論文 88-25-8-10

RISC 파이프라인 아키텍춰의 코드 최적화 알고리듬

(A Code Optimization Algorithm of RISC Pipelined Architecture)

金 殷 成*, 林 寅 七*

(Eun Sung Kim and In Chil Lim)

要 約

본 논문에서는 파이프라인 아키텍춰에서 수행되는 명령어들 간의 자원 종속 관계 때문에 발생하는 해저드를 효율적으로 해결하는 파이프라인 스케쥴링에 의한 코드 최적화 알고리듬을 제안한다. 동시에 수행되는 명령어들 간의 자원 상충에 기인하는 타이밍 해저드와 분기 타겟 결정을 위한 지연 시간 때문에 발생하는 시퀀싱 해저드를 파이프라인 인터록을 사용하지 않고 명령어 시퀀스를 재배열함으로써 처리해 준다. 타이밍 해저드와 시퀀스 해저드를 동시에 고려해 줌으로써 효율적인 코드를 생성할 수 있으며, 파이프라인이 동적으로 제어될 때 발생할 수 있는 구조적 해저드를 함께 고려하여 프로그램의 동적 수행 속도를 향상시킨다.

Abstract

This paper proposes a code optimization algorithm for dealing with hazards which are occurred in pipelined architecture due to resource dependence between executed instructions.

This algorithm solves timing hazard which results from resource conflict between concurrently executing instructions, and sequencing hazard due to the delay time for branch target decision by reconstructing of instruction sequence without pipeline interlock.

The reconstructed codes can be generated efficiently by considering timing hazard and sequencing hazard simultaneously. And dynamic execution time of program is improved by considering structral hazard which can be existed when pipeline is controlled dynamically.

I. 서 론

계속되는 반도체 기술의 발전으로 컴퓨터 아키텍춰의 성능 향상도 급속도로 이루어 지고 있으며, 특히 기판(board)이 아닌 단일 칩으로 구성되는 마이크로프로세서는 반도체의 기술 발전과 아키텍춰의 기술 혁신으로 탁상용 컴퓨터(desktop computer)및 워크스테이션(workstation)으로도 슈퍼미니급 이상의 컴퓨터가 보유하는 처리 능력을 가능케 하고 있다.[1]

좀 더 낮은 비용으로 고성능의 마이크로프로세서를 실현하기 위해 하드웨어 자원의 이용 효율을 극대화하고 대부분의 명령어를 단일 사이클로 수행할 수 있는 고속의 클럭 사이클을 갖는 RISC 기술이

*正會員 : 漢陽大學校 電子工學科
(Dept. of Elec. Eng., Hanyang Univ.)

接受日字 : 1988年 5月 24日

널리 이용되고 있다.[3],[6] RISC는 모든 명령어가 동일한 크기를 갖고 고정된 형식을 사용함으로써 명령어의 디코드 지연을 감소시키고 명령어들이 거의 일정한 데이타 패스(data path)를 거쳐 수행되므로 단일 사이클 동작이 가능하게 되어 많은 스테이지 수를 갖는 강력한 파이프라인 아키텍춰로 실현되어 고성능의 수행 능력을 갖추게 된다.[2],[3],[4]

파이프라인은 명령어 수행에 있어서 병렬성(parallelism)을 제공해 준다. 즉, 순차적인 명령어들이 서로 다른 스테이지를 점유하여 서로 독립된 하드웨어 부분을 사용함으로써 동시에 수행할 수 있게 된다. 한 명령어의 수행이 완결되는데 시간 T가 걸린다고 할 때, 파이프라인 시스템에서 한 명령어를 s 개의 단계로 나누어 수행한다면 시스템의 처리율(throughput)은 모든 파이프스테이지가 항상 사용 가능할 경우 순차적 시스템(sequential system)보다 s 배로 향상된다. 이상적인 파이프라인 성능을 매 스텝마다 명령어 수행을 개시하여 얻을 수 있지만, 실제 파이프라인 시스템에서는 동시에 수행되는 명령어들 간의 자원 상충에 의한 타이밍 해저드(timing hazard)와 분기 명령어를 수행하여 분기 타겟을 결정하기 위한 지연 시간 때문에 발생하는 시퀀싱 해저드(sequencing hazard)로 인하여 이상적인 성능 향상을 얻지 못한다.[5],[6]

