것처럼 각각 r5, r6, r7, r8등의 활성 레지스터로 변경된다. 이로써 각 복사본들은 서로 독립적인 LCV를 갖게 되고 결과적인 루프는 (그림 8)과 같다.

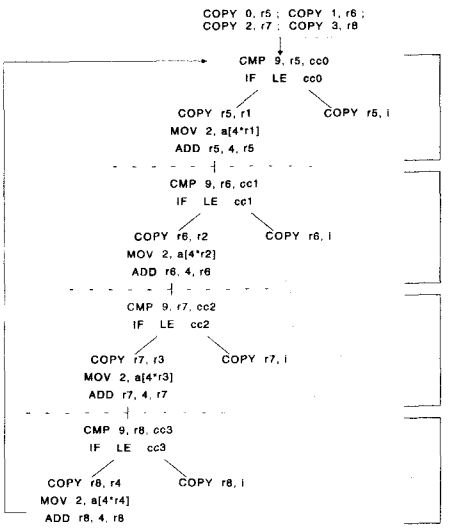
각 복사본들의 각 LCV는 제안된 알고리즘 단계 2에서 결정된 "x"의 값을 참조하여 초기화 되고, 이에 맞게 각 복사본들의 LCV 변경 명령어도 변경된다. 즉 (그림 9)에서 보는 것처럼 실제 LCV, 즉 i의 초기값은 0이고, 증감 폭은 1 이므로 각 복사본들의 LCV는 0, 1, 2, 3으로 초기화되고, LCV의 변경 명령어들도 적절히 변경된다. 이때, 변수 i의 정확한 값을 보존하기 위해 각 복사본들의 루프 exit 경로에 각각의 LCV의 값을 다시 i로 복사하는 명령어가 삽입되어야 한다.

루프 내부에 대한 모든 복사본들은 제안된 벡터 스케줄링 알고리즘의 단계 3~7 까지의 과정을 통해 서로 완전히 독립적인 루프가 된다. 루프는 네 번 분할 되었으므로, 각 명령어들은 4개의 복사본들이 존재하고, 제안된 벡터 스케줄링 알고리즘의 단계 1 에서 7까지의 과정을 통해 4개의 복사본들은 서로 동시에 수행될 수 있다는 것이 확인 되었다. 먼저, 각 LCV의 초기화 명령어들, "COPY 0, r5; COPY 1, r6; etc.",은 하나의 병렬 명령어로 쉽게 수집될 수 있다. 다른 병렬 명령어들은 2장에서 설명한 코드 이동 방법에 따라 수집될 수 있다. 그 결과는 (그림 10)에서 보는 바와 같다. 그림에서, 병렬화 된 루프의 첫번째 병렬 명령어는 "CMP 9, r5, cc0; CMP 9, r6, cc1; etc."라는 것을 볼 수 있다. 첫번째 "CMP" 명령어인 "CMP 9, r5, cc0"를 제외한 다른 "CMP" 명령어들은 이동 규칙에 따라 이동된 명령어이다. 루프의 두 번째 병렬 명령어는 "IF LE cc0; IF LE cc1; etc."이다. 여기서, 첫번째 "IF" 명령어를 제외한 나머지 세 개의 "IF" 명령어도 또한 명령어 이동 규칙에 따라 이동시킨다.

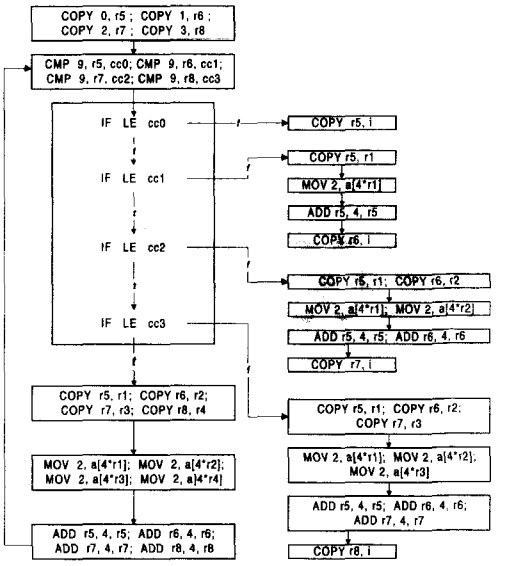


(그림 6) 벡터 스케줄링 (단계 4, 5 적용 후)
(Fig. 6) Vector Scheduling (After Step 4 and 5)

