적극적인 명령어 압축을 통한 성능향상

지 승 현†·김 석 일††

요 약

본 논문에서는 독립적으로 스케줄링할 수 있는 VLIW 명령어들을 소개함으로써, 컴파일러와 프로세서에서의 스케줄링 작업을 더욱 균등하게 분배할 수 있는 프로세서 구조를 제안하였다. 제안한 *APVLIW*(Aggressively Packed VLIW) 프로세서의 목표는 자료종속성을 포함한 VLIW 명령어들을 독립적으로 스케줄링이다. APVLIW 프로세서는 기존의 VLIW 코드로부터 대부분의 NOP(No OPerations)과 LNOP(Long NOPs) 명령어들을 제거함으로써 압축된 형태의 긴명령어 그룹을 생성한다. 본 논문에서 제안된 APVLIW 프로세서는 여러 개의 연산처리기와 동적 스케줄러의 쌍들과 자료종속성 정보를 사용하여 긴명령어내의 각 명령어를 독립적으로 스케줄링할 수 있다. 이러한 스케줄링 기법은 특히 루프를 포함한 프로그램을 실행할 때 효과적이다. 실험 결과를 통해서 캐시크기의 변화와 벤치마크 프로그램에 상관없이 APVLIW 프로세서가 VLIW 프로세서에 비하여 성능이 향상됨을 확인하였다.

Performance Improvement Through Aggressive Instruction Packing

Sunghyun Jee† · Sukil Kim††

ABSTRACT

This paper proposes balancing scheduling effort more evenly between the compiler and the processor, by introducing independently scheduled VLIW instructions. Aggressively Packed VLIW (APVLIW) processor is aimed specifically at independent scheduling Very Long Instruction Word (VLIW) instructions with dependency information. The APVLIW processor independently schedules each instruction within long instructions using functional unit and dynamic scheduler pairs. Every dynamic scheduler dynamically checks for data dependencies and resource collisions while scheduling each instruction. This scheduling is especially effective in applications containing loops. We simulate the architecture and show that the APVLIW processor performs significantly better than the VLIW processor for a wide range of cache sizes and across various numerical benchmark applications.

키워드 : 명령어 수준의 병렬성(ILP), VLIW, APVLIW, 독립적인 명령어 스케줄링(Independent Instruction Scheduling)

1. 서 론

프로세서 구조에 대한 연구는 클럭 속도를 높이는 방법에서 동시에 여러 명령어들을 중첩하여 실행시킴으로써 단위시간당 처리하는 명령어의 수를 증가시키는 병렬처리 기법으로 전환·발전되었다[1, 2]. 이에 따라서 프로그램에 내재한 명령어 수준의 병렬성(ILP : Instruction Level Parallelism)을 추출하여 실행하는 다양한 ILP 프로세서 기술이 광범위하게 연구되고 있다.

수퍼스칼라 구조(superscalar)[2-4]는 명령어할당 과정을 실행 시에 하드웨어가 수행하므로 스케줄러(scheduler)를 복잡한 논리회로(logic circuit)로 구성하여야 하지만 목적코드의 형태는 기존의 순차코드와 동일하다. 그에 비하여 VLIW 구조(Very Long Instruction Word)[2, 4-13, 16-20, 22]는 명령어할당 과정을 컴파일 과정에서 확정하고 병렬처리가 가능한 명령어들을 하나의 긴명령어로 구성하여 목적코드를 구성한다. 따라서 VLIW 구조용으로 목적코드를 구성할 때, 긴명령어내의 명령어간에 자료종속성이 존재하거나 가용한 연산처리기가 사용중인 경우에는 NOP(No OPeration)과 LNOP (Long NOPs) 명령어가 긴명령어에 존재하게 된다. 여기서 LNOP은 NOP 명령어들로 구성된 긴명령어를 의미한다. 즉, VLIW 코드에는 NOP이 다수 포함되어서 실행시에 수퍼스칼라 구조에 비하여 캐시 미스율이 높은 단점이 있다. SVLIW 구조(Superscalar VLIW)[14, 15, 21]는 VLIW 구조용 목적코드에서 LNOP이 존재하는 경우에 그 LNOP들을 제거한 목적코드를 생성한다. 생성된 목적코드를 실행하기 위해서, SVLIW 프로세서 구조는 시스템내의 스케줄링 장치가 긴명령어의 실행 여부를 실시간에 결정하도록 함으로써 VLIW

※ 이 논문은 한국과학재단의 해외 Post-doc. 연수지원에 의하여 연구된 것입니다.
† 정 회 원 : 천안외국어대학교 컴퓨터정보과 교수
†† 종신회원 : 충북대학교 컴퓨터과학과 교수
논문접수 : 2001년 10월 24일, 심사완료 : 2002년 3월 18일

구조의 경우와 동일한 효과를 얻는 형태이다. 그러나, SVLIW 구조의 경우에도 기명령아내에 존재하는 NOP까지 제거하는 것은 불가능하다. 또한명령어 스케줄링 측면에서, VLIW 프로세서와 SVLIW 프로세서는 목적코드를 긴명령어 단위로 스케줄링한다. 즉, 실행을 원하는 긴명령어는 현재 실행중인 긴명령어내의 모든 명령어들이 실행단계를 종료한 후에야 실행단계로 진입할 수 있으므로 성능향상에 제약을 받는다[15, 21].

본 논문에서는 기존 VLIW 코드내에 포함된 긴명령어 LNOP 뿐 아니라 SVLIW 코드에 포함된 명령어 NOP까지도 제거하여 적극적으로 압축한 형태로 생성된 목적코드를 실행할 수 있는 *APVLIW*(Aggressively Packed VLIW) 프로세서 구조를 제안하였다. APVLIW 프로세서내의 모든 연산처리기는 목적코드내에 포함된 자료종속성 정보를 이용하여 긴명령어들내의 각 명령어를 독립적으로 스케줄링 할 수 있다. 이와 같은 명령어들의 독립적인 스케줄링을 위해서 APVLIW 프로세서는 여러 개의 연산처리기와 동적스케줄러의 쌍들로 구성된다. 본 논문에서는 실험을 통하여 APVLIW 프로세서가 기존의 VLIW 프로세서나 SVLIW 프로세서에 비하여 성능이 향상됨을 확인하였다.

수퍼스칼라 프로세서 구조가 ILP 추출에 효과적인 구조라고 하더라도 수퍼스칼라 프로세서는 실시간에 대부분의 병렬처리 작업을 수행하므로 매우 복잡한 논리회로를 요구할 뿐만 아니라 과도한 실시간 오버헤드(run-time overhead)를 요구하는 등 성능향상에 큰 제약조건을 지닌다[20, 21, 23, 24]. 결과적으로 본 논문에서는 수퍼스칼라 프로세서를 고려하지 않았다. APVLIW 프로세서는 대부분의 병렬처리 작업을 컴파일 시에 수행하며 실시간에는 명령어 할당을 위한 단순한 스케줄링 장치만이 필요하다. 그러므로 APVLIW 프로세서는 수파스칼라 프로세서와 비교할 경우 단순한 논리회로로 구성될 뿐 아니라 적은 비용으로도 구현이 가능하다.

