VLIW 프로세서 구조에서의 명령어 중복스케쥴링

042 - 580 - 6212

정회원 김 석 주*, 김 석 일**

042 - 488 - 5105

Duplicated Instruction Scheduling Method on VLIW Processor Architecture

Suk-ju Kim*, Sukil Kim** *Regular Members*

요 약

본 논문에서는 VLIW구조에서 명령어 그래프를 스케쥴링할 때 비어있는 NOP 슬롯에 다른 명령어를 복사하여 스케쥴링하는 명령어 중복 할당 기법을 보였다. 명령어 중복 할당 기법은 VLIW 구조에서 동일한 명령어를 복사하여 할당하는 방법으로 이 방법을 적용하면 복사된 명령어들이 데이터 의존 관계를 해소하기 위해 채용된 바이패싱 회로와 같은 역할을 하는 특징이 있다. 또한 명령어 중복 할당 기법을 사용한 스케쥴링은 오버헤드가 적으므로 쉽게 적용할 수 있다. 실험을 한 결과, 간단한 바이패싱 회로를 갖는 VLIW에서 명령어 중복 할당 기법을 적용할 경우 8% ~ 25%의 성능 향상 효과가 있음을 알 수 있었다. 따라서 간단한 바이패싱 회로를 갖는 VLIW의 경우에는 본 논문에서 제안한 명령어 중복 스케쥴링을 적용하는 것이 적합할 것으로 판단된다.

ABSTRACT

In this paper, we present a duplicated instruction allocation method in ways that inserts copies of instructions to vacant NOP slots with respect to scheduling which utilizes an instruction graph in VLIW architecture. The Duplicated Instruction Scheduling engages in the multiple copying and allocating of same instruction in VLIW architecture. It is characterized by the fact that duplicated instructions generated from the scheduling method become to take on the same role as the bypassing scheme which is adopted to remove data dependencies. Besides small overhead of Duplicated Instruction Scheduling makes it applicable appropriately. From the experimental results, we get to know that Duplicated Instruction Scheduling can enhance the performance up to 8% ~ 25% in case for VLIW machine equipped with simple bypassing paths. In consequence, for a VLIW machine with simple bypassing paths, to apply suggested Duplicated Instruction Scheduling is expected to be advisable.

I. 서 론

명령어 수준의 병렬성을 이용하기 위한 VLIW (Very Long Instruction Word)구조는 여러 개의 연산처리기가 전역 공유 레지스터 파일에 연결되어 동기적으로 동작하며 제어 흐름을 컴파일러가 결정함으로써 수퍼스칼라 구조에 비해 상대적으로 하드웨어가 간단하여 매우 광범위하게 연구되고 있다[1][2]. 또한 컴파일러 최적화 기법과 VLSI 기술의 발달로 VLIW 구조의 실제적 구현 사례가 늘고 있다. 특히 근래에 들어와 멀티미디어를 빠르게 처리하는 프로세서 구조의 경우에는 미디어 정보가 가지고 있는 서브워드 병렬성(subword parallelism)을 활용하는 구조로 각광을 받고 있다[3].

VLIW 구조에서는 중앙의 제어기가 하나의 긴 명령어를 매 사이클마다 발생시킨다. 각각의 긴 명령어는 여러 개의 독립적인 명령어로 구성되어 있

* 혜천대학 컴퓨터·통신계열 멀티미디어 전공(kimse@hcc.ac.kr), ** 충북대학교 전기전자및컴퓨터공학부(ksi@cbucc.chungbuk.ac.kr)
논문번호 : 010338-1116, 접수일자 : 2001년 11월 16일
※ 본 연구는 한국과학재단 목적기초연구(R05-2002-000-01470-0)지원으로 수행되었음.

으며 병렬로 수행될 수 있고, 또 수백 비트로 구성되어 각 연산처리기를 매 사이클마다 직접 독립적으로 제어할 수 있는 제어 비트를 가지고 있다. 각각의 연산이 완료되는 사이클 시간은 정적으로 알려지며 각 연산이 파이프라인 형태로 수행된다.

VLIW 구조에는 RISC 프로세서의 특징인 파이프라이닝[4]이 적용되어 있다. 그러나 파이프라이닝의 단점으로 지적되는 것 중의 하나는 파이프라인이 정지됨으로써 파이프라인 최고 속도로 실행이 될 수 없다는 것이다. 그 이유는 파이프라인 해저드가 발생되기 때문인데 RISC 프로세서에서는 파이프라인의 RAW(Read After Write) 해저드를 해결하기 위한 방법으로 명령어 파이프라인의 실행단계가 종료되자마자 연산 결과를 다음 번 파이프라인 단계의 실행 유닛에서 사용할 수 있도록 하는 바이패싱(bypassing)회로를 사용한다. Viper(VLIW integer processor)[5] 와 같은 VLIW 구조에는 서로 다른 하나의 연산처리기에서 계산된 중간 결과를 다른 연산처리기에서도 사용할 수 있도록 바이패싱 회로를 제공하나 이러한 경우 바이패싱 회로의 복잡도가 연산처리기의 수가 늘어남에 따라 크게 증가하는 단점이 있다.

이러한 단점을 개선하기 위하여 본 논문에서는 바이패싱 회로대신에 여러 개의 연산처리기에서 동일한 계산을 수행하도록 하여 가상의 바이패싱 회로가 존재하는 것과 같은 효과를 얻을 수 있는 명령어 중복 스케쥴링 기법을 제안하였다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 파이프라인과 바이패싱 및 ILP (Instruction Level Parallelism) 명령어 스케쥴링을 검토하고 목적 코드 생성을 고찰한다. 제 3 장에서는 명령어 중복 스케쥴링 기법을 제안하여 그 알고리즘을 기술하였다. 제 4 장에서는 제안된 기법을 사용하여 여러 가지 명령어 그래프를 대상으로 모의 실험을 수행하여 실험 결과 및 그에 대한 분석을 행하였다. 제 5 장에서는 간단한 바이패싱 회로를 갖는 머신에서 명령어 중복 스케쥴링 기법을 적용하면 가상의 바이패싱 회로를 추가한 것과 같은 효과가 있음을 보였다.

Ⅱ. 바이패싱 기술

2.1 파이프라인과 바이패싱

파이프라인 구조의 단점으로 지적되는 것은 첫째 하드웨어의 복잡도가 증가된다는 것이다. 둘째는 파이프라인을 최대 속도로 연속하여 실행할 수 없다는 것인데 그 이유를 살펴보면 파이프라인이 원활하게 수행되지 못하게 하는 파이프라인 해저드라고 하는 현상이 생기기 때문이다. 이 해저드 중에 자료 의존에 의한 데이터 해저드가 발생하는데 데이터 해저드는 WAR(Write After Read), WAW(Write After Write) 및 RAW 해저드로 구분할 수 있다. 그중 RAW 해저드는 실행 코드에서 하나의 명령어가 이전에 실행된 명령어의 결과를 피연산자로 필요할 때 이전명령어가 그 결과를 쓰기 완료한 후에 읽도록 설정되어 있으나 파이프라인에서 이 사실을 위반할 때 발생한다.

RISC 프로세서는 바이패싱을 사용하여 RAW 해저드에 의한 지연을 해소한다[5]. 즉 바이패싱 하드웨어의 기능은 이 RAW 해저드를 해결하기 위하여 필요한 피연산자를 임시 파이프라인 레지스터에서 ALU의 입력으로 바로 전달시키는 것이다.