이 때 "IF" 명령어를 이동하는 것은 2장에서 설명한 것처럼 일종의 트릭이다. "IF" 명령어가 프로그램의 앞부분으로 이동될 때, "IF"명령어를 따라 전달되는 모든



(그림 7) 벡터 스케줄링 (단계 6, 7 적용 후)
(Fig. 7) Vector Scheduling (After Step 6 and 7)



(그림 8) 벡터 스케줄링 결과
(Fig. 8) Resulting Vector Schedule

명령어들은 "IF"명령어의 양쪽(True/False) 경로에 중복되어야만 한다. 이것이 "IF"명령어의 오른쪽(False) 경로에서 나타나는 명령어들의 수를 알고 있어야 하는 이유이다. 나머지 병렬 명령어도 유사한 방법으로 수집된다.

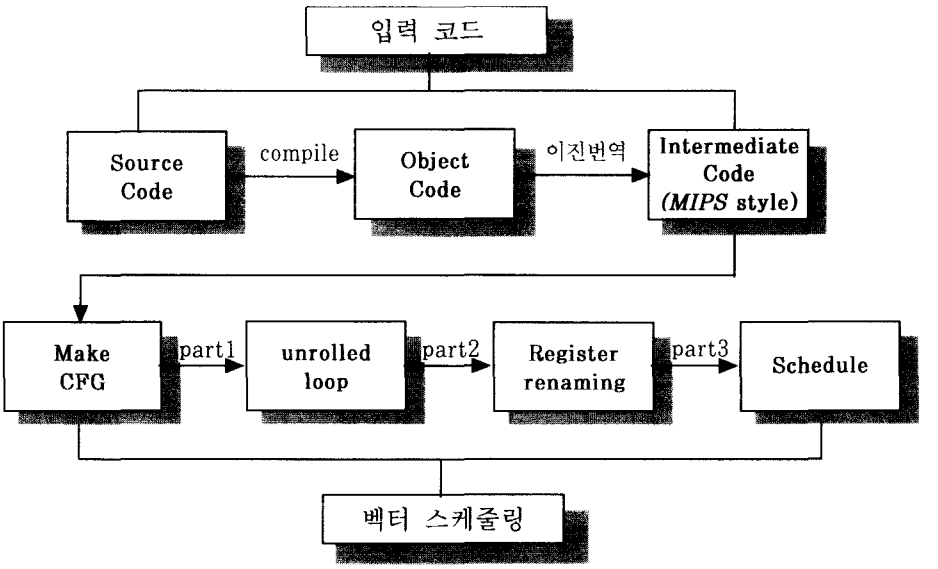
(그림 10)의 벡터 스케줄링이 적용된 결과와 (그림 5)의 소프트웨어 파이프라인 방법을 사용하여 스케줄한 루프를 비교해 보자. 먼저, 속도향상 측면에서 볼 때, 소프트웨어 파이프라인 방법이 적용된 루프는 매 반복마다 동시에 5개의 명령어를 실행하기 때문에 대략 5배의 속도향상을 보여준다. 벡터 스케줄 된 루프의 경우 그 루프를 네 번 풀어서 스케줄링을 수행했기 때문에 대략 4배의 속도향상을 나타낸다. 그러나, 루프를 푸는 횟수를 증가함에 따라 벡터 스케줄링은 더 높은 속도향상을 얻을 수 있지만, 소프트웨어 파이프라인의 경우 이 루프로부터 얻을 수 있는 최대의 속도향상이기 때문에 더 이상의 속도향상을 얻을 수 없다. 두 번째, 코드 크기 증가의 측면에서 볼 때, 벡터 스케줄은 루프를 푸는 과정에 기본을 두고 있기 때문에 소프트웨어 파이프라인 방법에 비하여 더 높은 코드 증가를 나타낸다. 그러나, 벡터 스케줄링은 주로 작은 크기의 가장 안쪽에 있는 루프(innermost loop)에 적용되므로 코드 크기 증가로 인한 캐쉬 미스(cache miss)의 영향은 크지 않을 것으로 사료된다. 마지막으로, 스케줄링 비용 측면에서 벡터 스케줄링 방법과 소프트웨어 파이프라인 방법을 비교해 보면, 소프트웨어 파이프라인 방법은 "uo(unifiable operations, 즉 동시에 수행될 수 있는 명령어들)"을 계산하는 데 대부분의 시간을 소모한다. 즉, 매번 명령어를 이동시킬 때마다 현재의 스케줄링 지점에서 어떤 명령어가 이동될 수 있는지를 항상 계산해야만 한다. 더욱이, 제한된 functional unit의 효율을 극대화하기 위해 수집된 명령어들 중에서 알맞은 명령어들을 선택해야 한다. 이를 위해 명령어들마다 우선권을 계산해야 하는데 어떤 명령어가 더 중요하고 덜 중요한지를 계산하는 것은 시간을 많이 소비하는 정교한 휴리스틱(heuristic)을 필요로 한다[19]. 이에 비해 벡터 스케줄링은 대부분의 시간을 상대적으로 기계적인 단계로 수행될 수 있는 벡터판명과 루프전개등에 소비한다.