본 논문의 제 2절에서는 기존 ILP 프로세서 구조와 APVLIW 구조에서의 명령어 인출슬롯과 실행이미지를 비교하였다. 제 3절에서는 APVLIW 프로세서 구조를 설계하고 동기화유지 방안 및 파이프라인 구성을 설명하였다. 제 4절에서는 APVLIW 프로세서와 기존 ILP 프로세서의 시뮬레이션 시스템을 구축하고 다양한 벤치마크 프로그램들에 대한 실험을 수행하였다. 마지막으로 제 5절에서는 논문의 결론을 맺고 향후 계획을 기술하였다.

2. 명령어 수준의 병렬성

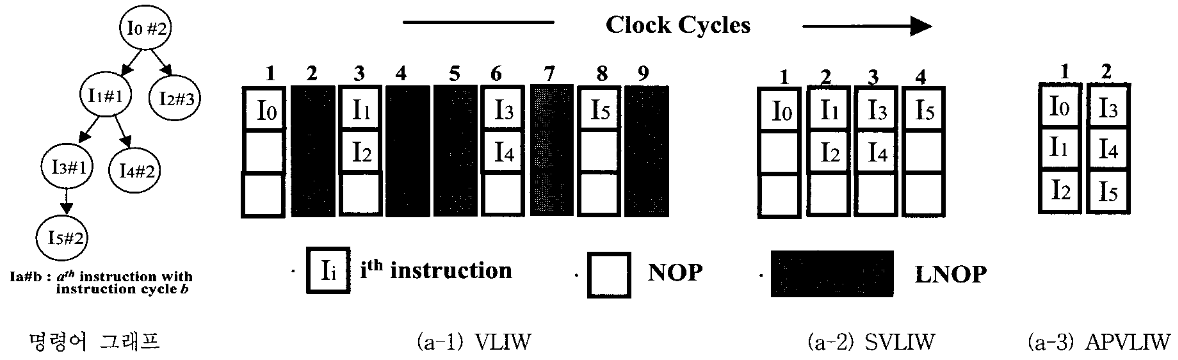
(그림 1)은 여러가지 ILP 프로세서에서 명령어 그래프로부터 생성한 목적코드를 실행하기 위하여 명령어를 인출하는 과정과 인출과정을 통한 실행이미지을 보여 준다. (그림 1)의 명령어 그래프에서 노드(node)는 명령어 I_i 및 실행사이클이 점유하는 길이를 나타내며 이를 '#'로 구분하였다. 또한, 에지(edge)는 노드간 자료종속성 관계를 나타낸다. (그림 1)(a-1)(a-2)(a-3)의 명령어 인출과정에서 사이클 별로 하나의 긴명령어는 수직방향으로 세 개의 사각형으로 표시된다. 본 실험에서 모든 명령어는 인출(F), 분석(D), 실행(EX), 그리고 쓰기(WB)인 네 단계의 파이프라인을 거치면서 실행된다고 가정한다.

(그림 1)(a-1)은 VLIW 프로세서의 명령어인출 과정을 보인다. VLIW 코드는 명령어간에 발생한 자료 종속성이나 자원충돌을 해결하기 위해서 많은 수의 LNOP과 NOP들을 포함한다. VLIW 프로세서는 (그림 1)(b-1)과 같이 스케줄링된 긴명령어내의 모든 명령어들이 실행을 종료한 후에 그 다음 긴명령어를 실행단계로 진입시킬 수 있다. 이때 긴명령어의 실행사이클은 긴명령어내의 명령어 가운데 실행사이클이 가장 긴 명령어에 의해서 결정된다.

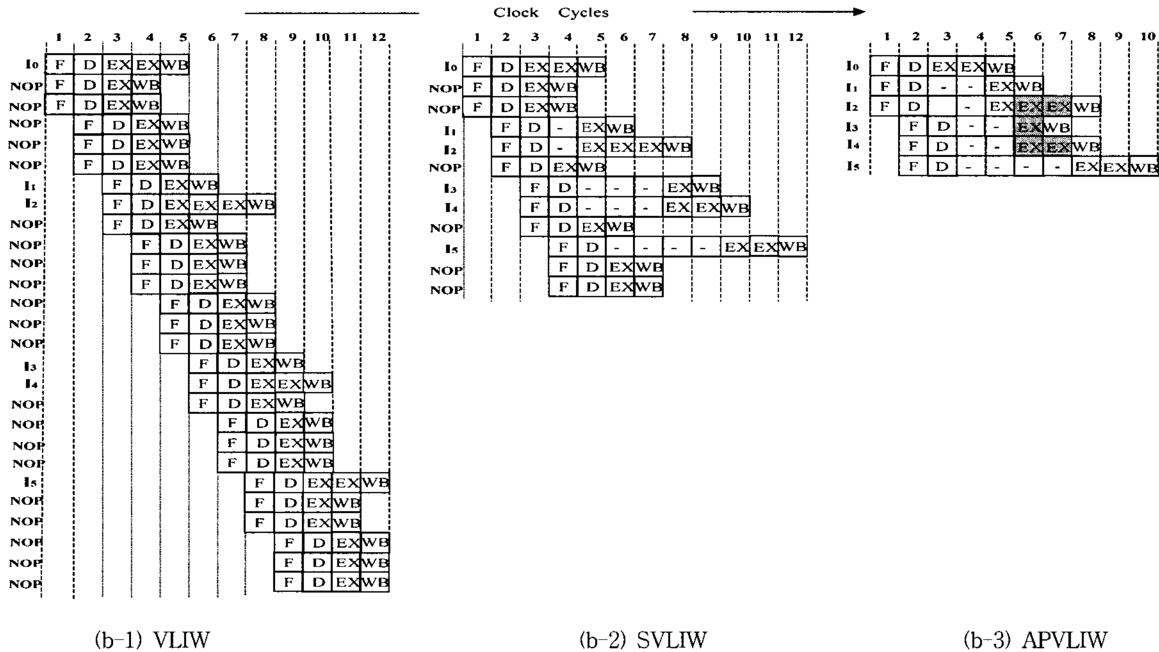
SVLIW 프로세서는 (그림 1)(a-2)의 인출슬롯과 같이 (그림 1)(a-1)에 포함된 긴명령어 LNOP이 모두 제거된 코드를 생성한다. 본 논문에서는 (그림 1)(a-1)에 보인 형태의 목적코드에 비하여 (그림 1)(a-2)의 목적코드가 짧기 때문에 이를 PPVLIW(Passively Packed VLIW) 코드라고 부른다. SVLIW 구조에서 실행 중에 자료종속성이 존재하거나 자원충돌이 발생할 경우에는 (그림 1)(b-2)에서 "-"로 표시된 바와 같이 실행이 지연된다. VLIW 프로세서와 동일한 스케줄링 기법을 적용하는 SVLIW 프로세서는 VLIW 구조와 동일한 실행사이클을 요구한다.