그러나 연산처리기의 개수가 증가하면 모든 연산처리기 사이를 연결하는 바이패싱 회로를 구현하는 것은 매우 큰비용을 필요로 한다. 그 이유는 하나의 연산 처리기의 출력을 다른 연산 처리기의 입력과 비교해야 하므로 필요한 비교기의 수가 연산처리기 개수의 제곱의 비율로 증가한다. 또한 비교기가 차지하는 공간이 급격히 증가할 뿐 아니라 피연산자를 바이패싱하기 위하여 필요한 바이패싱 경로는 다중의 데이터 경로를 연결하는 전역 버스이므로 큰 면적을 차지한다.

따라서 VLIW 프로세서의 실제 구현 사례를 살펴보면 특정한 연산처리기의 그룹들 사이에서만 바이패싱 회로를 구현하는데 초기 VLIW 프로세서의

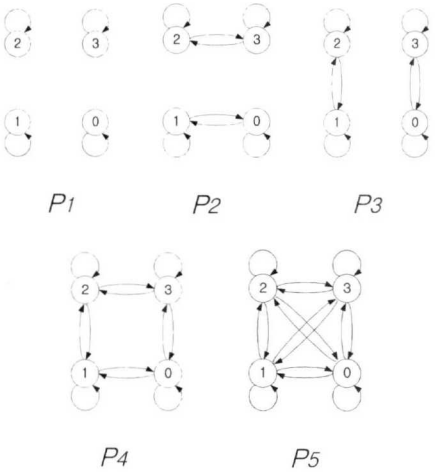


그림 1. 4개의 연산처리기의 바이패싱패싱 네트워크 토폴로지

전형적인 모형인 Viper는 그림 1의 P_4 와 같이 인접한 연산처리기들 사이에서만 바이패싱 회로를 갖도록 되어있다. 또한 피연산자를 바이패싱할 때 피연산자의 주소가 결정되는 ID 단계와 결과를 산출하는 EX 단계 사이를 줄임으로서 바이패싱 회로의 복잡도를 줄일 수 있다. Load/Store 연산을 위한 변위 주소법(displacement addressing mode)을 사용하는 대신에, 레지스터 간접 주소법(register indirect addressing mode)을 사용하여 MEM 단계를 제거한다. 하드웨어의 구현보다 간단하다는 사실 때문에 Viper 에서는 MEM 단계를 없앤 4 단계(IF-ID-EX-WB)의 파이프라인이 적용되었다.

2.2 ILP 명령어 스케쥴링

VLIW 구조용 컴파일러는 동시에 실행할 수 있는 명령어들을 나열하여 긴 명령어를 생성하기 위해 컴파일러에 의한 정적 스케쥴링 기법을 이용하여 병렬성을 추출한다. 정적 스케쥴링은 큰 명령어 창을 다룰 수 있으므로 전역 스케쥴링 (global scheduling) 방식[6][7][8][9] 이 주류를 이룬다.

전역 스케쥴링 기법은 기본 블록(basic block)의 경계를 넘어서 루프가 제거된 코드를 대상으로 병렬화를 수행하는 기법으로서, 주어진 제어 흐름 그래프에서 상호 독립적인 명령어 집단을 생성하여야 하는 공통적인 문제를 가지고 있다. 대부분의 전역 스케쥴링 기법은 이러한 문제를 해결하는 방법으로 먼저 어떤 집단에 포함시킬 수 있는 모든 명령어들의 가용 집합을 계산한 후, 경험적 방법으로 가용 집합에서 최선의 명령어를 선택하여, 선택된 명령어를 그 집단에 포함시키는 방법을 이용한다. 이러한 전역 스케쥴링 기법에는 트레이스 스케쥴링(trace scheduling)[8][9] 수퍼블록 스케쥴링(superblock scheduling)[7], 삼투 스케쥴링(percolation scheduling)[10] 등이 있다.

트레이스 스케쥴링은 프로그램의 분기 명령들에 의한 흐름 내에 있는 명령어들의 병렬성을 최대로 찾아내는 기법으로서 트레이스(trace)라고 부르는 실행될 확률이 높은 일련의 기본 블록을 인식하여 마치 조건문이 존재하지 않는 것처럼 취급하여 코드 발생기에 전달한다. 트레이스가 선택되면 그 다음 단계로 트레이스 내의 명령어들을 컴팩션(compaction)하는데 컴팩션이란 트레이스를 압축하여 적은 수의 긴 명령어를 구성하는 것이다. 확률적으로 가장 적성이 높은 경로인 트레이스 밖으로 분기하는 경우에 논리적 모순을 발생시키는 코드 이동을 허용한다. 컴파일러는 이런 경우에 원래 프로그램의 의미를 복구할 수 있도록 보상코드(compensation code)를 추가한다. 트레이스 스케쥴링은 Multiflow 사에 의해 설계된 트레이스 프로세서를 위해 구현되었다[6].

트레이스 스케쥴링에서 보상코드가 커지는 단점을 보완하기 위한 수퍼블록 스케쥴링은 각각 한 개의 진입점과 출구를 갖는 트레이스와는 달리 한 개의 진입점과 여러 개의 출구를 갖는 수퍼블록을 사용함으로서 고려 대상인 기본 블록의 수를 증가시키고, 보상 코드의 수를 감소시키기 위한 테일 중복(tail duplication)과 같은 기법을 적용하고 있다 [7]. 이 수퍼블록 스케쥴링 기법은 Illinois 대학의 IMPACT-I 컴파일러 내에서 구현되었다[11].

삼투 스케쥴링(percolation scheduling)[10]은 본래의 프로그램의 의미를 보존하며 기본 블록의 경계를 넘어 프로그램 그래프를 좀더 병렬수행이 가능한 형태로 변환하는 스케쥴링 기법이다. 삼투 스케쥴링은 조건문 처리의 결과에 따라 후속 명령을 처리할 수 있는 3-way 조건 분기문을 지원하는 Viper 구조에서 실행율을 높이기 위해 제안되었다.

2.3 ILP 명령어 스케쥴링

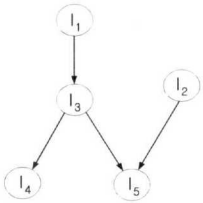
바이패싱 회로를 갖는 VLIW 머신에서 목적 코드를 생성하는 기법을 검토하기 위하여 먼저 복사 스케쥴링(duplication scheduling) 기법을 살펴본다. 어떤 태스크가 하나의 프로세서에 할당되었고 다른 프로세서에 할당된 태스크가 자료의존성으로 그 태스크에 영향을 준다고 할 때, 만일 통신비용이 과도하게 증가하면 자료를 생성하는 태스크를 직접 가져와 그 프로세서에서도 수행하고 그 결과를 이용하여 통신비용을 줄이고 병렬성을 충분히 이용할 수 있도록 하는 기법이 연구되었다[12][13][14].

그림 2 는 그림 2(a)의 소스프로그램을 토대로 두 개의 연산처리기로 구성된 P_1 구조용 목적 코드가 생성되는 과정을 나타낸다. 여기서 P_1 구조란 그림 1의 P_1 처럼 각 연산처리기 자체에만 바이패싱 회로가 있는 구조이다.

그림 2(b)는 그림 2(a)를 토대로 얻은 명령어 그래프이며 노드와 화살표 있는 간선은 각각 명령어와 명령어간의 자료 의존 관계를 나타낸다.