4. 벡터 스케줄링 시스템 구현 및 실험

4.1 벡터 스케줄링 시스템

본 논문에서 구현한 벡터 스케줄링 시스템의 전체적인 구조는 (그림 9)와 같다.

본 논문에 사용된 원시 코드는 일반적인 C 프로그램 코드로서 x86 기계를 기반으로 한 LINUX 시스템에서



(그림 9) 벡터 스케줄링 시스템 구조
(Fig. 9) Architecture of Vector Scheduling System

gcc를 이용하여 목적 코드로 변환이 된다. 여기서 생성된 목적 코드를 기준으로 명령어 대 명령어 스타일로 번역을 하는 이진 번역기를 역시 같은 환경하에서 LEX와 YACC을 사용하여 구현하였다. 이 이진 번역기를 사용하여 본 논문에서 구현한 벡터 스케줄링 시스템을 위한 MIPS 스타일의 중간 코드를 생성한다.

본 논문에서 제안된 벡터 스케줄링 알고리즘을 사용한 벡터 스케줄링 시스템은 다음과 같은 과정으로 구현되었다. 먼저 이진 번역기를 통해 얻어진 중간 코드를 입력으로 하여 입력 코드에 대해 LCV 초기화 명령어, LCV 변경 명령어, 그리고 루프 종료 명령어를 참조하여 루프의 몸체 부분만을 찾는다. 찾아낸 루프 몸체 부분을 기준으로 루프 컨트롤 그래프를 생성한다. 일단 루프 컨트롤 그래프가 생성되면 루프를 푸는 회수를 계산하여 루프의 복사본을 만들어 낸다. 각 복사본에서 사용되는 레지스터들에 대해서 서로 자료 종속성이 없어지도록 레지스터 재명명을 한다. 본 논문에서는 무한 레지스터를 가정하였으며 레지스터 재명명 과정은 입력으로 받은 루프에 대해 첫번째 복사본을 기준으로 사용되는 일반 레지스터의 수, 컨트롤 레지스터의 수, 그리고 메모리 주소를 위한 레지스터의 수를 계산한다. 여기서 계산된 각 레지스터들의 수를 기준으로 각 복사본에 대해서 레지스터 재명명을 실행한다. 그리고 마지막으로 이와 같은 과정을 통해 완성된 리스트에 대해서 각 복사본의 같은 순차 명령어를 하나의 병렬 명령어로 수집하여 스케줄링을 수행한다

(그림 13)은 벡터 스케줄링 시스템에 대한 입력코드가 생성되는 과정을 보여준다.