(그림 1)(a-3)은 (그림 1)(a-2)에서 나머지 NOP도 제거한 형태의 목적코드를 구성하고 긴명령어 단위로 인출하는 과정을 보인다. 이 코드는 프로세서가 실시간에 독립적으로 명령어의 실행 여부를 결정할 수 있도록 자료종속성 정보를 포함하여야 한다. 본 논문에서는 (그림 1)(a-3)의 목적코드를 *APVLIW(Aggressively Packed VLIW)* 코드라 하며 이 코드를 실행하는 구조를 APVLIW 프로세서 구조라고 부른다. 그림과 같이 각 연산처리기는 자료종속성 정보를 이용하여 명령어들을 독립적으로 실행할 수 있다. 즉, 그림의 회색 부분과 같이 여섯 번째 사이클에서 명령어 I_2, I_3, I_4와 일곱 번째 사이클에서 명령어 I_2, I_4가 동시에 실행한다.

이상의 관찰로부터 APVLIW 코드를 실행하는 APVLIW 구조는 기존의 VLIW 또는 SVLIW 구조에 비하여 높은 성능향상을 기대할 수 있다. 이와 같은 성능향상은 제안된 APVLIW 프로세서내 모든 연산처리기들이 다른 연산처리기의 실행여부와 상관없이 긴명령어내에 속하는 각 명령어를 독립적으로 스케줄링할 수 있기 때문이다. 그러나 이와 같은 명령어 스케줄링을 수행하기 위해서, APVLIW 구조는 목적코드내에 명령어간의 자료종속성에 관한 정보가 포함되어야 할 뿐만 아니라 이 정보를 이용하여 실행시에 각



명령어 그래프 (a-1) VLIW (a-2) SVLIW (a-3) APVLIW



(b-1) VLIW (b-2) SVLIW (b-3) APVLIW

(그림 1) ILP 프로세서별 명령어인출 및 실행이미지

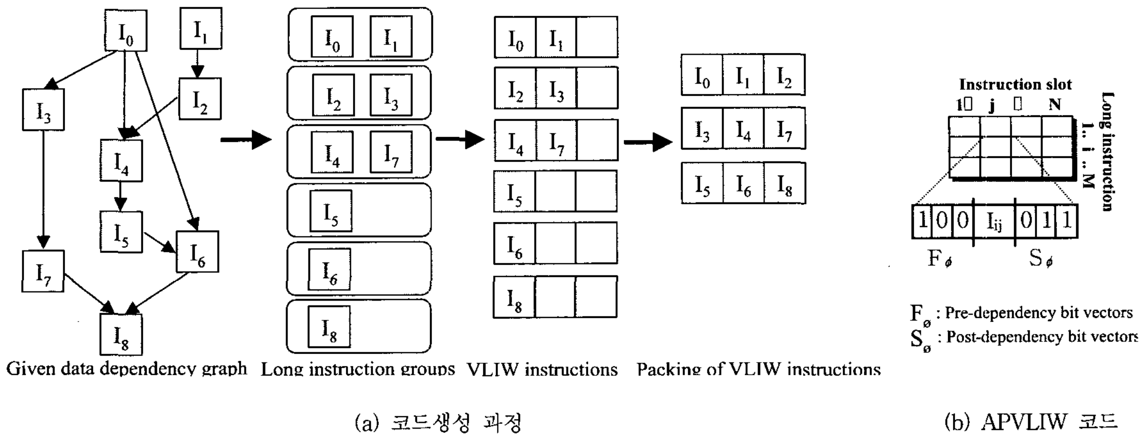
명령어가 실행 단계로 진입할 것인가 여부를 결정하는 장치가 추가되어야 한다.

3. APVLIW 프로세서 모델

3.1 APVLIW 코드 구성

APVLIW 프로세서 구조에서 인출된 긴명령어내의 각 명령어는 연산처리기에 할당된 후에 실행중인 명령어나 동시에 수행될 명령어와 자료종속성 관계가 존재하면 자료종속성 관계가 있는 이전 명령어들에게 할당된 연산처리기들의 실행 사이클이 종료될 때까지 실행할 수 없다. 이 과정에서 각 연산처리기는 디코딩 사이클을 수행하는 동안 다음 사이클에서 명령어를 실행 단계로 진입시킬 것인가를 결정하여야 한다. 따라서 각 명령어에는 자신을 실행할 연산처리기가 실행 단계 진입 여부를 결정하는데 사용될 정보가 기록되어야 한다.

APVLIW 코드는 (그림 2)(a)와 같이 VLIW 코드 생성 단계와 2) 명령어압축 단계를 수행한 후에 생성된다. APVLIW 코드의 생성과정은 다음과 같다. 먼저 APVLIW 컴파일러는 주어진 자료종속성 그래프로부터 긴명령어들로 구성되는 VLIW 코드를 생성한다. 동일한 긴명령어를 구성하는 명령어 간에는 자료종속성이나 자원충돌이 없으므로 동시에 실행할 수 있다. 이때 긴명령어내에 사용하지 않는 인출슬롯은 NOP 명령어로 채워진다. 명령어압축 단계에서는 생성된 VLIW 코드의 사용하지 않은 인출슬롯에 유용한 명령어들을 채우므로 각 명령어간에는 자료종속성이나 자원 충돌이 존재할 수 있다. 마지막으로 명령어간 동기와 독립적인 스케줄링을 위해서, 명령어마다 이전(prior) 명령어와 이후(subsequent) 명령어에 관한 자료종속성 정보가 삽입된다. 이때 종속성 레벨(dependency level)을 유지하기 위해서 종속성레벨 i 인 명령어들은 종속성레벨 $i+1$ 인 명령어들보다 항상 이전에 배치되어야 한다. APVLIW 컴파일러는 최종적으로 긴명령어들로 구성되는 APVLIW 코드를 생성하는데 이러한 컴파일러 기술은 기존 VLIW 컴파일러 기술을 적극적으로 활



(a) 코드생성 과정 (b) APVLIW 코드

(그림 2) APVLIW 코드 구성

용하는 장점을 지닌다.

