버스기반의 VLIW형 프로세서를 위한 최적화 컴파일러 구현

(Implementation of Optimizing Compiler for Bus-based VLIW Processors)

홍 승 표 † 문 수 묵 ††

(Seung-Pyo Hong) (Soo-Mook Moon)

요 약 최근의 고성능 프로세서들은 명령어 수준의 병렬처리(Instruction Level Parallel Processing)를 이용하여 성능향상을 꾀하고 있다. 특히 컴파일러의 도움을 받는 VLIW(Very Long Instruction Word) 방식의 프로세서는 고성능 DSP 및 그래픽 프로세싱 등 특수한 분야에서 사용이 증가하고 있다. 이러한 특수 목적의 프로세서 구조로서 버스 기반의 VLIW 구조가 제안되었으며[2], 이는 포워딩 하드웨어의 부담과 명령어 폭을 줄여주는 장점을 갖는다.

본 논문에서는 제안된 버스 기반의 VLIW 프로세서를 위해 개발된 최적화 스케쥴링 컴파일러를 소개한다. 우선 버스간 연결 및 자원사용을 모델링 하는 기법을 설명하고 이를 바탕으로 레지스터-버스 승진, 복사자 융합, 오퍼랜드 대체 등의 기계 의존적인 최적화 기법과 선택 스케쥴링, EPS(Enhanced Pipelining Scheduling) 기법 등 VLIW 스케쥴링 기법을 어떻게 구현했는지 설명한다. 이러한 최적화 기법들을 멀티미디어 응용 프로그램에 대하여 적용하여 보았고 약 20%의 성능향상을 보임을 확인하였다.

Abstract Modern microprocessors exploit instruction-level parallel processing to increase the performance. Especially VLIW processors supported by the parallelizing compiler are used more and more in specific applications such as high-end DSP and graphic processing. Bus-based VLIW architecture was proposed for these specific applications and it was designed to reduce the overhead of forwarding unit and the instruction width.

In this paper, a optimizing scheduling compiler developed for the proposed bus-based VLIW processor is introduced. First, the method to model interconnections between buses and resource usage patterns is described. Then, on the basis of the modeling, machine-dependent optimization techniques such as bus-to-register promotion, copy coalescing and operand substitution were implemented. Optimization techniques for general-purpose VLIW microprocessors such as selective scheduling and enhanced pipelining scheduling(EPS) were also implemented. The experiment result shows about 20% performance gain for multimedia application benchmarks.

1. 서 론

90년대 중반이후 고성능의 마이크로프로세서는 명령어 수준의 병렬성(ILP)을 이용하여 매 클럭 싸이클 마다 상호의존성이 없는 여러 개의 명령어들을 동시에 수행하여 성능을 획기적으로 향상시키고 있다. ILP 프로세서는 프로그램에서 병렬성을 추출하는 주체에 따라 수퍼스칼라 방식과 VLIW(Very Long Instruction Word) 방식으로 나뉜다. 수퍼스칼라 방식은 프로그램의 수행시간에 하드웨어에 의해 병렬성을 추출하므로 기존의 프로그램을 교체하지 않아도 된다는 장점이 있어서 범용 프로세서에서 널리 쓰인다. 반면, VLIW 방식은 병렬성 추출의 주체가 실행 파일을 생성하는 컴파일러

· 본 연구는 교육부의 학술연구비를 지원받아 서울대학교 반도체공동연구소에서 연구되었습니다. (프로젝트번호 ISRC 97-E-2102)

† 비 회 원 : 서울대학교 전기공학부
castor@altair.snu.ac.kr
†† 종신회원 : 서울대학교 전기공학부 교수
smoon@altair.snu.ac.kr

논문접수 : 1999년 11월 11일
심사완료 : 2000년 3월 9일

이므로 수퍼스칼라 방식에서는 수행하기 힘든 복잡한 최적화 알고리즘을 적용할 수 있고 병렬성 추출에 필요한 부가적인 하드웨어도 필요없다는 점이 장점이다. 하지만 프로그램의 호환이 안되고 프로세서마다 별도의 실행파일을 만들어야 한다는 점 때문에 널리 범용으로는 쓰이지 못하고 DSP 나 그래픽 처리등 특수 목적 프로세서로 쓰이고 있다. VLIW 방식은 하드웨어로 병렬성을 추출할 필요가 없으므로 동시에 수행할 수 있는 명령어의 수를 늘리기가 용이하다.[1]