어떤 명령어가 자기의 오퍼랜드로서 자기보다 선행하는 명령어에서 계산된 값을 사용하게 되는 경우에, 선행 명령어가 여전히 수행중에 있다면 그 결과값을 이용할 수 없게 되어 두 명령어 간에 데이타 의존 관계에 의한 타이밍 해저드가 발생한다. 타이밍 해저드가 발생하면 명령어가 자기의 오퍼랜드를 이용할 수 있을 때까지 그 수행을 지연시켜 프로그램을 수행한 결과가 프로그래머가 의도했던 것과 항상 일치될 수 있도록 해 주어야 한다. 파이프라인 인터록 메카니즘은 수행되는 명령어 시퀀스에서 타이밍 해저드를 검출한 후 파이프라인의 연속적인 수행을 일시 중지시켜 수행 명령어의 실행을 지연시켜 줌으로써 해결해 준다. 이 방법은 프로세서에 대한 하드웨어 오버헤드가 크고, 또한 올바른 수행 결과를 보장하기 위해 모든 명령에 대해서 검출을 위한 검사를 해야 하기 때문에 해저드의 존재 여부에 관계없이 모든 명령어에 오버헤드를 부과하게 된다. 그리고 해저드가 존재하는 경우에 있어서는 그 실행이 지연되기 때문에 연속적인 파이프라인 수행이 불가능하게 되어 프로그램의 수행 속도를 저하시키게 된다.[7]

이러한 복잡한 하드웨어를 사용하는 방법을 대신하여 인터록 메카니즘을 제거하고, 컴파일 시에 명령어들을 재배열하고, 필요한 경우 nop(no-operation)을 삽입하여 타이밍 해저드를 해결하는 방법이 제안되었다.[8]

한편 프로그램의 실행을 제어하는 분기 명령어는 프로그램을 수행할 때 사용되는 전체 명령어의 25~30%[9]를 차지하고 있으며, 특히 조건 분기 명령은 분기 타겟이 결정될 때까지 파이프라인을 일시적으로 중단시킬 수도 있기 때문에 파이프라인 성능에 중요한 제약이 되고 있다. 따라서 성능에 미치는 분기 명령의 영향을 감소시키기 위한 연구들이 활발히 진행되어 왔다.[10]~[13]

대부분의 이러한 연구들은 분기 예상을 하여 예상된 분기 타겟의 명령어를 수행하는 방식으로, 하드웨어를 사용하여 분기 예상 성공율을 높여 줌으로써 성능 향상을 꾀하고 있다. 분기 예상이 잘못된 경우에 있어서는 분기 명령어 다음에 실행되는 명령어를 무효화시키고 새로운 타겟의 명령어를 페치하여 수행하기 위한 방식 및 이를 위한 하드웨어가 필요하게 되며, 높은 분기 예상 성공율을 유지하기 위해 복잡한 알고리듬을 사용하고, 이를 실현하는 하드웨어의 부담이 크다는 단점이 있다.

지연 분기 방식(delayed branch)은 분기 명령의 수행으로 파이프라인 지연이 발생하게 되는 시퀀싱 해저드를 가장 적은 하드웨어를 사용하여 해결한다.[14]~[18] 즉, 분기 조건에 관계없이 분기 명령 후에도 다음 명령어들을 연속적으로 항상 수행하도록 하고, 컴파일 시에 명령어들을 재 구성하거나 지연 공간(delay slot)에 nop을 삽입하여 원하는 수행 결과를 보장해 주는 방식으로서, 지연 공간이 단일 사이클로 되어 있는 RISC 머신에서 널리 사용되고 있다.

소프트웨어 인터록에 의한 타이밍 해저드 및 시퀀싱 해저드의 해결은 단순하고 규칙적인 하드웨어를 사용함으로서 단일 VLSI 프로세서 칩 제조를 위한 중요한 요소가 되고, 하드웨어 오버헤드의 감소로 성능 향상을 얻을 수 있다. 그러나 컴파일 시에 명령어들을 재구성해야 되고, 그 결과가 항상 올바르게 수행되도록 보장해 주어야 하기 때문에 컴파일러의 부담이 커지게 되고, 필요한 경우 코드 시퀀스에 nop을 추가로 삽입해 주어야 하므로 코드 크기가 커지는 단점이 있다.