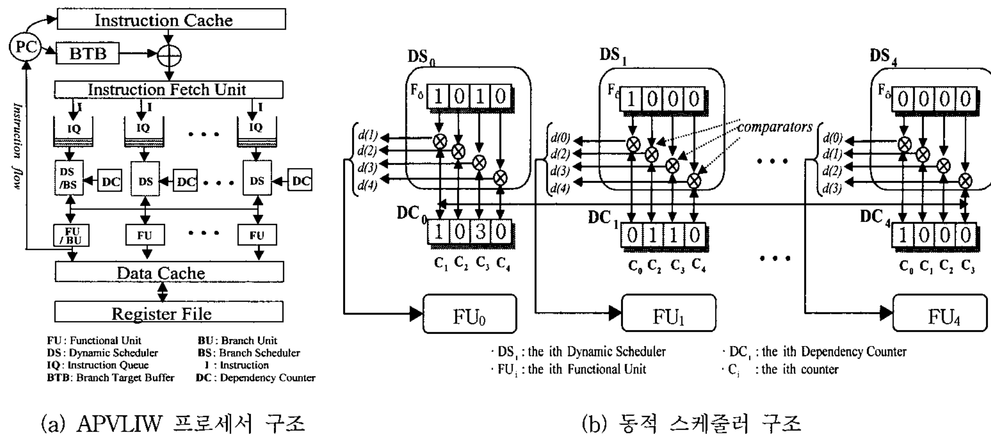
(그림 2)(b)는 APVLIW 코드 구조를 보여 준다. 그림에서 긴명령어는 네 개의 명령어들로 구성되며 각 명령어는 선행종속성 정보 F_ϕ, 명령어 I_{ij}, 그리고 후행종속성 정보 S_ϕ를 가진다. 여기서 F_ϕ는 I_{ij}의 이전 명령어들을 실행하는 연산처리기들에 대한 정보를 의미한다. I_{ij}는 i번째 긴명령어에 속하는 j번째 명령어를 의미한다. S_ϕ는 I_{ij}의 이후 명령어들을 실행할 연산처리기들에 대한 정보를 의미한다. APVLIW 컴파일러는 연산처리기에 대한 정보를 저장하기 위해서 F_ϕ와 S_ϕ를 각각 $(n-1)$개의 비트들로 구성된 비트벡터(bit vector)로 구성하고 연산처리기별로 한 비트를 할당한다. 이때 n은 연산처리기 개수이다. 예를 들어 I_{ij}의 선행종속성 정보 F_ϕ를 구성할 때 I_{ij}가 이전 명령어 $I_{lk}(k<j \text{ if } l=i;\ k=1,\cdots,n \text{ if } l<i)$와 자료종속성 관계이면 I_{lk}를 실행하는 연산처리기 FU_k용 비트는 1로 세트된다. 그렇지 않으면, 그 비트는 0으로 리셋된다. 이때 F_ϕ와 S_ϕ에는 I_{ij}를 실행하는 FU_j의 정보를 저장할 필요가 없으므로 FU_j용 비트는 비트 벡터에서 생략된다. 결과적으로 명령어당 추가되는 비트의 길이는 $2(n-1)$배가 된다. 이러한 정보의 추가는 궁극적으로 목적코드 증가를 초래하지만 VLIW 코드에 비해서 APVLIW 코드의 크기가 작다[21-24].

3.2 APVLIW 프로세서 구조

(그림 3)은 APVLIW 프로세서 구조를 보여준다. APVLIW 프로세서는 다수의 연산처리기(FU : Functional Unit)와 동적 스케줄러(DS : Dynamic Scheduler)의 쌍들, 명령어큐들(IQ : Instruction Queues)과 종속성 카운터들(DC : Dependency Counters), 명령어캐시, 데이터캐시, BTB(Branch Target Buffer) 및 레지스터 파일로 구성된다. IQ는 긴명령어로부터 분할된 명령어를 적재하고 연관된 DS에게 제공한다. 각 DC는 실행할 명령어의 자료종속성 여부를 검사하기 위해서 다른 FU에서 실행된 명령어(또는 실행중인 이전 명령어)들의 후행종속성 정보 S_ϕ를 저장한다. S_ϕ를 저장하기 위해서, 모든 DC는 $n-1$개의 카운터(counter)들로 구성된다. 각 카운터는 하나의 FU를 지칭하며 그 FU에서 실행된 명령어(즉, 현재 FU에서 실행할 명령어들과 종속성관계가 있는 이전 명령어)들의 개수를 센다. DS는 DC를 이용하여 매 사이클마다 다음에 실행할 명령어를 연산처리기에 할당시킬 것인지, 혹은 자원 충돌이나 자료종속성으로 인하여 실행을 지연시킬 것인지를 결정한다. 그 밖에 APVLIW 프로세서는 분기명령어 예측을 위해서 BTB 구조를 사용한다 [6, 9].

(그림 3)(b)는 동적 스케줄러(DS) 구조를 보여준다. 그림과 같이 각각의 DS는 매 사이클마다 다음에 실행할 명령어의 F_ϕ와 연관된 DC내 카운터 값을 비교하여 자원 충돌이나 자료종속성 여부를 분석한다. 즉, F_ϕ내 비트위치 γ가 1이면 각 DS는 연관된 DC내 카운터(γ와 동일한 위치)의 값 δ을 검사한다. 이때 δ=0은 자료종속성 관계가 있는 이전 명령어가 실행을 종료하지 않았음을 의미하므로 $d(i)$는 0. 그렇지 않으면 이전 명령어의 실행이 종료되었음을 의미하므로 $d(i)$는 1. 이와 같이 각 DS는 F_ϕ와 DC 값을 비교하여 종속성 관계가 있는 이전 명령어들의 실행이 모두 종료되었음(모든 $d(i)$가 1)을 확인한 후에 명령어를 연산처리기에 할당한다. 동시에 각 DS는 DC에 반영되어 있는 이전 명령어들의 후행종속성 정보 S_ϕ를 제거하기 위해서 F_ϕ내 세트된 비트와 동일한 위치의 카운터 값을 1만큼 감소시킨다.

APVLIW 프로세서의 명령어실행 방식은 (그림 3)(a)에서 각 FU의 앞에 위치한 IQ들로 인하여 순차적(in-order)으로 보이나 서로 다른 IQ내의 각 명령어들은 매 사이클마다 연관된 DS의 분석과정을 통하여 비순차적(out-of-order)으로 수행된다. 이와 같은 독립적인 스케줄링은 각 FU마다 구비된 DS와 DC에 의해서 가능하다.