그림 2(c)와 (d)는 그림 2(b)의 명령어 그래프를 대상으로 두 개의 연산처리기로 구성된 P_1 구조에서 각각 가능한 명령어 스케쥴링 결과이다. 즉, I_1 과 I_2간에는 자료 의존 관계가 존재하지 않으므로

```
I1: lw    $14, 20($sp)
I2: lw    $8, 22($sp)
I3: mul   $15, $14, 4
I4: addiu $24, $15, 1
I5: addu  $25, $15, $8
```



(a) Source 프로그램 (b) 명령어 그래프

F_0	F_1
I1: lw $14, 20($sp)	I2: lw $8, 22($sp)
I3: mul $15, $14, 4	nop
I4: addiu $24, $15,1	nop
I5: addu $25, $15,$8	nop

(c) P_1 구조의 목적 코드의 예 1

I1: lw $14, 20($sp)	I2: lw $8, 22($sp)
I3: mul $15,$14,4	nop
I4: addiu $24,$15,1	nop
nop	I5: addu $25,$15,$8

(d) P_1 구조의 목적 코드의 예 2

I1: lw $14, 20($sp)	I2: lw $8, 22($sp)
I3: mul $15,$14,4	nop
I4: addiu $24, $15, 1	I5: addu $25,$15,$8

(e) P_2 구조의 목적 코드

I1: lw $14, 20($sp)	I2: lw $8, 22($sp)
I3: mul $15,$14,4	I3: mul $15, $14, 4
I4: addiu $24, $15,1	I5: addu $25,$15,$8

f) 명령어를 중복 스케쥴링한 P_1구조의 목적 코드

그림 2. 서로 다른 바이패싱 연결 구조와 목적 코드

서로 다른 연산처리기에 스케쥴링되어 하나의 긴 명령어를 구성한다. I_3과 I_4는 각각 I_1과 I_3와의 자료의존관계로 인하여 동일한 연산처리기 F_0에 스케쥴링된다. 만일 I_4를 두 번째 연산처리기 F_1에 스케쥴링하는 경우에는 연산처리기 F_0에 스케쥴링된 명령어 I_3와의 자료 의존 관계로 인하여 명령어 I_3가 포함된 긴 명령어와 명령어 I_4가 포함된 긴 명령어 사이에 NOP으로만 이루어진 긴 명령어가 삽입되어야 한다. 따라서 명령어 I_4는 연산처리기 F_0에 스케쥴링되어야 한다. 명령어 I_4와 I_5간에는 자료의존관계가 없으므로 명령어 I_5는 I_4가 스케쥴링된 연산처리기에 스케쥴링하거나 다른 연산처리기에 스케쥴링이 가능하다. 그림 2(c)는 동일한 연산처리기 F_0에 스케쥴링하는 경우를 보인 것이다. 그러나 명령어 I_5를 다른 연산처리기 F_1에 스케쥴링하는 경우에는 명령어 I_5와 I_3간의 자료의존관계(**$15**)로 인하여 그림 2(d)와 같이 한 사이클 늦은 네 번째 사이클에 실행되도록 목적 코드가 생성되어야 한다.

P_2 구조의 경우에는 명령어 I_5를 실행하는데 필요한 피연산자 $15가 바이패싱 회로를 통하여 연산처리기 F_0로부터 제공될 수 있으므로 그림 2(e)와 같이 명령어 I_5와 I_4를 하나의 긴 명령어로 구성할 수 있다. 따라서 P_2 구조에서는 P_1 구조에서보다 한 사이클 빠른 계산이 가능하다. 만일 그림 2(d)에서 두 번째 긴 명령어의 NOP의 위치에 중복해서 명령어 I_3를 스케쥴링하고 명령어 I_5를 연산처리기 F_1에 스케쥴링하면 명령어 I_3 와의 자료의존관계가 해결되므로 그림 2(f)와 같이 목적 코드를 구성하는 긴 명령어의 수가 하나 적어지게 되므로 P_2 구조에서와 같이 전체적으로 1 사이클 빠른 계산이 가능하다. 즉, NOP이 포함될 코드 슬롯에 다른 연산처리기가 실행할 명령어를 중복 할당하는 경우에 전체적으로 목적 코드의 길이가 짧아져 프로그램의 실행 사이클을 줄일 수 있다.

그림 2(e)와 (f)를 비교하면 P_2 구조에서의 할당 결과는 P_1 구조의 NOP에 의미 있는 명령어를 중복 할당한 결과와 동일한 결과를 보인다. 따라서 명령어를 중복 할당하면 바이패싱 회로를 추가하지 않고도 가상의 바이패싱 회로를 추가한 효과를 얻을 수 있음을 보여준다.

Ⅲ. 명령어 중복 스케쥴링 기법

본 논문에서는 명령어 중복 스케쥴링(Duplicated Instruction Scheduling)기법을 제안하였다. 명령어 중복 스케쥴링 기법은 자료 의존 관계가 있는 한 쌍의 명령어들이 서로 다른 연산처리기에 할당된 경우 이들간에 필요한 연산자를 바이패싱 회로를 통하여 전달받지 않고 피연산자를 생성하는 명령어를 해당 연산처리기에서 직접 실행하도록 함으로써 피연산자를 전달받는 시간을 줄여 총 실행시간을 단축시키는 기법이다.

3.1 스케쥴링 알고리듬

명령어 중복 스케쥴링 기법을 설명하면 명령어들 사이의 자료 의존성을 나타내는 명령어 그래프에서 먼저 명령어의 그룹을 정하고 해당 그룹에 속한 명

령어에 대해 1)자료 의존 관계에 있는 명령어들이 다른 연산처리기에 할당된 경우, 이들로부터 레지스터 파일을 통하거나 바이패싱을 통하여 자료를 전달받는 경우의 수행완료 시간과 2)필요한 명령어를 복사하여 중복 수행할 경우의 명령어 실행 완료 시간을 비교하여 2)의 결과가 1)에 비하여 나은 경우에 명령어의 중복 스케쥴링을 허용하도록 하는 것이다.

명령어 그래프 $G=(D,I)$는 명령어 집합 $I=\{i_1, i_2, \ldots, i_n\}$ 과 자료 의존성을 표시하는 방향성 있는 간선(edge)을 나타내는 집합 $D=\{d_{ij} \mid 1\le i,j\le n\}$ 로 구성된다. 연산처리기 집합은 $F=\{F_1, F_2, \ldots, F_q\}$ 로 표시하기로 한다.

알고리즘은 먼저 명령어간의 자료 의존성에 기초하여 병렬로 수행될 수 있는 명령어들을 하나의 그룹으로 묶는다. 같은 그룹내의 명령어들은 임의의 순서로 실행될 수 있다. 명령어를 그룹으로 구분하기 위해 간선의 길이를 1이라 하고 명령어 그래프의 뿌리 노드(root node)로부터 어떤 노드에 이르는 경로의 길이 중에서 가장 긴 값을 그 노드가 속한 그룹을 나타내는 레벨로 이용하였다.

$$l_i = \max\{l_j \mid \forall\ i_j \ \text{ such that } d_{ji}\neq 0\} + 1$$

즉, 노드(명령어) i의 레벨은 노드 i의 부모 노드의 레벨 값 중에서 가장 큰 값에 1을 더한 값이 된다.