VLIW 방식의 프로세서는 수퍼스칼라 방식보다 하드웨어의 부담이 적지만 함수유닛 수가 많아짐에 따라 함수유닛 간의 포워딩 및 버스 간의 연결 오버헤드가 크다. 따라서 이러한 하드웨어의 부담을 덜고 같은 명령어 폭에서 동시에 실행가능한 명령어 수를 늘리기 위해 버스기반의 VLIW구조가 제안되었다[2]. 일반 RISC를 기반으로 한 VLIW 프로세서는 포워딩 하드웨어를 포함하며 명령어의 오퍼랜드로 레지스터를 쓰는데 비해 버스구조의 VLIW 프로세서는 포워딩 하드웨어가 없어 더 빠른 클럭 싸이클을 쓸 수 있으며 명령어의 오퍼랜드로 레지스터(통상 32-64개) 보다 적은 수의 버스(통상 8개)를 쓰므로 한 명령어의 폭이 줄어들어 결국 동시에 실행시킬 수 있는 명령어의 수가 늘어난다. 본 논문에서는 이러한 버스기반의 VLIW 방식의 프로세서를 위한 최적화 컴파일러의 구현을 목적으로 한다. 버스와 함수유닛 간의 연결에 대한 기계 기술을 바탕으로 함수유닛을 통한 복사자를 통해 포워딩과 같은 효과를 내도록 하였고, 가능할 경우 복사자 융합과 유사한 명령어의 오퍼랜드 대체를 통해 레지스터 파일의 병목현상을 줄였다. 또한 복사자가 제한적인 상황에서 전역 스케쥴링 기법의 일종인 선택 스케쥴링과 소프트웨어 파이프라이닝의 일종인 EPS(Enhanced Pipelining Scheduling)[3]를 구현하여 수행시간의 단축을 꾀하였다.

다음의 2장은 VLIW 프로세서의 구성과 그의 모델링을 설명하고 일반 RISC 프로세서와 비교하였다. 3장에서는 프로세서 모델링을 바탕으로 구현된 레지스터-버스 승진과 오퍼랜드 대체, 선택 스케쥴링과 EPS 등의 최적화 기법에 대해 설명한다. 4장에서는 실험환경 및 실험결과를 설명하고 5장에서 본 논문의 요약으로 끝을 맺는다.

2. 버스기반 프로세서와그의 모델링

2.1 버스기반의 VLIW 프로세서의 구조

구현된 최적화 컴파일러는 가상의 버스 구조를 기반으로 한 VLIW 프로세서를 대상으로 한다.[2] 여기에서 버스는 CPU 내부에서 각 함수유닛과 레지스터 파일 등의 입력과 출력이 되는 버스로, 어셈블리어 수준에서 프로그래머가 제어할 수 있다는 점이 보통의 RISC 프로세서와 다른 점이다.

버스구조의 프로세서는 버스의 수가 레지스터의 수보다 적어, 명령어의 오퍼랜드를 지정하는 비트수가 적으므로 같은 명령어 폭에서 더 많은 명령어를 동시에 수행시킬 수 있다. 같은 64비트 명령어의 경우 RISC 명령어는 2-4개가 포함되지만 버스구조의 경우 8개의 명령어가 포함된다. 또한 버스나 함수유닛 간의 포워딩 하드웨어가 없으므로 클럭 속도를 높일 수 있다.[2]

본 연구는 64비트에 8개의 명령어가 동시에 실행 가능한 프로세서를 기반으로 하였다. 프로세서의 함수유닛은 ALU, 곱셈기, 메모리 접근 유닛, VRAM 접근 유닛 등 7개가 있고 각 함수유닛의 출력은 서로 다른 버스에 연결되고 이 버스들이 다시 다른 함수유닛이나 레지스터 파일, 메모리 유닛의 입력으로 연결된다. 이들 함수유닛과 버스간의 데이타 경로를 RISC와 비교하면 그림 1과 같고, 전체적인 버스기반 VLIW 프로세서의 구조는 그림 3과 같다.