Thomas Gross는 타이밍 해저드와 시퀀싱 해저드를 분리하여 처리하는 방법을 제안하였다.[8],[16] 즉, 타이밍 해저드 해결을 위한 코드 스케쥴링을 한 후에 시퀀싱 해저드를 처리하여 지연 분기의 최적화를 수행하고 있으나, 알고리듬이 복잡하고 지연 분기의 최적화는 타이밍 해저드 해결을 위해 재구성된 코드 시퀀스에 적용하게 되므로 많은 제약을 갖게 되어 생

성된 코드가 비효율적이다.

Patterson 이나 Radin 등은 peephole optimization 에 의해 시퀀싱 해저드를 처리하는 방법을 제안하였다.[10],[15] 이 방법은 간단하기는 하나 최적화될 수 있는 여지가 많고, 타이밍 해저드는 인터록 메카니즘을 사용하여 해결하므로 고려하지 않고 있어서 파이프라인 수행 성능이 떨어지게 된다.

본 논문에서는 파이프라인 아키텍춰에서 발생하는 타이밍 해저드와 시퀀싱 해저드를 동시에 고려하여 해결하는 파이프라인 스케쥴링에 의한 코드 최적화 알고리듬을 제안한다. 기본 블럭에 속한 명령어 시퀀스를 읽어 자원 종속 관계를 나타내는 방향성 비순환 그래프인 RDG(resource dependent graph) 를 작성하고, 노드와 에지에 각각 인터록 가중치 I와 누적 가중치 C 를 주고 이 가중치에 의해 생성될 코드 시퀀스를 역방향으로 선택하여 구한다. 또한 동적 특성을 갖는 구조적 해저드(structral hazard) 를 코드 최적화의 수행 시 함께 고려하여 프로그램의 동적 수행 속도를 향상시킨다.

Ⅱ. 해 저 드

명령어들을 동일한 시간 사용의 패턴으로 동시에 수행하는 정적 파이프라인에서 타이밍 해저드와 시퀀싱 해저드가 발생하게 되고, 그 해결 방법에 따라 파이프라인의 복잡도 및 성능에 많은 영향을 미치게 된다. 또한 수행되는 명령어나 사용되는 데이타에 따라 점유되는 파이프스테이지가 달라 질 수 있는 동적 파이프라인에서는 파이프스테이지의 다양한 사용 때문에 발생되는 구조적 해저드를 처리해 주어야 한다.

1. 타이밍 해저드

n 개의 명령어를 동시에 수행할 수 있는 파이프라인 시스템은 단일 스텝으로 동작되는 순차적 시스템보다 그 성능을 n배로 향상시킬 수 있으나 실제적으로는 가능하지 않다. 즉, 최대 s배의 처리율로 수행된다는 것은 동시에 수행되는 n 개의 명령어에 대한 n^2 개의 스텝들이 서로에 대해 완전히 독립적이 되어야 함을 의미하며, 실제 명령어 스트림을 수행할 때 이 목표를 달성할 수 없다. 그림 1 은 명령어 상호간에 데이타 종속 관계를 갖는 경우를 보여 준다. 한 명령어가 선행 명령어에서 계산된 값을 참조할 때, 선행 명령어가 수행중이기 때문에 그 값을 이용할 수 없게 되므로 파이프라인 타이밍 해저드가 발생하게 된다.

(정의 1) i를 명령어라 하고 ps를 파이프스테이지라 하자. 명령어 i가 사이클 ps 동안에 자원 r의 값을 읽는다면, 자원 r은 (i, ps) 에서의 소스 필드(source field) 라 한다. 만약 명령어 i의 ps 사이클 동안에 r이 기입된다면, 자원 r은 (i, ps) 에서의 데스티네이션 필드(destination field) 라 한다.

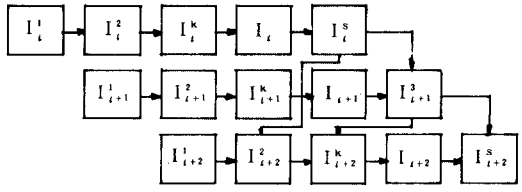


그림 1. 데이타 종속 관계를 갖는 명령어들의 병렬 수행

Fig. 1. Parallel executing instructions with resource conflict.