명령어의 그룹을 결정한 후 각 그룹에 속한 명령어를 어떤 연산 처리기에 할당하는지를 설명하기 위해 몇 가지 기호에 대해 정의한다. 그림 3에서와 같이 명령어 i_i를 어떤 연산처리기 F_k에 할당하려고 할 때, 명령어 i_i에 직접적인 자료 의존 관계로 영향을 주는 명령어들의 집합을 G_i라고 하고 G_i에 속하는 명령어 i_i의 수행 시작을 가장 늦추는 명령어를 i_f라고 한다. μ는 어떤 명령어에 대해 그 명령어가 할당된 연산처리기를 나타내는 함수이다. 명령어 i_i가 연산처리기 F_k에 할당되는 경우 $\mu(i)=k$ 로 나타낸다. 또한, $\psi_i^{\mu(i)}$를 명령어 i_i가 연산처리기 $F_{\mu(i)}$에서 실행 단계를 완료한 시간 이라고 한다. $\delta(F_{\mu(i)}, F_{\mu(j)})$를 명령어 i_i 가 실행된 연산처리기 $F_{\mu(i)}$ 로 부터 명령어 i_j가 실행될 연산처리기 $F_{\mu(j)}$로 결과가 전달되는데 필요한 통신 시간이라고 한다.

명령어 i_i를 연산처리기 F_k에 할당하는 것과 관련하여 실행 단계의 시간을 조사해 보기로 한다. 정의로부터 명령어 i_i와 자료 의존 관계로 영향을 주는 명령어의 집합 G_i에 속한 명령어 i_f에 의하여 명령어 i_i 의 실행을 가장 늦춘다고 하면 i_f의 실행이 완료된 후 그 결과가 명령어 i_i에 전달되어야 하므로 명령어 i_i가 연산처리기 F_k에서 실행단계에 들어가는 시간 β_i는 그림 3에 보인 것과 같이 $\beta_i=\psi_f^{\mu(f)}+\delta(F_{\mu(f)}, F_k)$ 이다.

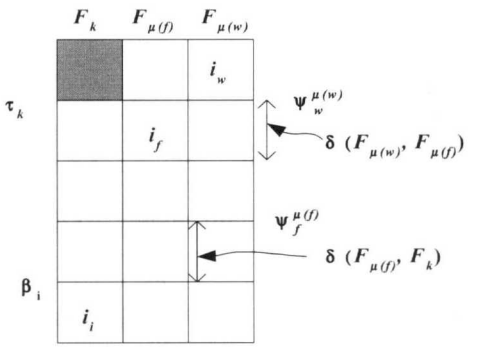


그림 3. 할당된 명령어의 실행완료 및 자료 전달 시간

여기서 $\psi_f^{\mu(f)}$는 명령어 i_f의 실행 단계가 완료되는 시간이며 $\delta(F_{\mu(f)}, F_k)$ 는 그 결과가 F_k로 전달되는데 걸리는 통신시간이다.

이 때, 자료 의존 관계가 있는 두 연산처리기 $F_{\mu(f)}$와 F_k 사이에 바이패싱 회로가 없다면, 연산처리기의 파이프라인 단계 WB가 종료된 후에야 다른 연산처리기의 파이프라인 EX단계에서 사용할 수 있다. 이 경우 필요한 통신 지연 시간을 파이프라인 1 사이클이라고 가정한다. 만일 두 연산처리기 간에 바이패싱 회로가 있다고 가정하면 두 연산처리기 간에는 파이프라인 사이클의 지연이 없이도 연속적인 계산이 가능하므로 통신시간을 0 사이클이라고 할 수 있다.

명령어 i_i가 연산처리기 F_k에서 실행단계에 들어가는 시간 β_i가 구해지면 연산처리기 F_k에 이미 할당되었던 명령어들의 실행이 완료된 시간과 β_i를 비교한다.

만일 τ_k가 β_i 보다 크거나 같다면 명령어 i_i의 실행단계는 이미 할당되었던 명령어의 실행단계가 완료된 후 바로 시작할 수 있으므로 명령어 i_i의 실행단계에 걸리는 시간 ex_i (1 사이클로 가정)를 τ_k에 더하면 명령어 i_i가 연산처리기 F_k에서 실행단계를 완료한 시간 ψ_i^k 를 구할 수 있다. 즉,

```
Algorithm : Duplicated Instruction Scheduling
  input : Instruction set I={i1, i2, ..., in}
          Data dependency set D={dij | 1≤i, j≤n }
          Functional unit set F={F1, F2, ..., Fq}
  output : VLIW codes
  BEGIN
  Let Ma , Ba be queues for every functional unit.  // a =1,2, . . . , q
  for every instruction ij do
  Determine the level of instruction.      // lj = max{ lk | for ∀ ik, dkj≠0} + 1
  end for
  Let Lm to be the set of instruction ij such that lj = m.
  for each group Lm do
  for every instruction ii ∈ Lm , 1≤ i ≤ | Lm | do
    Reset Ba , a =1,2, . . . , q .      // Queue Bk is an interim temporary queue.
     for every functional unit Fk do
         Find if ∈ Gi such that
                βi = ψf^μ(f) + δ( Fμ(f), Fk)     is the latest.
          if there exists bypassing between Fμ(f) and Fk
                then δ( Fμ(f), Fk)  = 0
                else δ( Fμ(f), Fk)  = 1
   // βi is the time when the communication between if and ii is completed.
   // τk is the completion time of the instructions already allocated to Fk.

        if τk ≥ βi    // if ii can not be allocated to Fk before βi
           then
           // Duplicating if to Fk does not make any good.
             Put instruction ii to queue Bk .
             ψi^k = τk + exi
           else
   // Check that duplicating instruction if to Fk makes the completion time earlier.
             Find iw ∈ Gf such that
                  twk = ψw^μ(w) + δ( Fμ(w), Fk)     is the latest.
             if   βi ≥ max { τk , twk + exf }
                then
                  Put instruction if to queue Bk .
                  Recalculate βi^re
                  Put instruction ii to queue Bk .
                  ψi^k = βi^re + exi
                else
   // Duplicating instruction if to Fk does not make any good.
                  Put instruction ii to queue Bk .
                  ψi^k = βi + exi
           end if
        end if
    end for
    Find functional unit Fb with the min{ ψi^k | k=1, . . . , q   }.
    Copy queue Bb to queue Mb.
  end for
  end for
  END
```

그림 4. 명령어 중복 스케줄링

$\phi_i^k = \tau_k + ex_i$ 이다.

만일 시간 τ_k가 β_i보다 이른 시간($\tau_k < \beta_i$)이라면 명령어 i_f를 연산처리기 F_k에 중복 할당하는 것이 명령어 i_i의 실행 완료 시간을 단축시킬 수 있는지를 검사한다. 중복 할당 여부는 명령어 i_i를 연산처리기 F_k에 할당하고자 할 때 1) 명령어 i_i에 자료 의존 관계로 영향을 주는 명령어들이 다른 연산처리기에 할당된 경우, 이들로부터 레지스터 파일을 통하거나 바이패싱을 통하여 자료를 전달받는 경우의 명령어 i_i의 실행단계 진입 시간 β_i와 2) 필요한 명령어를 연산처리기 F_k에 복사하여 수행한 후 그 복사된 명령어가 실행 단계를 완료한 시간을 비교하여 2)의 결과가 1)에 비하여 나은 경우에 2)의 필요한 명령어를 복사하여 실행하도록 허용하도록 하는 것이다.