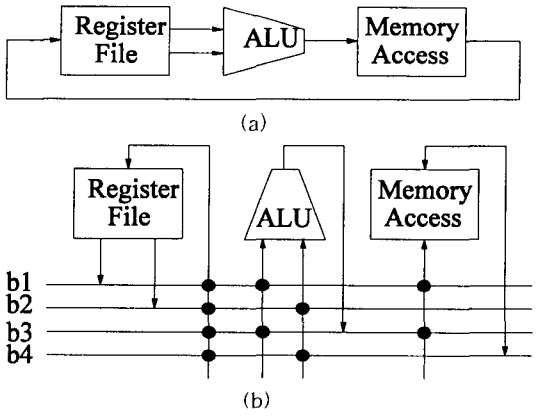

그림 1 프로세서의 데이타패스 (a) RISC, (b) 버스구조 VLIW 프로세서

버스구조의 프로세서가 일반 RISC형 프로세서와 가장 큰 차이점은 어셈블리 레벨에서 버스가 명시적으로 쓰이고 있어 파이프라이닝했을 경우 포워딩을 프로그래머가 제어할 수 있다는 점이다. RISC에서 그림 2(a)와 같은 명령어는 버스기반 프로세서에서 그림 2(b)에 해당한다. b1,b2,b3는 각각 레지스터 파일의 read port1, read port2, ALU의 출력단에 연결된 버스이다.

RISC의 경우 파이프라인 단계 사이의 포워딩은 하드

웨어로 처리하므로 mux와 비교기, 포워딩을 결정하는 로직 및 이들의 연결이 필요하게 된다. 이는 함수유닛의 수에 비례하는 크기를 차지하며 프로세서 자체의 클럭이 느려지는 원인이 된다. 따라서 버스구조 VLIW 프로세서에서는 포워딩이 없는 하드웨어를 가정하였는데, 이 프로세서에서는 버스가 명시적으로 사용되므로 소프트웨어적으로 포워딩을 제어할 수 있다.

단, 버스간의 복사는 자유롭게 이루어질 수 없는데, 이는 버스간 연결의 오버헤드와 함수유닛의 출력과의 충돌 때문이다. 따라서 버스간의 복사는 함수유닛을 통해서만 이루어지므로 제한되는 경우가 있어, 뒤에 설명될 스케쥴링의 제약요인이 된다.

```
add r3,r1,r2;
```

(a) RISC 명령어

```
b1=read r1;
b2=read r2;
b3=add b1,b2;
r3=write b3;
```

(b) 버스기반 프로세서의 명령어

그림 2 RISC와 버스기반 프로세서의 명령어 비교

2.2 하드웨어 모델링

2.2.1 버스간의 복사자 생성 제한

버스 사이에 데이타가 이동할 때는 직접 연결되지 않고 함수유닛을 통하도록 한다. 버스간 데이타의 이동이 레지스터 파일을 통할 경우 저장하기와 읽기의 2 사이클이 걸리는 반면 함수유닛을 통한 복사자는 1 사이클이 걸리므로 함수유닛을 쓸 수 있는 경우라면 복사자를 쓰는 것이 유리하다. 이러한 하드웨어적 연결은 다음과 같은 복사자 생성 행렬로 모델링 할 수 있다.

$$C(i,j)=\begin{cases}1, & if\ b_j = copy(b_i)\ is\ possible,\\ 0, & otherwise\end{cases}$$

예를 들어 ALU를 복사유닛으로 쓴다면 다음과 같은 복사자 생성이 가능하다.

b3=copy(X); X=b1,b3,b5,b7

즉, b3를 대상으로 하는 복사자 중, b1,b3,b5,b7을 소스 오퍼랜드로 갖는 복사자만이 만들어 질 수 있다. 이와 같이 복사자의 생성을 제한하는 이유는 전술한 바와 같이 버스의 연결에 따른 하드웨어의 부담을 줄이고 명령어 폭을 줄이기 위함이다.[1)]

1) 오퍼랜드가 4개중 하나인 경우 2비트, 8개중 하나인 경우 3비트로, 오퍼랜드가 제한될수록 명령어 폭은 줄어든다.