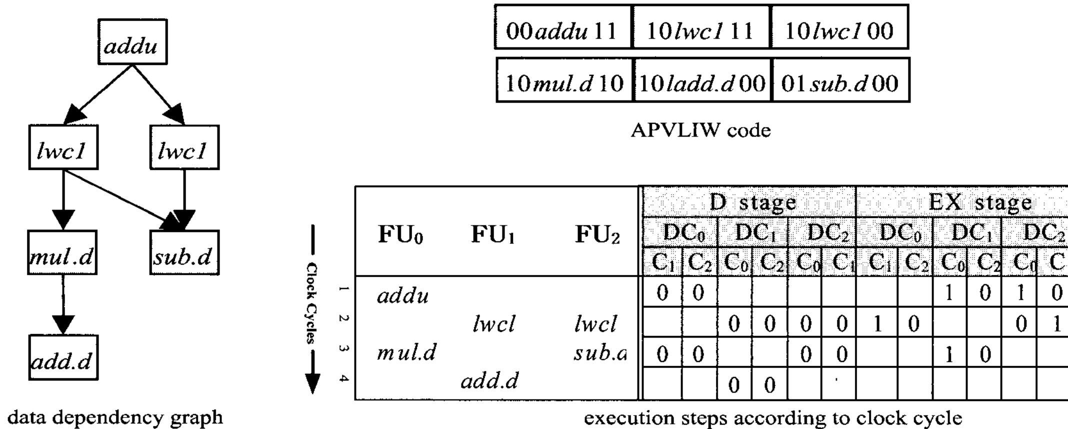
(a) APVLIW 프로세서 구조 (b) 동적 스케줄러 구조

(그림 3) APVLIW 프로세서 구조

3.3 명령어 파이프라인 절차

APVLIW 프로세서에서 모든 명령어는 인출, 분석, 실행, 그리고 쓰기의 네 단계인 명령어 파이프라인에 의해서 수행된다. 각 파이프라인은 실행 단계 외에는 매 단계를 한 사이클에 수행한다. 인출(F : Fetch) 단계에서 인출 유니트는 명령어 캐시로부터 하나의 긴명령어를 인출하고, 인출한 긴명령어를 명령어들로 분할하여 IQ에 적재한다. 분석(D : Decode) 단계에서 분석유니트는 각 IQ에 적재된 명령어들을 가져와 분석하여 명령어 실행에 필요한 정보를 알아낸다. 동시에 모든 DS는 앞 절에서 설명한 바와 같이 명령어간 자료종속성 여부를 분석한 후 DC 값에 반영된 이전 명령어들의 S_ϕ을 제거하기 위해서 DC내 카운터(F_ϕ내 세트된 비트와 동일한 위치) 값들을 1만큼 감소시킨다. 그 후 연관된 FU에 다음에 실행할 명령어를 할당한다. 실행(EX : EXecute) 단계에서는 각 FU가 분석 단계로부터 전달된 명령어를 실행한다. 동시에 각 FU는 마지막 실행 사이클 동안에 실행중인 명령어의 S_ϕ를 이용하여 자료종속성 관계가 있는 이후 명령어들에게 실행중인 명령어의 종료 사실을 알려서 실행단계에 진입하지 못하는 이후 명령어들이 다음 사이클에서 실행단계로 진입할 수 있도록 한다. 즉, 각 FU는 자신의 DC에 그 정보를 저장하기 위한 방안으로 DC내 카운터(종료사실을 전달하는 FU용 카운터) 값을 1만큼 증가시킨다. 마지막으로 쓰기(WB : Write Back) 단계에서는 실행 결과 값을 레지스터 파일에 저장한다. 이상과 같이 APVLIW 프로세서는 분석 단계와 실행 단계에서의 카운터 값을 이용하여 비순차적으로 명령어들을 실행할 수 있다. 본 실험에서는 이 과정을 수행하기 위해서 실행 단계가 분석 단계를 제어하도록 설계하였다.

APVLIW 프로세서는 분기명령어를 예측하기 위하여 동적예측(dynamic strategy) 기법인 BTB 구조를 이용한다. 분기명령어를 실행할 경우, 프로세서는 BTB에 저장된 분기예측 정보를 이용하여 곧바로 다음 사이클에 타겟명령어를 인출하도록 한다. 그 후에 분기예측이 참으로 판명될 때까지, 분기가 발생되어 연속으로 실행된 명령어들의 결과값과 분기문 발생이후에 변경되는 모든 DC 값들을 임시 기억장소에 저장한다. APVLIW 프로세서는 임시 기억장소로써 각각 레지스터 파일의 복사본과 DC의 복사본을 운영



(그림 4) 명령어 실행과정

한다. 즉, 분기가 예측되는 시점부터 DC 값들은 임시 기억장소에 있는 DC의 복사본에 복사되어 분기예측이 확정될 때까지 실행결과 값을 DC의 복사본에 일시 저장한다. 만약 분기예측이 참이면 그동안 임시 기억장소에 저장된 값들은 각각 레지스터 파일과 DC에 기록된다. 만약 분기예측이 거짓이면 각 IQ도 잘못된 분기 예측 후에 인출된 명령어들을 적재하고 있으므로 각 IQ는 임시 기억장소와 더불어 클리어(clear) 된다. 마지막으로 APVLIW 프로세서는 분기결과를 이용하여 분기예측 정보를 변경한 후 BTB에 저장한다.

(그림 4)는 주어진 자료종속성 그래프로부터 생성된 APVLIW 코드에 대한 실행과정을 보여준다. 명령어 *addu*의 F_ϕ가 00이므로 연산처리기 FU_0는 먼저 *addu*를 실행한다. 동시에 *addu*의 S_ϕ가 11이므로 FU_0는 DC_1과 DC_2의 첫 번째 카운터 C_0 (FU_0를 가르키는)을 1씩 증가시킨다. 다음번 분석단계에서 DS_1과 DS_2는 각각 실행할 명령어 *lwc1*의 F_ϕ(1로 세트된 첫 번째 위치)와 자신과 연관된 *DC*내 카운터를 비교한다. 이때 *DC*내 각 카운터 C_0(F_ϕ내 1과 동일한 위치)이 1 이상이면 *addu*의 실행이 종료되었음을 의미하므로 각 *DC*내의 카운터 C_0(F_ϕ내 1로 세트된 위치) 값을 하나씩 감소하면서 동시에 FU_1과 FU_2는 각각 명령어 *lwc1*을 수행한다. 이와같은 *DC*내 카운터 값의 갱신과정은 실행전에 *DC*에 저장된 *addu*의 S_ϕ를 제거하기 위해서 요구된다.

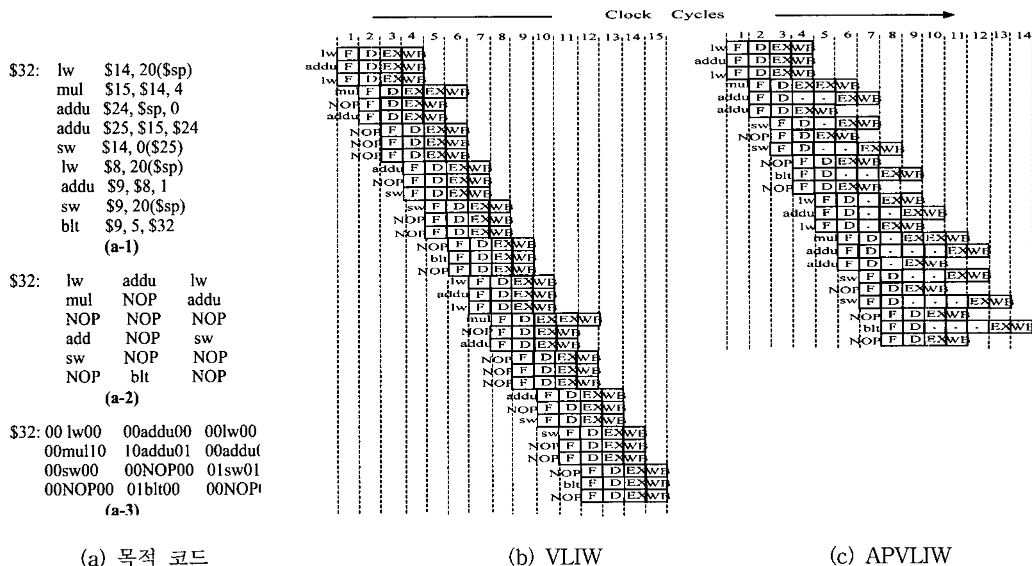
3.4 루프의 실행 효과

루프 실행시에 APVLIW 프로세서는 루프의 서로 다른 반복 실행으로부터 인출된 명령어들 중에서 자료종속성 관계가 없는 명령어들을 동시에 스케줄링할 수 있다.

(그림 5)는 VLIW와 APVLIW 프로세서에서 루프 프로그램을 실행할 때 분기예측이 적중한 경우의 i와 $i+1$번째 반복실행에 대한 실행이미지이다. (그림 5)(a-1)의 MIPS 어셈블리 코드로부터 VLIW 코드와 APVLIW 코드를 생성하면 각각 (그림 5)(a-2) 및 (그림 5)(a-3)과 같다. 이때 각각의 긴명령어는 세개의 명령어들로 구성되며, 명령어 *mul*의 실행사이클은 두 사이클이고 나머지 명령어들의 실행사이클은 한 사이클이다. 모든 명령어는 인출(F), 분석(D), 실행(EX), 그리고 쓰기(WB)의 파이프라인에 의해서 실행된다.