2)의 필요한 명령어란 명령어 i_i를 가장 지연시키는 명령어 i_f를 의미하므로 연산처리기 F_k에 명령어 i_f가 복사될 경우 명령어 i_f가 실행단계에 진입할 수 있는 시간 tw_k는 명령어 i_f에 자료 의존 관계로 영향을 주는 명령어들의 집합 G_f 에서 i_f의 수행을 가장 지연시키는 부모 노드 i_w (즉 $i_w \in G_f,\ d_{wf} \neq 0$)가 실행 완료한 후 연산처리기 F_k로 자료가 전달되어 오는데 필요한 시간을 더한 값이 된다.

따라서 명령어 i_f가 연산처리기 F_k에서 실행 단계를 완료하는 시간은 tw_k에 i_f의 실행 시간을 더한 값이 된다. 이 값을 기존에 할당된 명령어들 중 마지막 명령어의 실행단계 완료 시간인 τ_k와 비교하여 이들 중 큰 값을 취하면 그 값이 명령어 i_i가 실행단계에 진입할 수 있는 시간이 된다. 여기서 그 값을 이미 구한 β_i (i_i 가 연산처리기 F_k에서 실행단계에 들어가는 시간)과 비교하여 만일 그 값이 β_i보다 크다면 명령어 i_f를 연산처리기 F_k에 복사하는 것이 실행시간 단축에 도움이 되지 않는다. 이 경우는 명령어 i_i만 연산처리기 F_k에 할당한다. 여기서 그 값이 β_i보다 작다면 명령어 i_f를 복사하는 것이 명령어 i_i의 실행을 앞당길 수 있으므로 명령어 i_f의 복사를 허용한다.

이 때 명령어 i_i를 가장 지연시키는 명령어 i_f를 복사하여 그 실행 단계가 완료되면 자료 의존성으로 명령어 i_i 에 영향을 주는 명령어의 집합 G_i에 속한 다른 명령어들로부터 자료 전달이 그 시간 안에 이루어질 수 있는지를 다시 검사한다. 즉 β_i를 다시 구하여 구한 값 β_i^{re}이 명령어 i_i를 실행 단계로 진입시키는 시점이 된다. 따라서 명령어 i_i의 실행단계 완료 시간은 $\phi_i^k = \beta_i^{re} + ex_i$ 가 된다.

위에서 설명된 방법을 통해 어떤 명령어가 연산처리기에 할당되어 그 명령어의 실행 단계가 완료되는 가장 빠른 시점을 모든 연산처리기의 각각에 대해 구할 수 있다. 그 들 중 가장 빠르게 실행을 완료할 수 있는 연산처리기에 명령어를 할당하면 된다. 이상의 과정을 정리한 것이 그림 4에 보인 알고리즘이다.

그림 4에 보인 알고리즘의 복잡도는 다음과 같다. 명령어 그래프 내의 모든 명령어의 전체 개수를 n이라 하고 연산처리기의 개수를 q라고 하면 각 명령어의 실행 단계 완료 시간을 모든 연산처리기에 대해 조사해야 하므로 $n \cdot q$ 회수만큼 실행 단계 완료 시간을 조사해야 한다.

이 때, 명령어 그래프에서 명령어 i_i에 대하여 i_i가 연산처리기 F_k에 할당되었을 때의 실행 단계 완료 시간 ϕ_i^k를 계산하기 위하여 명령어 i_i의 수행을 가장 지연시키는 명령어 $i_f \in G_i$를 찾아야 하며 i_f를 중복 할당할지 여부를 결정하기 위하여 명령어 i_f의 수행을 가장 지연시키는 명령어 $i_w \in G_f$를 찾아야 한다. 이 때, 명령어 i_f와 i_w를 찾기 위해 검색하여야 하는 명령어의 개수는 각각 명령어 i_i에 자료 의존 관계로 직접 영향을 주는 명령어의 개수와 명령어 i_f에 자료의존 관계로 직접 영향을 주는 명령어의 개수이다. 명령어 그래프 내의 모든 명령어의 전체 개수 n이 i_f와 i_w을 찾기 위해 검색하는 명령어 개수의 최대 값이 되므로 알고리즘의 시간 복잡도는 $O(n^2q)$이 된다.

긴 명령어는 같은 실행 단계 완료 시간을 갖는 명령어들이 하나의 긴 명령어를 이루고 복사된 명령어들이 긴 명령어의 일부를 자동적으로 차지한다고 가정하면 쉽게 만들 수 있다.

3.2 예제

3.1절에서 제안한 알고리즘을 예제에 적용하여 명령어 중복 스케쥴링의 효과를 보이기 위해 예를 통하여 살펴보기로 한다. 그림 5는 그림 1에 보인 여러 가지 바이패싱 토폴로지에 대하여 그림 5(a)와 같은 명령어 그래프로 표현되는 명령어를 스케쥴링 할 때 각각 명령어 중복 스케쥴링을 적용한 경우와