2.2.2 오퍼랜드 대체자의 제한

복사자는 복사자 전파와 무효 코드 제거를 통하여 없어질 수 있다. RISC 프로세서에서 복사자의 전파는 용이한 일이나, 버스기반 프로세서에서는 복사자가 전파되어 명령어의 소스 오퍼랜드를 대체하는 것도 제한이 된다. 앞에서 설명한 버스간의 복사자 생성이 제한되더라도 오퍼랜드 대체가 일어날 수 있으면 복사자가 없어질 가능성이 있으므로 실제로는 실행이 불가능한 복사자도 생성할 수 있다. 이렇게 실행 불가능한 복사자를 생성하기 위해서는 소스 오퍼랜드의 대체 제한에 대한 정보가 필요한데, 이는 첫 번째 소스 오퍼랜드에 대해서 다음과 같은 오퍼랜드 대체 행렬로 기술할 수 있다.

$$S_1(i,j)=\begin{cases}0, & if\ b_j = FU_j(b_i, b_x)\ is\ possible\\ 1, & otherwise\end{cases}$$

$$S_2(i,j)=\begin{cases}0, & if\ b_j = FU_j(b_x, b_i)\ is\ possible\\ 1, & otherwise\end{cases}$$

예를 들어 ALU의 오퍼랜드의 대체를 생각해 보면 다음과 같은 형식이 가능하다.

b3=add(X,Y); X = b1,b3,b5,b7
Y = b2,b4,b6,b8

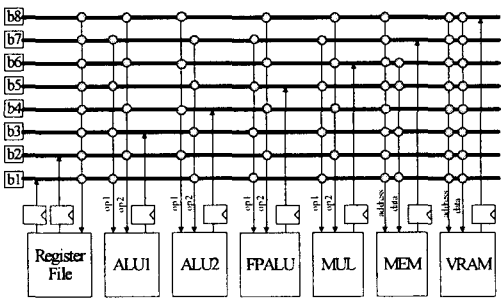

그림 3 전체 하드웨어의 구조

즉, b3를 대상으로하는 ALU 명령어중 첫번째 소스 오퍼랜드로는 b1,b3,b5,b7 만이 쓰이고 두 번째 소스 오퍼랜드로는 b2,b4,b6,b8 만이 쓰인다. 이와 같이 명령어의 소스 오퍼랜드를 제한하는 이유는 복사자 생성을 제한하는 이유와 같다. 전체적인 버스기반 VLIW프로세서의 구조는 그림 3과 같다.

3. 최적화 기법

버스구조 프로세서는 하드웨어의 오버헤드와 명령어의 폭을 줄이기 위해 적은 수의 버스만이 존재한다. 따

라서 즉시 쓰이지 않는 값은 바로 레지스터에 저장하도록 하는데, 경우에 따라 다음에 쓰이지 않을 값도 레지스터를 경유하게 되어 결과적으로 레지스터 파일 접근이 프로그램 수행의 임계경로가 되는 경우가 있다.

3.1 Motivating Example

다음과 같은 두 명령어 사이에 의존성이 있는 RISC 코드를 고려해보자.

r3 = add r2, r4;
r5 = sub r3, r4;

하나의 RISC 명령어에 해당하는 VLIW 코드는 2개의 레지스터 읽기 명령어, 1개의 레지스터 쓰기 명령어 그리고 1개의 ALU 명령어로 이루어진다. 원시 VLIW 코드와 리스트 스케쥴링 후의 VLIW 코드는 그림 4의 (a), (b)와 같다.

b1=read r2; (1)	b1=read r2 b2=read r3; (1),(2)
b2=read r3; (2)	b3=add b1,b2 b2=read r4; (3),(6)
b3=add b1,b2; (3)	r3=write b3; (4)
r3=write b3; (4)	b1=read r3; (5)
b1=read r3; (5)	b3=sub b1,b2; (7)
b2=read r4; (6)	r5=write b3; (8)
b3=sub b1,b2; (7)	
r5=write b3; (8)	
(a) 원시 코드	(b) 리스트 스케쥴링 후의 코드

b1=read r2 b2=read r3; (1),(2)
b3=add b1,b2 b2=read r4; (3),(6)
b3=sub b3,b2; (7*)
r5=write b3; (8)