VLIW 프로세서에서 (그림 5)(a-2)의 코드를 실행할 경우 (그림 5)(b)와 같이 매 사이클마다 하나의 긴명령어를 인출하여 순차적으로 긴명령어를 실행한다. 그에 비하여 (그림 5)(c)의 APVLIW 프로세서에서 각 연산처리기는 자료종속성 정보를 이용하여 (그림 5)(a-3)의 각 명령어를 독립적으로 스케줄링한다. 예를 들어, 세 번째 긴명령어내의 명령어 *sw*와 두 번째 긴명령어내의 명령어 *addu*는 각각 다른 사이클에서 인출되었더라도 여섯 번째 사이클에서 동시에 실행한다. 또한, *i*번째 반복에서 인출된 명령어 *blt*와 *i+1*번째 반복에서 인출된 명령어 *lw*는 다른 반복으로부터 인출되었더라도 여덟 번째 사이클에서 동시에 실행한다.

(그림 5)에서의 성능향상은 APVLIW 프로세서가 루프의 서로 다른 반복으로부터 인출된 명령어들 중에서 자료종속성 관계가 없는 명령어들을 동시에 스케줄링을 할 수 있기 때문이다. 따라서 루프의 반복 횟수가 클수록, APVLIW 프로세서는 VLIW 프로세서에 비하여 적은 실행사이클을 요구한다. 또한, (그림 6)의 실행결과에는 반영되지 않았으나 APVLIW 프로세서는 VLIW 프로세서에 비하여 평균 명령어 인출사이클을 줄일 수 있다. 즉 APVLIW 프로세서는 압축된 형태의 긴명령어를 매 사이클마다 인출하기 때문이



(a) 목적 코드 (b) VLIW (c) APVLIW

(그림 5) 분기예측이 적중한 경우의 명령어 실행이미지

다. 또한, APVLIW 프로세서는 적은 크기의 목적코드를 실행하기 때문에 상대적으로 낮은 캐시 미스율을 가진다[21].

4. 실험 및 고찰

4.1 시뮬레이션 시스템 및 실험환경

본 연구에서는 APVLIW 프로세서의 성능을 평가하기 위해서, 기존 VLIW 코드를 수행하는 VLIW 프로세서, PPVLIW 코드를 수행하는 SVLIW 프로세서, 그리고 APVLIW 프로세서의 실행 모델을 확인할 수 있는 시뮬레이션(simulation testbed) 시스템을 구현하였다.

시뮬레이션 시스템은 목적코드를 생성하는 컴파일러(compiler)부와 생성된 목적코드를 실행하는 시뮬레이터(simulator)부로 구성된다. 컴파일러 부는 어셈블러(Mipspro C++ compiler), 메크로 확장기(macro expander), 그리고 병렬처리기(parallelizer)로 구성된다. 어셈블러는 C 언어로 구성된 벤치마크 프로그램을 입력받아서 MIPS R4000 어셈블리 코드를 생성한다[17]. 이때 어셈블러는 최적화 플래그 -O와 어셈블리코드 생성 플래그 -S를 이용하여 레지스터 재명명(register renaming), 불필요한 수식 제거(common subexpression elimination)의 최적화 과정을 거친 MIPS 코드를 생성한다. 메크로 확장기는 매크로 명령어를 MIPS 코드로 확장한다. 생성된 MIPS 코드를 입력받은 병렬처리기는 기존의 VLIW 컴파일러 기술을 바탕으로 기본 블록(basic block)에서의 ILP 추출을 기본으로 한 루프확장(loop unrolling), 함수삽입(function inlining) 및 분기문 예측(branch prediction)의 컴파일링 기법을 이용하여 기본블록의 확장된 범위에서 ILP 추출을 수행하여 프로세서별 목적코드를 생성한다[6, 9-11]. 이때 $VLIW_c$은 VLIW 코드, $VLIW_{PP}$은 PPVLIW 코드, $VLIW_{AP}$은 APVLIW 코드를 의미한다.

Fixed Parameters	
processor pipeline	four-stage(F, D, EX, WB)
decoded instruction size	4 bytes
integer instruction latency	1 cycle
floating point instruction latency	1~32 cycle(depend on instruction)
data cache size	perfect(no miss penalty)
cache mapping method	derect mapped
cache replacement policy	LRU(Least Recently Used)
Variable Parameters	
Parameter	*Default Value*
a number of integer unit	2
a number of floating-point unit	2
next long instruction miss penalty	3
instruction cache size	16k bytes

(그림 6) 입력 파라메타 값들

일반적으로 VLIW 프로세서의 성능은 VLIW 컴파일러에 의해서 결정된다. VLIW 컴파일러가 응용 프로그램으로부터 높은 ILP를 추출할수록 VLIW 프로세서는 그에 비례하여 높은 성능향상과 낮은 캐시미스율을 얻을 수 있다. 본 논문에서 제안된 APVLIW 프로세서는 VLIW 코드로부터 압축된 목적코드를 생성하므로 VLIW 컴파일러로부터 발생하는 장점을 이용할 수 있다. 시뮬레이터 부는 컴파일러 부에서 생성한 세 가지 목적코드를 입력으로 받아서 프로세서 시뮬레이터에서 목적코드별로 실행을 한 후에 총 실행사이클을 출력한다. 본 실험에서는 다양한 실험환경을 구현하기 위해서 (그림 6)과 같이 파라메타 값들을 입력받도록 설계하였으며, 그 입력 파라메타값은 크게 고정 값과 변동값으로 나누어진다.

<표 1>은 본 실험에서 사용되는 벤치마크 프로그램들과 명령어 유형별 분포 비율(I/F : Integer instruction/Floating-point instruction)를 나타낸다. 본 실험에서는 실행사이클이 길고 비정형적인 실수명령어 분포가 높은 수치계산(numeric) 프로그램을 벤치마크로 선정하였다. 이것은 실수명령어 분포가 높은 프로그램을 실행할 경우에 APVLIW 프로세서의 스케줄링이 효과적이라고 판단되기 때문이다. 벤치마크에서 사용되는 모든 데이타집합은 double precision 타입이다. <표 2>는 각 벤치마크로부터 생성된 $VLIW_c$에 대한 $VLIW_{PP}$와 $VLIW_{AP}$의 크기 비율(ratio)를 보여준다. 비록 $VLIW_{AP}$가 많은 비트들로 구성된 자료종속성 정보를 포함하더라도 $VLIW_{AP}$는 $VLIW_c$보다 45%가 작으며 $VLIW_{PP}$와는 동일한 크기의 목적코드를 가짐을 알 수 있다.