4.2 실험

소프트웨어 파이프라인 기법에 비교할 때 벡터 스케줄링의 장점과 단점을 정리해 보면 〈표1〉과 같다.

〈표 1〉 소프트웨어 파이프라인 기법과 비교할 때 벡터 스케줄링의 장단점
〈Table 1〉 Advantage & Disadvantage of Vector Scheduling

장 점	단 점
명령어를 이동시킬 때 마다 스케줄링을 할 필요가 없다. Speed-up이 우수하다. 명령어 복사로 인한 코드 확장이 없다. MMX 등의 멀티미디어 프로세서가 요구하는 SIMD 형태의 명령어 산출에 적합하다.	벡터 루프임을 판명해야 한다. 벡터 루프에만 적용될 수 있다. 레지스터의 사용이 상대적으로 많다.

```
/*source code :: c program */
#include <stdio.h>

#define SIZE 100

int     i, x[SIZE], y[SIZE]


main()
{       for(i=0;i<n;++i)
          x[i]=y[i+1]+y[i]
}
```

```
/*object code :: assembly code */
        movl $0, i
.L2:    movl $0, i
.L2:
        cmpl$9,n
        jle.L5
        jmp.L3
        .align4
.L5
        movl i, %eax
        movl i, %eax
        movl %edx, %edx
        leal 0(,%ecx,4), %edx
        movl i, %ecx
        movl y+4(%edx), %ebx
        addl y(,%ecx,4), %ebx
        movl %bx,x(,%eax, 4)
.L4:
        incl I
        jmp .L2
        .align 4
.L3:
```

```
/*intermediate code*/
l1      copy    0 i
        cmp     n i cc0
        if      le cc0 exit
        copy    i r1
        copy    i r4
        mov     r4 r3
        mul     r3 r5
        lea     mr5 r4
        copy    i r3
        add     y r4 r5
        add     r5 4 r5
        mov     mr5 r2
        mul     r3 4 r5
        add     y r5 r6
        add     mr6 r2 r2
        mul     r1 4 r5
        add     x r5 r6
        mov     r2 mr6
        add     i 1 i
        jmp     l1
exit
```

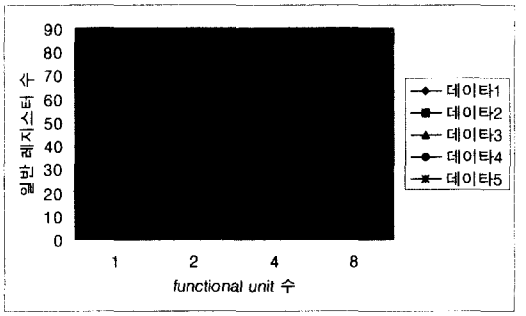
(그림 10) 벡터 스케줄링 시스템 입력 코드 생성 과정
(Fig. 10) Process of input code generation for Vector Scheduling System

본 논문에서는 Livermore loop의 kernel.c 와 "Numerical Methods in FORTRAN"[14]으로부터 발췌된 벡터루프에 대해 다음과 같이 실험하였다.<표 2>는 functional unit의 수를 증가시켰을 때 각 루프들에 대해 벡터 스케줄링의 성능을 실험한 결과이다. GR은 general purpose 레지스터를 말하며,CR은 조건(condition) 레지스터를 의미한다.