(c) 승진 및 복사자 제거 후의 코드

그림 4 레지스터-버스 승진 및 복사자 제거의 예

VLIW 명령어 (5)는 r3에서의 의존성때문에 (6)와 같이 스케쥴되지 못했다. 그러나 이 부분을 자세히 살펴보면 명령어 (4)와 (5)에서 r3는 단지 RISC 스타일의 프로그램과 의미를 맞추기 위해 중간에 값을 저장하는 임시적인 장소로만 쓰였음을 알 수 있다. 따라서 유효성 분석(Live Analysis)[6]을 통해 r3가 유효하지 않다고 판단된 경우는 r3를 저장할 필요가 없다. 이때 (4) 및 (5)는 생략되고 명령어 (7)의 첫 번째 오퍼랜드 b1이 b3로 대체되어 b3=sub b3,b2로 바뀌어 2 사이클이 단축될 가능성이 있다. r3가 유효한 경우에는 (4)는 생략할 수 없지만 (5)는 여전히 생략이 가능하다. 위와 마찬가지로 (7)의 첫 번째 오퍼랜드 b1은 b3로 대체되어 그림 4(c)의 (7*)가 되며 이 경우 1사이클이 단축된다. 이 문제는 기존의 메모리-레지스터 승진 (Memory-to-Register Promotion)[11]과 비슷한 레지스터-버스 승진의 문제이며 이로 인해 생기는 버스간의 복사자가 버스 구조에서 제한된다. 따라서 우선 1) 레지스터-버스 승진이 가능한지 검토한 후, 2) 복사자가 생성 가능하거나 아니면 복사자가 융합되어 다른 버스로 대체될 수 있는지 검토하여 두 조건이 성립할 경우 앞의 예와 같은 최적화를 행하게 된다.

3.2 레지스터-버스 승진

레지스터-버스 승진은 고전적인 최적화 기법인 메모리-레지스터 승진[11]과 비슷한 알고리듬이다. RISC 프로세서에서 메모리에의 접근은 레지스터 파일의 접근보다 긴 시간이 걸리고 경우에 따라서는 캐쉬 미스를 내므로 줄이는 것이 유리하다. 마찬가지로 버스구조의 프로세서에서는 레지스터가 바로 함수유닛의 입력이나 출력으로 쓰일 수 없으므로 레지스터를 버스로 대체하여 수행시간의 단축을 꾀한다.[7]

메모리-레지스터 승진은 다음과 같이 이루어진다. 메모리의 유효범위(Live Range)는 store(definition)들과 load(use)들의 집합으로,

- 이들이 같은 메모리 위치에 접근하며
- 각 load에 닿는 모든 store가 이 집합에 포함되어야 한다.

메모리의 유효범위에서 load와 store가 모두 같은 메모리 위치에 접근하는 지는 어셈블리를 분석하거나 컴파일러 전단부에서 오는 정보를 활용한다. 이러한 메모리의 유효범위는 store는 copy로 대체되고 load는 지워짐으로써 메모리-버스 승진이 이루어지며, 위와 비슷한 과정으로 레지스터-버스 승진이 이루어진다. 레지스터의 유효범위는 write(definition)들과 read(use)들의 집합으로, 각 read 명령어에 닿는 모든 write 명령어가 집합에 포함되어야 한다. 레지스터는 메모리와 달리 같은 위치에 접근하는지에 대한 추가적인 정보는 필요하지 않다. 이러한 레지스터의 유효범위에서 레지스터에 대한 write 명령어는 버스간의 복사자로 바뀌고 레지스터 read 명령어는 지워진다. 그림 4 (b)와 같은 VLIW코드에서 명령어 (4)는 버스간의 복사자로 바뀌고 명령어 (5)는 지워져 그림 5과 같이 변환된다.

b1=read r2 b2=read r3; (1),(2)
b3=add b1,b2 b2=read r4; (3),(6)
b1=copy b3; (4*)
b3=sub b1,b2; (7)
r5=write b3; (8)

그림 5 레지스터-버스 승진 후의 코드

레지스터-버스 승진(Promotion)의 알고리듬은 다음과 같다.

```
Algorithm Promotion(a register live range)
    mark write as invalid;
    mark read as copy;
    if( the marked copy is permitted ) {          /* refer copy matrix */
        delete write marked invalid;
        change read to copy;
        return SUCCESS;
    } else {                                  /* the marked copy is restricted */
        if( operand substitution is permitted ) { /* refer substitution matrix */
            delete write marked invalid;
            substitute operand;
            return SUCCESS;
        } else {                              /* operand substitution is restricted */
            return FAILURE;
        }
    }
```

3.3 제한된 복사자와 그의 융합

앞절에서 설명한 바와 같이 레지스터-버스 승진이 이루어지지만 실제 버스구조에서는 버스간의 복사자가 제한되므로 복사자 조건과 융합에 대한 검토가 필요하게 된다. 버스간의 연결은 되어 있지 않으므로 버스간의 복사자는

- 함수유닛을 통하는 형태로 나타나거나
- 융합되어 없어져야 한다.