〈표 1〉 벤치마크 프로그램 집합

Benchmarks	Description	I/F(%)
LIVERMORE	Do loop for various kernel operations	65.3/34.7
MM	Matrix Multiply using floating point instructions	68.4/31.6
CLINPACK	Set of linear algebra subroutine	75.7/24.3
WHETSTONE	Loop instructions for arithmetic computation	65.6/34.4
FFT	Matrix fourier Transformation	43.3/56.7

〈표 2〉 목적 코드 크기 비교

Benchmarks	$VLIW_C$	$VLIW_{PP}$	$VLIW_{AP}$
LIVERMORE	1	0.723	0.725
MM	1	0.568	0.591
CLINPACK	1	0.673	0.673
WHETSTONE	1	0.438	0.385
FFT	1	0.385	0.400
AVERAGE	1	0.557	0.554

4.2 실험 및 성능분석

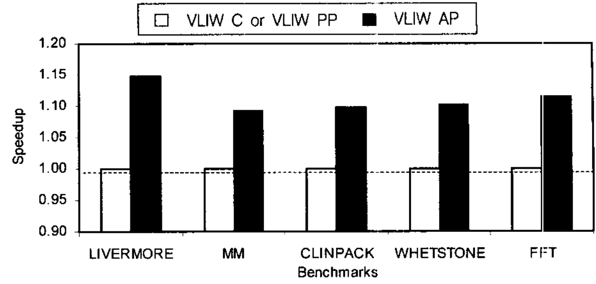
본 절에서는 시뮬레이션 시스템에서 벤치마크 프로그램들을 실행할 경우 각 프로세서별 총 실행사이클을 측정하

고 측정된 실행사이클을 이용하여 VLIW 프로세서의 성능에 대한 SVLIW 프로세서와 APVLIW 프로세서의 상대적인 성능비(speedup)을 비교・분석하였다.

4.2.1 스케줄링 기법에 따른 효과

(그림 7)은 프로세서별 명령어 스케줄링 기법을 비교하기 위해서 VLIW(또는 SVLIW) 프로세서에 대한 APVLIW 프로세서의 성능비를 보여준다. 본 실험에서는 스케줄링 기법만을 비교하기 위해서 명령어캐시로 인한 효과는 실험에서 제외하였다. 즉, 명령어캐시 크기는 무한(perfect)하다고 가정하므로 실행중에 캐시미스는 발생하지 않는다. 본 실험에서는 실험시간을 단축하기 위해서 벤치마크에 포함된 루프의 반복 횟수를 일정한 비율로 줄여 실험하였다.

(그림 7)과 같이 무한한 크기의 명령어캐시를 사용할 경우 APVLIW 프로세서의 장점인 캐시효과, 낮은 캐시미스율과 적은 인출사이클,이 제거되더라도 APVLIW 프로세서는 VLIW 프로세서에 비하여 9%~13% 범위의 성능향상을 얻을 수 있다. 이러한 성능향상은 APVLIW 프로세서의 독립적인 명령어 스케줄링에 의해 얻어진다. 독립적인 스케줄링 기법은 VLIW 프로세서와 SVLIW 프로세서의 경우와 비교할 경우 긴명령어들 사이의 대기시간(waiting time)을 줄일 수 있다. 반면에 VLIW 프로세서와 SVLIW 프로세서는 실행중인 긴명령어에 포함된 모든 명령어들의 실행이 종료되기 전에 다음 긴명령어가 실행단계로 진입할 수 없다. 또한, (그림 7)에서 VLIW와 SVLIW 프로세서가 동일한 성능을 가짐도 확인하였다.

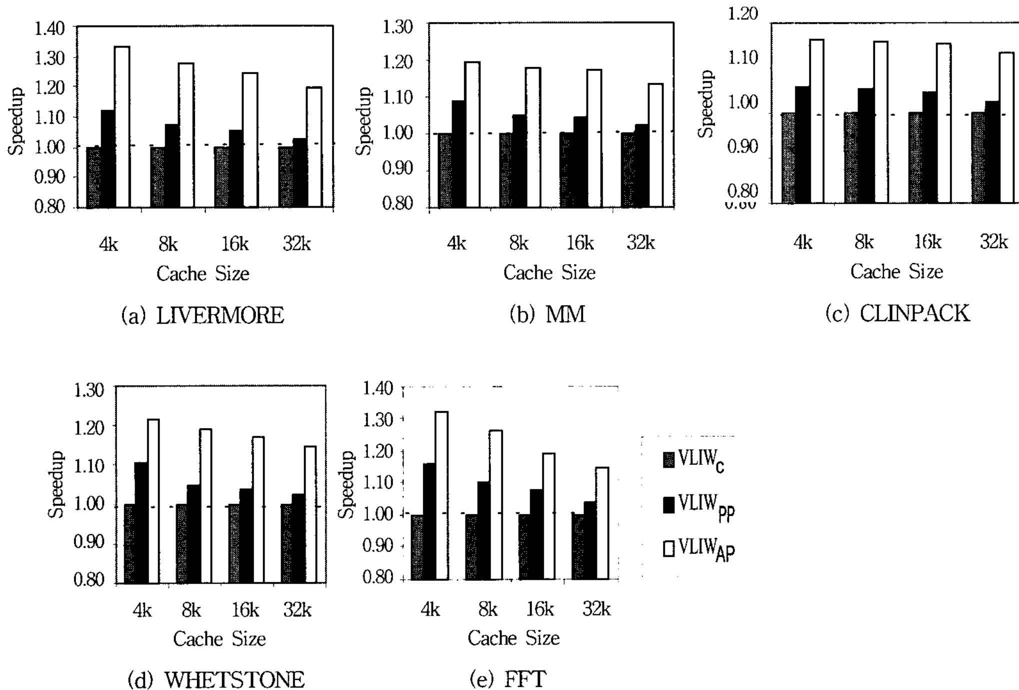


(그림 7) 스케줄링 기법에 따른 성능비 비교

4.2.2 명령어캐시에 의한 효과

(그림 8)는 프로세서별 명령어캐시의 크기변화에 따른 성능을 비교하기 위해서 캐시 크기별로 VLIW 프로세서에 대한 SVLIW 프로세서 및 APVLIW 프로세서의 성능비를 보여준다. 본 실험에서는 명령어캐시의 크기를 4k~32k바이트로 변화시키면서 실험하였고 실험시간을 단축하기 위해서 각 벤치마크 프로그램내에 포함된 루프의 반복 횟수를 일정한 비율로 줄였다.

(그림 8)과 같은 성능향상은 APVLIW 프로세서의 명령어 스케줄링 기법으로 인하여 얻어진다. 성능향상의 다른 주된 요인인 APVLIW 프로세서의 작은 목적코드 크기는 실행시에 평균 명령어인출 사이클을 줄일 수 있을 뿐 아니라 캐시미스율도 낮출 수 있다. (그림 8)을 통하여 VLIW 프로세서는 캐시의 크기가 커질수록 캐시미스율이 낮아지므로 APVLIW 프로세서와의 성능비 차이가 크게 줄어듬을 알 수 있다. 그러나 APVLIW 프로세서는 이미 작은 크기의 목적코드를 가지므로 캐시크기의 증가로 인한 성능향상



(그림 8) 캐시크기의 변화에 따른 성능비 비교

의 효과가 크지 않다. 그러나 캐시 크기가 무한히 증가하더라도 APVLIW 프로세서는 (그림 7)의 결과와 같이 VLIW 프로세서와 SVLIW 프로세서에 비하여 항상 높은 성능비를 가진다.