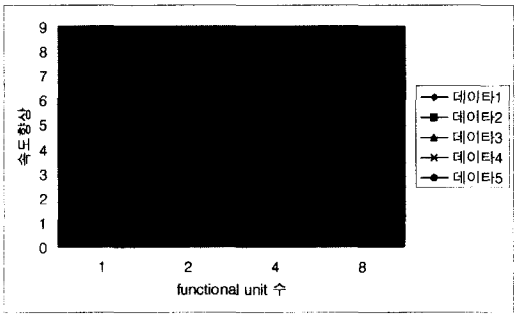
<표 2>에서 속도향상이 하드웨어가 제공하는 functi-onal unit의 수에 거의 정비례한다는 것을 알 수 있으며(그림 15 참조), 사용되는 레지스터의 수도 역시 functional unit의 수에 비례하여 많아짐을 알 수 있다(그림 14 참조). 또한, 입력 코드의 명령어 수에 관계없이 일정한 속도 향상을 얻을 수 있음을 알 수 있다. 하지만 functional unit의 수가 일정한 경우 루프의 푸는 횟수는 일정한 횟수 이상에서는 속도향상에 기여하지 못한다는 것을 알 수 있다(그림 16 참조).

<표 2> 실험 데이터
<Table 2> Experimental Data

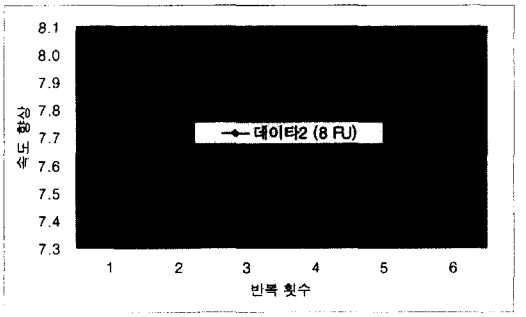
		Sequential code	2 (FU)	4	8
Test((Int) 11) in kernel.c 명령어수:18	GR	6	14	28	56
	CR	1	2	4	8
	Speed-up	1	1.98	3.92	7.86
FOURTH ORDER BOUNDARY VALUE PROBLEM 명령어수:31	GR	8	18	36	72
	CR	1	2	4	8
	Speed-up	1	1.99	3.98	7.99
LARGEST IGENVALUE OF MATRIX BY ITERATION 명령어수:30	GR	9	20	40	80
	CR	1	2	4	8
	Speed-up	1	1.99	3.97	7.97
HYPERBOLIC PATIAL DIFFERENTIAL EQUATION 명령어수:28	GR	7	16	32	64
	CR	1	2	4	8
	Speed-up	1	1.99	3.95	7.93
TWO DIMENSIONAL BOUNDARY VALUE PROBLEM 명령어수:18	GR	5	12	24	48
	CR	1	2	4	8
	Speed-up	1	1.98	3.93	7.89



(그림 11) functional unit의 증가에 따른 레지스터 증가 비율
(Fig. 11) Number of Register VS. number of functional units



(그림 12) functional unit의 증가에 따른 속도향상 비율
(Fig. 12) Speed-up VS. number of functional unit



(그림 13) 루프의 반복 횟수 증가에 따른 속도 향상 비율
(Fig. 13) Speed-up VS. loop unrolling counts

〈표 3〉은 8 functional unit의 경우에 대해 벡터스케쥴과 소프트웨어 파이프라이닝의 속도향상을 비교한 것이다. 소프트웨어 파이프라이닝은 벡터스케쥴링에서와 같이 무한 레지스터를 사용하였고 윈도우크기 역시 무한하다고 가정하였다. 이러한 가정은 소프트웨어 파이프라이닝이 주어진 functional unit을 가지고 얻을 수 있는 최대 속도향상을 보이도록 하기 위함이다. 무한 functional unit이 주어지는 경우 이론적으로 소프트웨어 파이프라이닝은 x 개의 명령어를 가지고 있는 벡터루프에 대해 각 명령어가 1 cycle씩 걸린다고 가정할 때 최대 x배의 속도향상을 이룰 수 있다. 예를 들어 18개의 명령어를 갖는 kernel.c의 경우 18개 이상의 functional unit이 주어지는 경우 최고 18배의 속도향상을 보일 수 있다. 그 이상의 속도향상이 불가능한 이유는 각 반복이 파이프라인될 때 적어도 1 cycle의 skew가 반복 사이에서 일어나도록 요구하고 있기 때문이다. 벡터스케쥴은 이러한 인위적인 skew를 요구하지 않기 때문에 루프내의 명령어의 수와 상관없이 functional unit의 수에만 제한을 받는 속도향상을 얻을 수 있다.