첫째 조건을 보면 그림 5의 명령어 (4*)에서 b1은 레지스터 read 단자이므로 b3를 함수유닛을 통해 복사할 수 없다(복사자 생성 행렬 C(1,3)=0). 그러나 둘째 조건을 보면, 첫번째 오퍼랜드 b1은 b3로 대체할 수 있다.(오퍼랜드 대체행렬, S(1,3)=10) 따라서 b1이 유효하지 않을 경우 그림 5에서의 복사자(4*)는 생략되고 명령어(7)에서 b1은 b3로 대체된다. 레지스터-버스 승진과 복사자 융합을 통해 최종적으로 최적화된 코드는 그림 4 (c)와 같다. 스케쥴되지 않은 VLIW 코드보다는 4 사이클이, 리스트 스케쥴링이 행해진 VLIW 코드보다는 2 사이클이 줄어들었음을 알 수 있다.

3.4 선택 스케쥴링 기법

선택 스케쥴링 기법[3]은 비환형 그래프(DAG)에서 동시에 수행될 수 있는 명령어의 집합을 찾는 스케쥴링 방법이다.

이 방법은 비환형 그래프의 근원노드[2]에 빈 노드를 생성하고 이후의 모든 경로에서

1) 데이타 의존성이 없는 명령어들을 모아,
2) 그 중에서 자원제한에 맞는 명령어들을 선택하여
3) 그 명령어를 근원노드로 코드 이동하여 하나의 VLIW 명령어 그룹을 만들어낸다.

이러한 과정을 근원노드의 종속자들에 대하여 반복적으로 적용하여 그래프에 있는 모든 명령어들이 스케쥴될 때까지 반복하여 스케쥴된 코드를 생성한다.

버스기반 프로세서에서는 복사자가 제한되므로 위의 알고리듬에 약간의 제한이 가해진다. 보통의 RISC 프로세서에서는 유용한 명령어의 집합에서 추측 명령어를 선택하여 코드 이동할 때, 다른 경로에서 목적 레지스터(target register)이 유효(live)하더라도 레지스터 재명명(renaming)을 통하여 복사자를 남기고 분기문을 거슬러 올라갈 수 있으나, 버스구조의 프로세서에서는 각 함수유닛의 출력이 할당된 버스에 고정되어 있고 버스를 재명명하지 못하므로 코드 이동에 제한을 받는다. 따라서 코드 이동중, 추측 명령어 이동을 할 때, 다른 경로에서 원래의 대상 버스가 유효할 때는 복사자 생성 행렬을 참조하여 복사자가 생성가능할 경우에만 명령어를 이동시킨다.(그림 6)

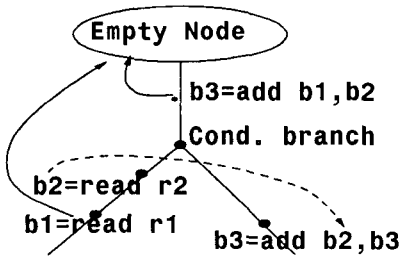

그림 6 추측명령어의 이동

3.5 소프트웨어 파이프라이닝 : EPS

소프트웨어 파이프라이닝은 루프반복(iteration)의 경계를 넘어 루프를 압축하여 수행시킴으로서 루프의 수행시간을 단축시키는 방법이다. EPS 기법은 루프의 적절한 간선을 끊어 환형 그래프를 비환형 그래프(DAG)로 변환하고 이 그래프에서 적절한 방법으로 서로 데이타 의존성이 없는 명령어들을 하나로 묶는 일을 반복한다. 그래프에서 동시에 실행가능한 명령어들을 찾는데는 전술한 선택 스케쥴링 기법이 사용되었다. 이러한 작업의 결과로 나중에 스케쥴된 명령어 그룹에는 루프의 후간선(backedge)을 넘어 다음에 반복되어 수행될 명령어들도 포함되게 된다.[4]