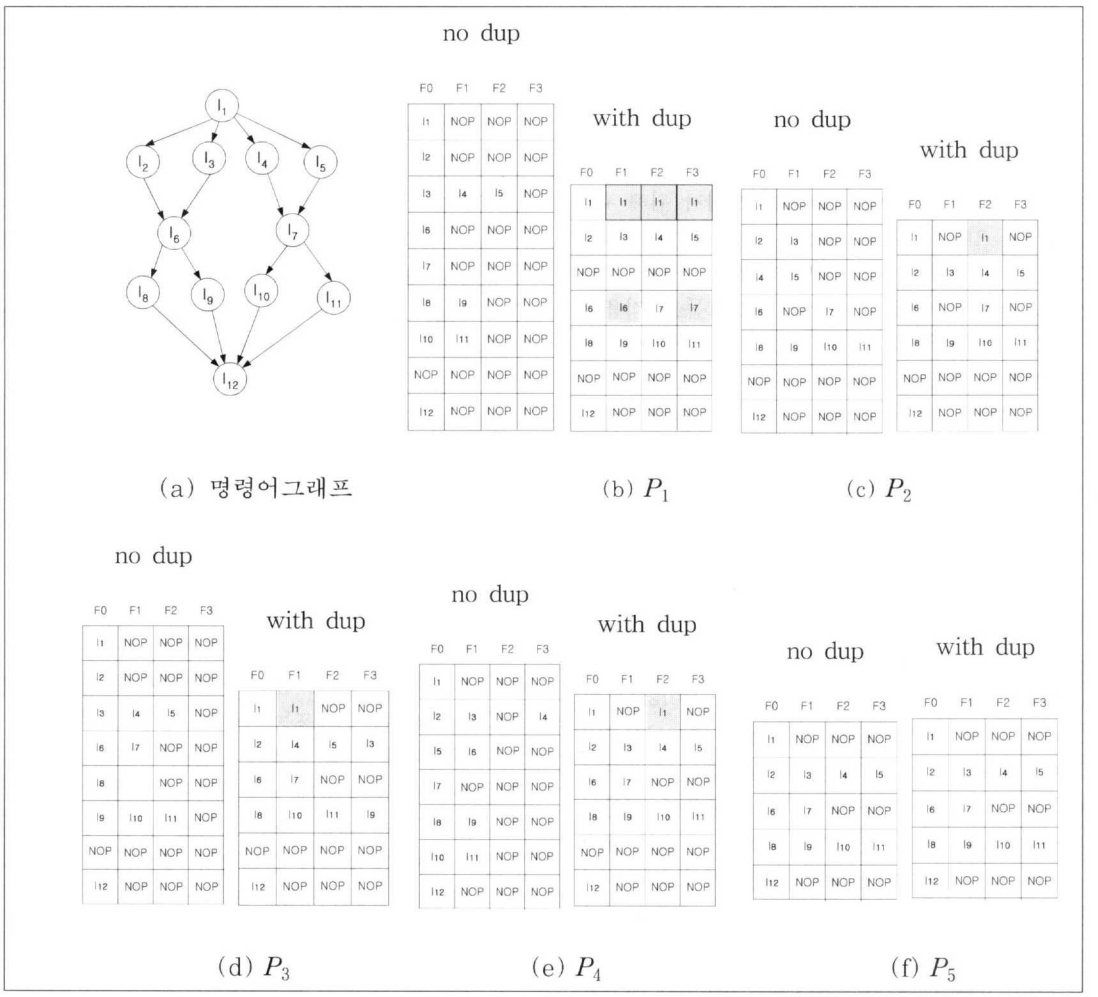


그림 5. 바이패싱 토폴로지별 명령어 중복 스케줄링 결과

적용하지 않은 경우를 보인 것이다.

그림 5(a)의 명령어 그래프에서 각 노드는 명령어를 표시하며 화살표 있는 간선은 각 명령어간의 자료 의존 관계를 표시하고 있다. 그림 5(b)에서 왼쪽 도표는 P_1 구조에서 명령어의 중복 스케줄링을 실행하지 않고 각 연산처리기 자체에서만 실행 결과가 바로 바이패싱되어 다음 명령어의 실행 단계에서 사용할 수 있도록 한 것이다. 이때의 실행 시간은 총 9 사이클이 걸렸다. 그림 5(b)의 오른쪽 도표는 P_1 구조에서 명령어 중복 스케줄링을 실행한 것으로 음영으로 나타낸 슬롯이 중복된 명령어가 할당된 곳으로 연산처리기 F_1, F_2 및 F_3의 첫 번째 빈 명령어 슬롯(NOP)에 명령어 I_2, I_3, I_4, I_5의 부모 노드인 명령어 I_1이 중복 스케줄링된 것을 보여준다.

P_1 구조의 각 연산처리기 사이에는 바이패싱 회로가 없으므로 명령어 I_1과 자료 의존 관계가 있는 명령어 I_2, I_3, I_4, I_5는 명령어 I_1을 중복 스케줄링함에 의하여 1 사이클 지연 없이 실행될 수 있다. 또 명령어 I_6과 I_7을 중복 스케줄링하면 명령어 I_6, I_7, I_8, I_9을 동시에 실행할 수 있어 한 사이클이 감소된다. 이와 같이 명령어 중복 스케줄링으로 인하여 전체 실행 시간이 7 사이클로 감소된다. 그림 5(f)의 P_5 구조는 모든 연산처리기 사이에 바이패싱 회로가 존재하는 완전 연결 구조이다. 여기서는 명령어 중복 스케줄링의 유무에 관계없이 5 사이클의 빠른 실행 시간을 얻을 수 있었으나 명령어 중복 스케줄링 효과는 얻을 수 없었다.

이는 바이패싱 회로가 복잡하여짐에 따른 하드웨

어적 지원으로 인해 명령어 중복 스케쥴링의 효과가 나타나지 않은 것으로 보인다.

Ⅳ. 실험 및 고찰

명령어 중복 스케쥴링의 효과를 보이기 위해 4개의 정수형 연산처리기로 구성된 타겟 머신을 대상으로 제안한 명령어 중복 할당 기법의 성능을 평가하였다. 명령어 중복 스케쥴링 알고리듬을 적용할 때 성능에 영향을 끼칠 수 있는 요소로서 명령어 수준 병렬성, 명령어들 간의 자료의존성의 빈도, 그래프의 크기를 고려하였으며 이들 요소들에 따라 다양한 명령어 그래프를 발생할 수 있는 프로그램을 개발하여 실험하였다.

4.1. Target 머신

실험에서 가정하는 타겟 머신은 동일한 연산처리기로 구성되며 각 연산처리기는 IF(instruction fetch), ID(Instruction Decode), EX(Execute)와 WB(Write Back)의 4단계로 명령어를 처리하는 파이프라인을 갖는 구조라고 가정하였다. 각 연산처리기 파이프라인의 매 단계는 한 사이클을 점유하며 EX단계에서 다른 연산처리기에서 수행중인 연산의 결과를 필요로 할 때 바이패싱 회로가 존재할 때에는 다른 연산처리기에서 WB단계를 거치기까지 기다릴 필요 없이 다른 연산처리기의 EX 단계의 결과를 피연산자로 EX단계에서 사용할 수 있다. 바이패싱 회로가 없을 때 다른 연산처리기로부터 피연산자를 가져와야 하는 경우에는 그 연산처리기가 WB 단계를 수행한 후에 그 결과를 사용할 수 있으므로 EX 단계가 한 사이클 지연되는 것으로 가정하였다. 즉, 피연산자를 필요로 하는 EX가 다음 사이클에 수행되어야 하므로 NOP 명령어가 삽입되게 된다.

또한, 타겟 머신은 명령어 처리를 파이프라인 방식으로 처리하므로 매 사이클마다 각 연산처리기가 IF, ID, EX, WB의 단계 중 동일한 하나의 단계를 수행하므로 명령어의 인출이나 피연산자의 인출 시 캐시미스가 발생하지 않는 것으로 간주하였다. 즉, 캐시미스가 발생하면 파이프라인이 정지되므로 이는 실험에서 캐시 또는 피연산자를 저장하는 레지스터 파일의 크기가 무한대임을 가정하는 것과 마찬가지 의미를 지닌다.

4.2 명령어 그래프 발생

실험에서 사용할 명령어 그래프는 하나의 기본 블록으로 구성된 그래프로 노드(node)는 명령어, 간선은 명령어간의 자료 의존 관계를 나타낸다. 실험의 대상이 되는 명령어 그래프를 구분하는 요소로서 다음과 같은 항목을 고려하였다.

그래프 길이는 명령어 그래프의 뿌리 노드에서 마지막 노드까지의 경로 중에서 가장 큰 값이다. 즉, 그래프의 시작 노드에서 각 명령어 i_i에 이르는 경로의 최대 길이를 위에서 정의한대로 그 명령어가 속한 그룹의 레벨이라고 하면 레벨의 최대값을 그래프의 길이 L로 정의하였다.

명령어 수준 병렬성 δ는 경로길이가 같은 노드들을 하나의 그룹이라고 했을 때, 각 그룹에 포함되는 명령어의 최대 값으로 이는 그래프 전체의 병렬성과 같은 값이다.

즉, $\delta = \max(\delta_l), \quad l = 1,2,\cdots,L,$

여기서 δ_l는 레벨이 l인 노드들의 개수이다.

명령어 그래프 밀도 d는 인접된 그룹에 속한 노드들 사이에 존재할 수 있는 최대 간선의 수에 대한 실제로 존재하는 간선 수의 비이다. 따라서 d가 100%이면 이 그래프는 주어진 명령어 수준 병렬성을 만족하되, 인접 레벨의 그룹에 속한 모든 노드들 사이에 자료의존관계가 존재하는 그래프이다. 만일 d가 0%이면 이는 결국 그래프 상에 어떤 노드간에도 자료의존관계가 존재하지 않으므로 결국 하나의 레벨로 구성된 그래프임을 의미한다.