functional unit의 수가 제한되는 경우에도 〈표 3〉에서 보는 바와 같이 소프트웨어 파이프라이닝은 벡터스케쥴에 비해 낮은 속도향상을 보인다. 이 경우에도 소프트웨어 파이프라이닝은 루프내의 명령어의 수에 따라 얻을 수 있는 최대 속도향상이 제한을 받는다. 즉 루프 내 명령어의 수가 functional unit수의 정확한 배수일 때만 function unit의 수에 근접한 속도향상을 얻을 수 있다.

실험에 사용된 위 루프들은 모두 벡터루프 들이다.

〈표 3〉 벡터스케쥴과 소프트웨어 파이프라이닝 및 원시코드 수준 벡터화의 속도향상 비교
〈Table 3〉 Comparing Vector scheduling with software pipelining and Source-code-level vectorization in speed-ups

	벡터 스케쥴	소프트웨어 파이프라이닝	원시코드수준 벡터화
Test((Int) 11) in kernel.c 명령어수:18	7.86	5.77	8
FOURTH ORDER BOUNDARY VALUE PROBLEM 명령어수:31	7.99	5.96	8
LARGEST IGENVALUE OF MATRIX BY ITERATION 명령어수:30	7.97	5.88	8
HYPERBOLIC PATIAL DIFFERENTIAL EQUATION 명령어수:28	7.93	5.38	8
TWO DIMENSIONAL BOUNDARY VALUE PROBLEM 명령어수:18	7.89	4.5	8

따라서 위 루프들을 원시코드 수준에서 벡터화를 한다 면 목적코드 수준에서 벡터화를 하는 것과 유사한 속도 향상을 얻을 수 있을 것이다. 하지만 목적코드 수준에서 벡터화를 시도하는 가장 큰 이유는 원시코드가 존재하지 않고 목적코드만 존재하는 프로그램을 병렬화 하기 위함이다. 원시코드가 존재하는 데도 목적코드 수준에서 벡터 스케쥴을 시도하는 것이 어떤 장점과 단점이 있을 수 있는 가는 앞으로의 연구과제이다. 단점으로는 우선 1 장에서 지적한 바와 같이 벡터판명과정과 벡터화 과정이 더 복잡할 수 있다는 것이다. 또한 원시코드 수준에서 존재하던 벡터성이 목적코드로 변환되었을 때 그대로 모두 존재하는 가의 문제가 있다. 특히 원시코드 상에서 수행될 수 있는 정교한 벡터화 기법이 목적코드로 변환되었을 때에도 적용될 수 있는가에 대한 연구가 필요하다. 원시코드가 존재하는 데에도 목적코드로 변환한 후 벡터화 할 때의 장점이 될 수 있는 것은 벡터 명령어를 제공하지 않는 병렬 프로세서의 경우 직접 목적코드 상에서 병렬화 함으로써 아키텍츄어에 적합한 병렬화를 시도할 수 있다는 것이다. 즉 아키텍츄어에 독립적인 원시코드 상에서 보다는 아키텍츄어의 제한조건 들을 직접 볼 수 있는 목적코드 상에서 보다 효율적인 벡터화가 가능할 것이라는 예측이다.