5. 결론 및 향후 계획

본 논문에서 제안된 APVLIW 프로세서는 수퍼스칼라 프로세서의 독립적인 스케줄링 기법과 VLIW 프로세서의 컴파일시 ILP 추출의 두 가지 장점을 혼합한 프로세서 구조이다. 명령어 스케줄링의 방안으로서, APVLIW 프로세서는 연산처리기와 동적스케줄러의 쌍들을 이용하여 긴명령어내의 각 명령어를 독립적으로 스케줄링한다. APVLIW 프로세서는 특히 루프를 포함한 프로그램을 실행시킬 때 높은 성능향상을 얻을 수 있다. 실험결과를 통해서도 명령어캐시 크기의 변화와 사용된 벤치마크 프로그램들에 상관없이 APVLIW 프로세서가 VLIW 프로세서나 SVLIW 프로세서에 비하여 성능이 향상됨을 확인하였다. 이러한 APVLIW 프로세서의 성능향상은 독립적인 명령어 스케줄링과 작은 크기의 목적코드로 인해서 얻어진다. 실행중에 무한한 크기의 캐시를 사용할 경우 APVLIW 프로세서의 장점인 캐시로 인한 효과, 낮은 캐시미스율과 적은 명령어 인출사이클, 가 상쇄되더라도 APVLIW 프로세서는 VLIW 프로세서에 비하여 9%~13% 범위의 성능향상을 얻을 수 있다. APVLIW 프로세서 구조는 APVLIW 타입의 프로세서 구조와 컴파일러 기술 분야에 대한 새로운 연구영역를 제시하고 있다. 특히, 자료종속성 정보의 최적화, APVLIW 코드 구성에 효과적인 컴파일러 설계, 연산처리기 확장성은 지속적으로 연구되어야 할 분야이다.

참 고 문 헌

[1] Ken Sakamura, "21st-century microprocessors," *IEEE Micro*, pp.10-11, July/Aug., 2000.

[2] Michael J. Flynn, *Computer Architecture*, Jones and Bartlett Publishers, 1995.

[3] P. P. Chang, D. M. Lavery, S. A. Mahlhe, W. Y. Chen, and Wen-Mei. W. Hwu, The Importance of Prepass Code Scheduling for Superscalar and Superpipelined Processors, *IEEE Transactions on Computers*, Vol.44, No.3, pp.353-370, 1995.

[4] Shyh-Kwei Chen, W. Kent Fuchs, and Wen-Mei W. Hwu, "An analytical approach to scheduling code for superscalar and VLIW architectures," *Proc. International Conference on Parallel Processing.*, pp.I258-I292, 1994.

[5] J. A. Fisher, The VLIW machine : A multiprocessor for compiling scientific code, *IEEE Transactions on Computers*, pp.45-53, July, 1984.

[6] Barry Fagin, "Partial Resolution in Branch Target Buffers," *IEEE Computers*, Vol.46, No.10, October, 1997.

[7] Joseph A. Fisher, "Trace Scheduling : A Technique for Global Microcode Compaction," *IEEE Transactions on Computers.*, Vol.C-30, No.7, pp.478-490, July, 1981.

[8] Roger Espasa and Mateo Valero, "Exploiting instruction-and data-level parallelism," *IEEE Micro*, Vol.17, No.5, Sept/Oct., 1997.

[9] S. A. Mahlke, R. E. Hank, J. E. M. McCormick, D. I. August, and W. W. Hwu, A Comparison of Full and Partial Predicated Execution Support for ILP Processors, *Proceedings of the 22th international Symposium on Computer Architectures*, pp.138-150, 1995.

[10] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling : A technique for object code compatibility in VLIW architecture," *Proceedings of 28th International Symposium on Microarchitecture*, March, 1995.

[11] Arthur Abnous and Nader Bagherzadeh, "Pipelining and bypassing in a VLIW processor," *Transactions on Parallel and Distributed Systems*, Vol.5, No.6, pp.658-664, June, 1994.

[12] T. M. Conte and S. W. Sathaye, Dynamic rescheduling ; A technique for object code compatibility in VLIW architecture, *proceedings of the 28th Annual International Symposium on Microarchitecture*, pp.208-218, March, 1995.

[13] Kevin W. Rudd and Michael J. Flynn "Instruction-level parallel processors-dynamic and static scheduling tradeoffs," *Proc. The Second AIZU International Symposium on Parallel Algorithms/Architecture Synthesis.*, pp.74-80, Mar., 1997.

[14] Shusuke Okamoto and Masahiro Sowa, "Hybrid processor based on VLIW and PN-Superscalar," *Proc. PDPTA'96 International Conference.*, pp.623-632, 1996.

[15] Sunghyun Jee and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor," *Journal IEEE Korea Council*, Vol.1, No.1, Dec., 1997.

[16] Susan J. Eggers, Joel S. Emer, Henry M. Levy, and Jack L. Lo, "Simultaneous multithreading," *IEEE Micro*, Vol.17, No.5, Sep., 1997.

[17] *MIPS R4000 Microprocessor User's Manual*, MIPS Computer Systems, Inc., 1991.

[18] B. R. Rau, Dynamically scheduled VLIW processors, *Proceedings the 26th Annual International Symposium on Microarchitecture*, pp.138-148, Mach, 1997.

[19] T. Hara and H. Ando, "Performance comparison of ILP machines with cycle time evaluation," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pp.213-224, March, 1996.

[20] A. F. de Souza and P. Rounce, Dynamically Scheduling VLIW instructions, *Journal of Parallel and Distributed*

Computing, pp.1480-1511, 2000.

[21] Sunghyun Jee and Sukil Kim, A Design of A Processor Architecture for Codes With Explicit data Dependencies, *Proceedings of the tenth SIAM Conference on Parallel Processing for Scientific Computing 2001*, March, 2001.

[22] Erik R. Altman, R. Govindarajan, and Guang R. Gao, A Unified Framework for Instruction Scheduling and Mapping for Functional Units with Structural Hazards, *Journal of Parallel and Distributed Computing*, pp.259-293, 1998.

[23] Sunghyun Jee and Kannappan Palaniappan, "Dynamically Scheduling VLIW Instructions with Dependency Information," *Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures*, IEEE Press, 2002.

[24] Sunghyun Jee and Kannappan Palaniappan, Performance Evaluation For a Compressed-VLIW Processor, *Proceedings of the 17th ACM Symposium on Applied Computing*, March 2002.

지 승 현

e-mail : jees@missouri.edu

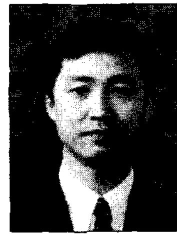
1993년 충북대학교 학사학위 취득

1995년 충북대학교 전자계산학과 석사취득

2000년 충북대학교 박사취득

1999년~현재 천안외국어대학 컴퓨터정보과 교수로 재직중

관심분야 : 병렬처리 컴퓨터구조, 병렬처리 네트웍, 멀티미디어 프로세서 구조 등



김 석 일

e-mail : ksi@cbucc.chungbuk.ac.kr

1975년 서울대학교 학사학위 취득

1975년~1995년 국방과학연구소 선임 연구원으로 근무

1985년~1989년 미국 North Carolina State University에서 공학박사 취득

1990년~현재 충북대학교 컴퓨터과학과 교수로 재직중

관심분야 : 병렬처리 컴퓨터구조, 슈퍼컴퓨팅, 이기종 분산처리, 시각장애사용자 인터페이스등