이러한 EPS 기법은 소프트웨어 파이프라이닝의 문제를 DAG 스케쥴링 문제로 치환하여 DAG 스케쥴링을 반복함으로써 커널코드를 따로 계산하지 않고 파이프라

2) Topological sort에서 첫 노드, root

인의 시작코드, 커널, 종료코드를 가진 파이프라인된 루프가 자연히 생성된다는 특징이 있다.[10]

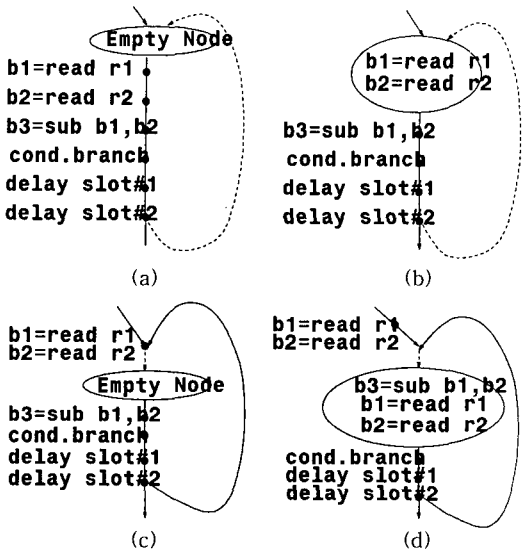

그림 7 소프트웨어 파이프라이닝 과정

복사자 생성 제한 및 오퍼랜드 대체에 대한 제한은 선택 스케쥴링 기법에 반영이 되어 있고 EPS 기법은 명령어 스케쥴링 과정에서 선택 스케쥴링 기법을 이용하므로 복사자 생성 제한에 대한 별도의 고려가 필요하지 않다. 따라서 EPS에 관한 더 이상의 논의는 생략하도록 한다. 그림 7은 순차코드가 VLIW 코드로 소프트웨어 파이프라이닝 되어 그룹핑 되는 예를 보인 그림이다.

4. 실험 환경 및 실험 결과

4.1 실험 환경

소프트웨어 개발환경에서 컴파일러는 GCC의 후단부를 버스구조의 VLIW 프로세서에 맞게 고쳐 C 파일(sample.c)에서 어셈블리 파일(sample.s)을 생성한다. 이렇게 생성된 어셈블리 파일은 그 자체로 목적 파일(sample.o)과 시뮬레이션이 가능한 실행 파일(sample.x)을 만들 수 있지만 프로세서의 병렬성을 충분히 활용하지 못하는 경향이 있으므로 목적 파일을 만들기 전에 최적화기를 통하여 최적화된 어셈블리 파일(samplex.s)을 만든다. 이렇게 만들어진 어셈블리 파일로 최적화된 목적 파일(samplex.o)과 시뮬레이션이 가능한 최적화된 실행 파일(samplex.x)을 생성한다. 이를 시뮬레이터 위에서 수행시키면 수행 결과와 싸이클 수를 얻을 수 있다.(그림 8)

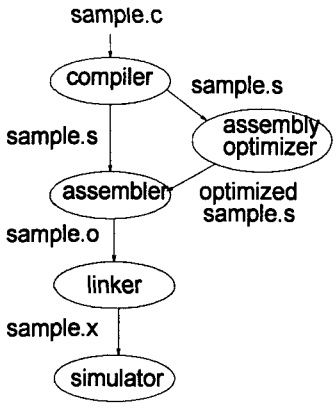

그림 8 최적화 과정

4.2 실험 결과

본 실험에서는 최적화기를 통한 어셈블리로 만들어진 실행 파일과 최적화기를 통하지 않는 실행 파일의 사이클 수를 비교하여 성능 향상을 측정하였다. 벤치마크는 MPEG decoder, Text display, Nested loop 등이 있으며 이들에 대해 무효코드 제거와 같은 기본적인 최적화 기법과 본 논문에서 설명된 버스-레지스터 승진과 선택 스케쥴링과 EPS와 같은 일반적인 VLIW 프로세서를 위한 최적화 기법을 구현하여 시뮬레이션 결과를 표 1에 나타내었다.