5. 결론 및 향후 연구 방향

VLIW(Very Long Instruction Word) 프로세서나 이미 상용화되어 있는 수퍼스칼라 프로세서들은 인스트럭션 레벨의 병렬성을 드러내기 위해 소프트웨어 파이프라이닝 기법에 크게 의존 하고 있다. 그러나 소프트웨어 파이프라이닝 기법은 단순히 자료 종속성에 대한 정보 만에 의지하여 스케줄링을 수행하기 때문에 프로그램 수행시간의 주요 부분을 차지하는 벡터 루프에 대해서는 그 무한 병렬성을 충분히 드러내지 못한다는 문제점을 안고 있다. 본 논문은 벡터루프의 병렬성을 최대한 드러낼 수 있는 벡터 스케줄링 기법을 제시하고 그 효용성을 예제와 실험을 통해 보여주고 있다. 벡터 루프의 병렬화는 원시코드 레벨에서 많은 연구가 진행되고 그 결과가 발표되어 왔다. 그러나 목적코드 레벨에서는 루프 제어 변수(loop control variable)로 인한 자료 종속 사이클 때문에 이론적으로 벡터 루프는 존재할 수 없고 따라서 벡터 루프의 병렬성을 최대한 드러내기 위해서는 새로운 벡터 스케줄링 기법이 필요하게 된다. 본 논문은 벡터 루프를 functional unit의 수만큼 전개(unroll)한 후, 루프 제어 변수로 인한 자료 종속 사이클을 레지스터 재명명에 의해 모두 제거하고, 루프 몸체 내의 각 명령어들이 전개(unroll)된 반복들에 대해 동시에 수행될 수 있도록 전통적인 코드 이동 기법에 의해 명령어별로 모아서 병렬 명령어로 묶어 줌으로써 루프의 병렬성을 드러내는 벡터 스케줄링 알고리즘을 제시하였다. 소프트웨어 파이프라이닝 기법이 얻어낼 수 있는 최대 병렬성이 functional unit의 수와 루프내의 명령어의 수에 의해 동시에 제한되는 데 반해 제안된 벡터 스케줄링 기법은 제공되는 functional unit의 수에 의해서만 추출할 수 있는 병렬성이 제한된다. 또한 소프트웨어 파이프라이닝과는 달리 동일한 명령어가 여러 반복에 대해 동시에 수행될 수 있다는 것을 명시적으로 보여줌으로써 MMX 프로세서등의 멀티미디어 프로세서들이 제공하는 SIMD형 병렬 명령어들(padd, pmul, 등)의 산출에 유리할 것으로 사료된다.

참 고 문 헌

[1] Abnous, A. and Bagherzadeh, N., "Pipelining and bypassing in a VLIW processor," *Proc. Trans. Para. Dist. Sys.*, Vol. 5, No. 6, pp. 658-664, June 1994.

[2] Aho, A. V., Sethi, R. and Ullman, J. V. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] Aiken, A. and Nicolau, A., "Perfect Pipelining: a new loop parallelization technique," *Proc of the 1988 European Symposium on Programming*, pp. 221-235, Springer Verlag Lecture Notes in Computer Science No. 300, March 1988.

[4] Allen, J. R. and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Trans. on Programming Languages and Systems*, Vol. 9, No. 4, pp. 491-542, Oct. 1987.

[5] Briggs, Preston, Cooper, Keith, Kennedy, Torczon., "Colouring Heuristics for Register Allocation," *Proc. ACM SIGPLAN Conference*, 1989.

[6] Capitanio, A., Dutt, N., and Nicolau, A., " Partitioning of variables for multiple-register -file VLIW architectures," *Proc. Inter. Conf. Para. Pro.*, pp. I298-I301, 1994

[7] Chen, S, W., Fuchs,K., and Hwu, W., "An analytical approach to scheduling code for superscalar and VLIW architectures," *Proc. Inter.Conf. Para. Pro.*, pp. I258-I292, 1994.