4.3 실험 결과 및 분석

표 1은 L이 10인 경우에 δ를 3~5, d를 60%~80%로 변화시키면서 생성한 명령어 그래프를 대상으로 스케쥴링한 결과이다. 각 경우에 대하여 주어진 특성 값에 대해 무작위로 30개 씩 명령어 그래프를 생성시켜 실행 시간의 합계를 구한 것이다.

표 1을 토대로 각 구조별로 기존의 스케쥴링 기법을 적용한 결과를 기준으로 명령어 중복 스케쥴링 기법을 적용하였을 때의 성능을 측정하기 위하여 상대적 속도 향상을 아래의 수식(1)과 같이 정의하였다.

$$\zeta_p = \frac{\tau_p^o}{\tau_p^d} \tag{1}$$

여기서 τ_p^o와 τ_p^d는 각각 Viper 구조 P_p에서 기존의 명령어 스케쥴링 기법과 명령어 중복 스케쥴링을 이용하여 명령어 그래프를 할당하였을 때의 실행 결과이다.

표 1. 여러 가지 명령어 그래프에 대한 실험 결과(unit time)

(a) d = 60%

병렬성 (δ)	바이패싱 토폴로지							
	P_1		P_2		P_4		P_5	
	org	dup	org	dup	org	dup	org	dup
3	441	407	359	350	318	312	305	305
4	492	393	365	324	336	305	274	274
5	534	452	438	391	347	346	244	244
6	544	436	420	348	344	315	246	246

(b) d = 70%

병렬성 (δ)	바이패싱 토폴로지							
	P_1		P_2		P_4		P_5	
	org	dup	org	dup	org	dup	org	dup
3	468	433	358	354	326	323	318	318
4	513	421	395	352	348	320	270	270
5	544	471	445	394	364	360	304	304
6	576	514	497	451	398	384	348	348

(c) d = 80%

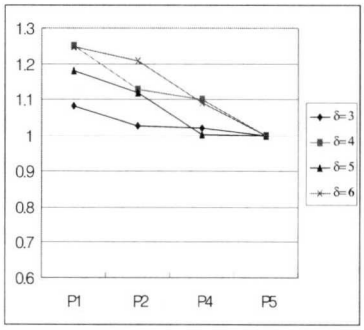
병렬성 (δ)	바이패싱 토폴로지							
	P_1		P_2		P_4		P_5	
	org	dup	org	dup	org	dup	org	dup
3	494	458	374	368	338	335	325	325
4	521	427	396	353	363	324	266	266
5	565	514	409	388	396	395	306	306
6	553	489	443	412	384	381	331	331

그림 6은 수식(1) 의 ζ_p를 측정한 것이다. 그림에서 P_2구조와 P_3구조는 연결이 같은 구조이므로 P_3구조를 별도로 표시하지 않았다.

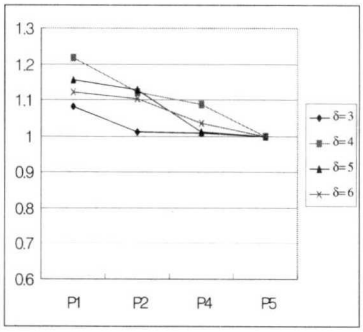
그림 6(a)은 자료의존성의 정도를 나타내는 밀도가 60인 경우에 병렬성을 변화시키면서 각 구조별로 상대적 속도향상(ζ_p)을 측정한 것이다. 실험 결과, 예상했던 바와 마찬가지로 병렬성에 관계없이 P_1구조에서 상대적 속도향상(ζ_1)이 가장 높았으며, P_5구조에 가까울수록 상대적 속도 향상이 1로 수렴하여 P_5에서는 상대적 속도향상(ζ_5)이 1이 된다. 이는 예상했던 바와 같이 모든 연산처리기간에 바이패싱 회로가 존재하는 P_5의 경우에는 명령어 중첩 스케쥴링의 효과가 전혀 없다. 이는 명령어 그래프 밀도가 70% 및 80%로 늘어나더라도 마찬가지임을 그림 6(b)과 (c)를 통해서 알 수 있다. 명령어 그래프 밀도가 70%와 80%인 명령어 그래프의 경우에 명령어 수준 병렬성이 4인 경우가 다른 경우에 비하여 상대적 속도향상이 가장 크게 나타났는데 이는 가정된 연산 처리기의 개수가 4이기 때문인 것으로 추정된다.

또한, 명령어 수준 병렬성이 4인 경우를 제외하고는 대체적으로 명령어 수준 병렬성이 증가할수록 상대적 속도 향상이 커지는 것을 알 수 있다. 이는 명령어 수준 병렬성이 높아짐에 따라 같은 레벨에 위치하는 명령어의 수가 증가하며 모든 연산 처리기들이 바이패싱 회로나 명령어 중복 스케쥴링에 의해 바로 다음 레벨의 명령어를 실행할 수 있는 개연성이 높아지기 때문이다. 따라서 간단한 바이패싱 회로를 가정한 머신에서 명령어 수준 병렬성이 큰 프로그램을 실행할 때 명령어 중복 스케쥴링의 효과도 증가하리라고 예상된다.

그림 7은 명령어 수준의 병렬성이 일정할 때 명령어 그래프 밀도 변화에 따른 상대적 속도 향상을 나타낸 것이다. 그림 7(a)과 (b)는 명령어 수준 병렬성이 각각 3과 4인 경우로 밀도의 변화에 따른 상대적 속도 향상의 변화가 큰 차이를 보이지 않는다. 이는 작은 병렬성을 갖는 프로그램에서는 자료의존성이 증가하더라도 상대적으로 연산처리기의 수가