대부분의 경우 최적화 기법이 성능향상에 기여하지만, Text Display 벤치마크는 EPS 를 적용시켰을 경우 오히려 성능이 나빠짐을 볼 수 있다. 이는 최내각 루프의 수행빈도가 높지 않은 경우, 부기코드의 부담때문에 성능저하가 일어날 수 있음을 보여주는 예라 할 수 있다.

표 1 시뮬레이션 결과(싸이클 수)

벤치마크	최적화 이전	레지스터-버스 승진 only	EPS only	최적화 이후 (Speed Up)
MPEG2 Decoder	3,557,175	3,556,823	3,556,327	2,817,746 (21%)
Nested Loop	55,698	55,476	55,534	54,724 (1.7%)
Text Display	380,665	380,665	380,923	380,923 (-0.07%)

5. 맺 음

본 논문에서는 버스기반의 VLIW 프로세서를 상정하

고, 이 때문에 복사자 생성과 오퍼랜드 대체가 제한되는 상황에서 최적화 기법들을 구현하였다. 버스와 함수유닛 간의 연결에 대한 기계기술을 바탕으로 함수유닛을 통한 복사자를 통해 포워딩과 같은 효과를 내도록 하였고, 가능할 경우 복사자 융합(Copy Coalescing)과 비슷한 명령어의 오퍼랜드 대체를 통해 레지스터 파일의 병목현상을 줄였다. 또한 복사자가 제한적인 상황에서 전역 스케쥴링 기법의 일종인 선택 스케쥴링과 소프트웨어 파이프라이닝의 일종인 EPS을 구현하여 MPEG2 decoder 에서 최고 20%의 성능향상이 이루어짐을 확인하였다. 이러한 컴파일 기법은 하드웨어의 구성과 밀접한 관계가 있으므로 하드웨어에서의 버스간의 연결 및 버스와 함수유닛간의 연결을 모델링하여 최적화 기법의 효과가 극대화되도록 하였다.

참 고 문 헌

[1] A. Abnous, J. Gray, A. Naylor and N. Bagherzadeh. "VIPER: A VLIW Integer Microprocessor," *Journal of Solid-State Circuits*, 28(12):1377-1382, December.1993.

[2] Dowan Kim, *Microarchitecture Design of a VLIW Microprocessor*. MS thesis, Seoul National University, 1997.

[3] S.-M Moon and K. Ebcioglu. "Parallelizing Non-numerical Code with Selective Scheduling and Software Pipelining," *ACM Transactions of Programming Languages and Systems*, 19(6), November, 1997

[4] K. Ebcioglu and T. Nakatani. "A New Compilation Techniques for Parallelizing Loops with Unpredictable Branches on a VLIW architecture," In *Languages and Compilers for Parallel Computing*, PP. 213-219. MIT Press, 1989.

[5] V.H. Allen et al. Software pipelining. ACM Computing Survey, 27(3), September 1995.

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

[7] Fred C. Chow and John L. Hennessy. "The Priority-Based Coloring Approach to Register Allocation," *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 4, pages 501-536, Oct.1990.

[9] B. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," Proceedings of the 27th Annual International Symposium on Microarchitecture(Micro-27), Nov. 1994

[10] K. Ebcioglu, R. Grove, K.-C. Kim, G. Silberman, and I. Ziv, "VLIW Compilation Techniques in a Superscalar Environment," In Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implmentation, pages 36-48, June 1994.

[11] A.V.S. Sastry and Roy D.C. Ju, "A New Algorithm for Scalar Register Promotion based on SSA Form" In Proceedings of the SIGPLAN 1998 Conference on Programming Language Design and Implmentation, June 1998.

홍 승 표

1997년 서울대학교 전기공학부 졸업(학사). 1999년 서울대학교 전기공학부 졸업(석사). 1999년 ~ 현재 서울대학교 전기공학부 박사과정. 관심분야는 내장형 컴파일러, 재구성이 가능한 하드웨어

문 수 묵

1987년 서울대학교 컴퓨터공학과 졸업(학사). 1993년 University of Maryland, Computer Science 졸업(Ph.D). 1993년~ 1994년 IBM T. J. Watson 연구소. 1994년 ~ 현재 서울대학교 전기공학부 조교수. 관심분야는 자바 JIT 컴파일러, 내장형 컴파일러, 동적 컴파일