[8] Chriss, S., Bryce, C., and etc, "Instruction level profiling and evaluation of the IBM RS/6000," *Proc. Inter. Symp. Comp. Arch.*, pp. 180-189, 1991.

[9] Ebcioglu, K., "A Compilation Technique for Software Pipelining of Loops with Conditional Jumps," *In Proc. of the 20th Annual Workship on Microprogramming*, pp. 69-79, ACM Press, 1987

[10] Ebcioglu, K and Nakatani, T., "A New Compilation Technique for Parallelizing Loops with unpredictable Branches on a VLIW Architecture," *in Languages and Compilers for parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua (eds.), *Research Monographs in Parallel and Distributed Computing*, pp. 213-229, The MIT Press, 1990

[11] Ferrante, J., Ottenstein, J. and Warren, J.D., "The Program Dependence Graph and its Use in Optimization," *ACM Tras. On Programming Languages and Systems*, Vol. 9, No. 3, pp. 319-349, July 1987

[12] Fisher, J. A., "Trace Scheduling:A Technique for Global Microcode Compaction," *IEEE Trans. On Computers*, Vol. C-30, No. 7, July 1981.

[13] Gross, T. and Ward, M. "The supression of compensation code. In D. Gelernter et al. (eds)." *Proc. of the 3rd Wokshop on Progra -mming Languages and Compilers for paral -lel Computing*, pp. 260-273, Pitman/MIT Press.

[14] McCormick, J.M. & Salvadori, M.G., *Numerical Methods in FORTRAN*, Prentice-Hall, Inc., 1964.

[15] Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M., "Dependence Graphs and Compiler Optimizations," *Proc. of the 8th ACM Symp.On Programming Lan -guages*, illiamsburg, VA, pp. 207-218, Ja n. 1981.

[16] Lam, M., "Software Pipelining: An effective scheduling technique for VLIW," PhD thesis, Carnegie Mellon, May 1987.

[17] Nicolau, A. "Percolation Scheduling: A Par -allel Compilation Technique," TR-85-678, Dept. of Computer Science, Cornell Univ.

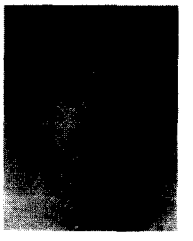
[18] Polychronopoulos, C. D., Kuck, D. J., and Padua, D. A., "Execution of Parallel Loops on Parallel Processor Systems," *Inter. Conf. Para. Proc.*, pp. 519-527, August, 1986.

[19] Ruttenberg,J. et al., "Software Pipelining Showdown:Optimal vs. Heuristic Methods in a Production Compiler," Proc. of PLDI, ACM Press, New York, 1996, pp.1-11.

[20] Schlansker, M., Conte T.M., Dehnert J., Ebcioglu, K., Fang J.Z., and Thompson C.L., "Compilers for Instruction-Level Parallelism", Computer, December 1997, pp. 63-69.

[21] Wolfe, M., *Optimizing Supercompilers for Supercomputers*, The MIT Press, 1986

[22] Dong-Ho, Lee, Ki-Chang Kim, "Vectorizer for ILP machines", to appear in 인하대학교 산업과학기술연구소 논문집 26, 1998. 2.

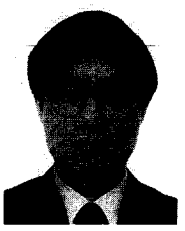


이 동 호

1996년 인하대학교 전자계산공학과 졸업(공학사)

1998년 인하대학교 대학원 전자계산공학과 졸업(공학석사)

관심분야 : 병렬화 컴파일러, 네트워크 등



김 기 창

1986년 California State Polyt-echnique University, Pomona 졸업(학사)

1992년 University of California, Irvine 졸업(공학석사, 박사)

1992년~94년 IBM T. J. Watson 연구소 연구원

1994년~현재 인하대학교 전자계상공학과 조교수

관심분야 : 병렬화 컴파일러 등