(a) d = 60%



(b) d = 70%

그림 6. 명령어 수준 병렬성의 변화에 따른

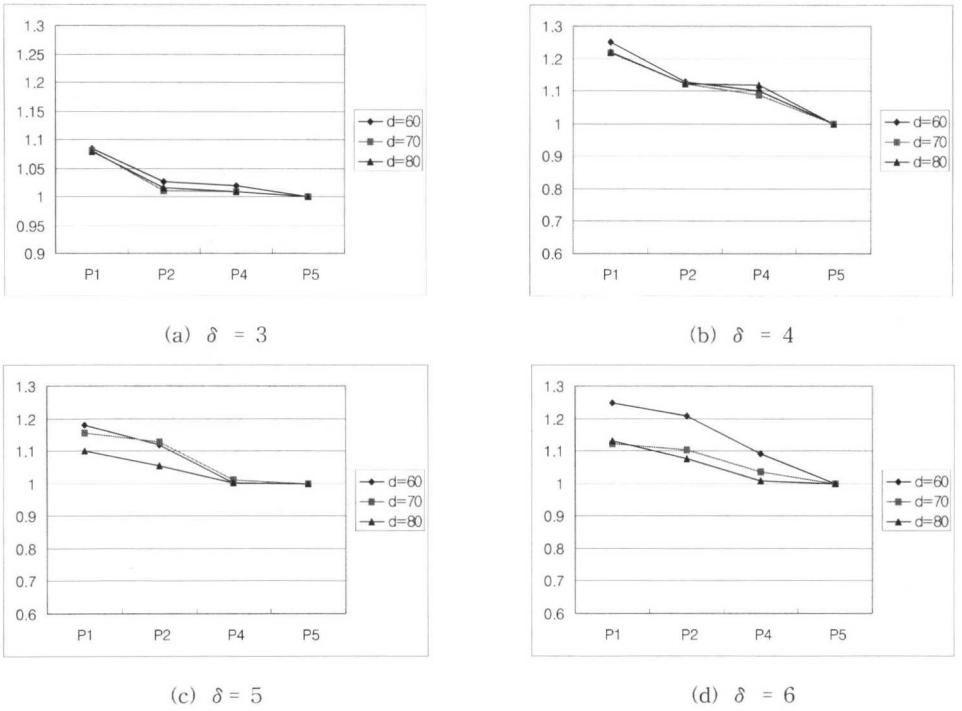


그림 7. 명령어 그래프 밀도 변화에 따른 속도 향상

많으므로 여분의 연산처리기에 명령어 중복 스케쥴링에 의한 가상의 바이패싱 회로가 많이 만들어지는 효과가 유지되어 지연이 발생할 소지가 적다. 그러나 그림 7(c)와 (d)에 나타난 것처럼 명령어 수준 병렬성이 각각 5와 6인 경우로 밀도가 높아질수록 상대적 속도 향상이 낮아지는 것을 알 수 있다.

이는 명령어 그래프 밀도가 높아짐에 따라 명령어간의 자료의존성이 높아 순차 처리해야 하는 명령어의 수가 늘어나기 때문인 것으로 판단된다.

V. 결 론

본 논문에서는 연산처리기 간에 정보를 직접 전달할 수 있는 바이패싱 회로가 있는 VLIW 구조에서 다양한 바이패싱 회로의 연결 구조를 가정하고 명령어 중복 스케쥴링 기법에 의하여 명령어 사이클이 줄어드는 지를 연구하였다.

특히, 간단한 형태의 바이패싱 회로를 갖는 구조에서 명령어 중복 스케쥴링에 의하여 복잡한 바이패싱 회로를 대신할 수 있는지를 연구하였다.

명령어 중복 스케쥴링 기법이란 자료 의존 관계에 있는 한 쌍의 명령어들이 서로 다른 연산처리기에 할당된 경우, 이들간에 필요한 피연산자를 바이패싱 회로를 통하여 전달받지 않고 피연산자를 생성하는 명령어를 해당 연산처리기에서 직접 실행하도록 함으로써 바이패싱 회로를 통해 전달받는 것과 같은 효과를 얻고자 하는 기법이다. 이 기법은 연속된 명령어간의 자료 의존 관계로 말미암아 불필요하게 추가된 빈 명령어(NOP) 대신 의미 있는 명령어를 중복 할당함으로써 파이프라인에서 피연산자를 레지스터 파일을 통해 가져오기 위한 지연을 없앰으로써 보다 빠른 계산을 가능하게 하여준다.

본 논문에서는 여러 가지 형태의 바이패싱 네트웍 토폴로지에 대해 명령어 중복 스케쥴링 기법을 적용하고 기존의 리스트 스케쥴링 기법을 적용한 경우와 비교하였다. 그 결과, 바이패싱 회로가 각 연산처리기내에만 존재하는 경우에 명령어 중복 스케쥴링 기법의 효과가 가장 큰 것을 알 수 있다. 특히 바이패싱 회로가 간단할수록 명령어 중복 스케쥴링이 효과적이므로 바이패싱 회로가 간단한 머신에서 명령어 중복 스케쥴링을 통해 실행 코드를 만들 경우 성능향상의 효과가 있었다.

연산처리기의 수가 증가하면 모든 연산처리기간에 바이패싱 회로를 추가하는 것은 하드웨어 비용

을 크게 증가시킨다. 집적도의 향상으로 연산처리기의 개수가 증가하는 추세를 고려하면 가상의 바이패싱 회로가 존재하는 것과 같은 효과를 얻을 수 있는 명령어 중복 스케쥴링은 매우 유효할 것이다. 즉 하드웨어의 제작비용을 낮추기 위하여 간단한 토폴로지의 바이패싱 회로를 가지도록 하되, 이로 인한 성능 저하를 방지하기 위하여 명령어 중복 스케쥴링 기법을 이용하여 보완하도록 하는 것이 바람직 할 것으로 판단된다. 앞으로는 서로 종류가 다른 연산처리기를 갖는 타겟 머신을 대상으로 명령어 중복 스케쥴링의 효과를 실험할 예정이다.

참 고 문 헌

[1] Boyoun Jeong, Joongnam Jeon and Sukil Kim, "Design of VLIW architectures minimizing dynamic resource collisions," *Journal of KISS*, Vol. 24, No. 4, pp. 357-368, 1997

[2] Sung-Hyun Jee, No-Kwang Park and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor," *Journal of IEEE Korea Council*, Vol. 1, No. 1, pp. 101-110, 1997

[3] Ruby B. Lee, "Subword Parallelism with MAX-2," *IEEE Micro*, Vol.16, No.4, pp.51-59, 1996

[4] B. R. Rau and J. A. Fisher, "Instruction-Level Parallel Processing: History, Overview and Perspective," *Journal of Supercomputing*, Vol.7, No.1/2, pp. 9-50, 1993

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

[6] Robert P. Colwell, Robert P. Nix and et al., "A VLIW architecture for a trace scheduling compiler, " *IEEE Trans. Computer*, Vol. 37, No. 8, pp. 967-979, 1988

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proc. 18th ISCA*, pp.266-275, 1991

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

[9] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Trans. Computer*, Vol.30, No.7, pp.478-490, 1981

[10] A. Aiken and A. Nicolau, "A Development Environment for Horizontal Microcode," *IEEE Trans. Software Engineering*, Vol.14, No.5, pp. 584-594, 1988

[11] Wen-mei W. and et al. "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", *Journal of Supercomputing*, Kluwer Academic Publishers, pp. 229-248, 1993

[12] B. Kuratrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, Vol.12, No.8, pp23-32, 1988

[13] S. Manoharan, "Augmenting work greedy assignment schemes with task duplication," *ICPADS '97*, pp. 772-777, 1997

[14] 문현주, 이승수, 김석주, 김석일, "NOP 명령어 슬롯을 활용하는 VLIW 코드 생성 기법", *정보과학회 추계학술대회 논문집*, 제27권, 제 2호, pp. 615-617, 2000

김 석 주(Suk-ju Kim) 정회원

1982년 2월 : 서울대학교 전자계산기 공학과 졸업
1984년 2월 : 한국과학기술원 전산학과(공학 석사)
현재 : 충북대학교 컴퓨터과학과 박사 과정
1997년 9월~현재: 혜천대학 조교수
<주관심 분야> 병렬처리

김 석 일(Sukil Kim) 정회원

1975년 : 서울대학교 전기공학과 (공학사)
1975~1990년 : 국방과학연구소 근무
1989년 : 미국 North Carolina 주립대학 (공학박사)
1990년~현재 : 충북대학교 컴퓨터과학과, 전기전자및컴퓨터공학부 교수
현재: 충북대학교 전기전자및컴퓨터공학부 학부장, 충북대학교 BK사업단장
<주관심 분야> 병렬컴퓨터 구조, 슈퍼컴퓨팅, 병렬처리언어, 시각장애인 사용자 인터페이스 등