(정의 2) 명령어 i_2 는 명령어 i_1 보다 후에 그 수행이 시작되고 스테이지 (i_2, ps_2) 의 위상이 (i_1, ps_1) 보다 앞서거나 같다고 할 때, 자원 r이 (i_1, ps_1) 에서의 데스티네이션 필드이고 또한 (i_2, ps_2) 에서 소스 필드로 사용된다면 자원 r에 대해 명령어 i_1, i_2 는 데스티네이션-소스 파이프라인 상충된다고 하고, 자원 r이 (i_1, ps_1) 에서의 소스 필드이고 또한 (i_2, ps_2) 에서 데스티네이션 필드로 사용된다면 이들 명령어는 자원r에 대해 소스-데스티네이션 파이프라인 상충된다고 한다.

데스티네이션-소스 상충은 파이프라인 구조에서 발생할 수 있는 타이밍 해저드 중 가장 일반적인 것이다.

(정의 3) 2 개의 명령어가 모두 같은 파이프스테이지 동안에 동일한 레지스터 r에 기입하려고 할 때, 자원 r에 대해 이들 명령어는 데스티네이션-데스티네이션 파이프라인 상충된다고 한다.

데스티네이션-데스티네이션 상충이 발생되었을 때 첫번째 논리적 동작 수행의 결과가 절대로 사용될수 없어야 한다. 그렇지 않으면 소스-데스티네이션 상충이 발생할 수 있기 때문이다. 데드 코드 제거(dead code removal) 에 의해 데스티네이션-데스티네이션 상충은 발생될 경우가 흔치 않지만, 코드 스케쥴링 시 레지스터의 상태를 확정해 주지 못하는 이러한 상충을 반드시 제거해 주어야 한다.

타이밍 해저드는 자원 상충에 의해 발생하는 것이다. 본 논문에서는 명령어 레벨에서 파악할 수 있는 자원인 레지스터만을 고려한다. 타이밍 해저드는 버스 충돌(bus contention) 이나 캐쉬 미스(cache miss) 등의 상충에 의해서도 발생할 수 있다. 이러한 상충

은 명령어 레벨에서는 파악될 수 없으며 동적 특성을 갖고 있기 때문에 하드웨어에 의해 프로세서에서 해결해 준다고 가정한다. 이러한 고려는 실제 적용에 아무런 제약도 주지 않는다.

2. 시퀀싱 해저드

파이프라인 시스템에서 현재 수행되는 명령어가 분기 명령어라면 분기진행 방향이 결정되기 전에 수행되는 후속 명령어는 실제로 수행되어야 할 분기 명령어의 다음 명령어가 아닐 수도 있게 되며, 이 경우에는 시퀀싱 해저드가 존재하게 된다. 시퀀싱 해저드를 발생하는 분기 명령어는 제어의 흐름을 변경시키는 명령어로서 조건 및 무조건 분기, 트랩(trap) 명령어나 감독자 호출(supervisor call), 프로시듀어나 함수 호출 등이 이에 속한다.

시퀀싱 해저드는 프로그램 계수기(program counter)에 관련된 자원 상충이다. 분기 명령어는 프로그램 계수기의 현재 값을 참조하고, 이 값을 근거로하여 새로운 프로그램 계수기 값을 계산한다.

(정의4) 명령어 i가 타겟 T를 갖는 분기 명령어라 하자. 분기 명령어의 실행을 개시한 후에 타겟 T의 명령어를 수행하기 전에 n개의 명령어가 차례로 수행된다면, 즉, 실행되는 명령어의 순차 흐름이 i, i+1,…, i+n, T라고 하면 이 분기를 지연 n의 지연 분기(delayed branch) 라 한다.

(정의5) 현재 수행 중인 명령어에 지연이 발생하는 경우, 지연되는 사이클 시간 만큼의 시간적 공간을 지연 슬로트(delay slot) 라 한다.

지연 분기 방식은 시퀀싱 해저드에 상관없이 계속적으로 명령어를 수행해 간다. 프로그램 계수기는 지연 슬로트 수 n만큼의 명령어가 페치(fetch)될 때까지 수정되지 않고, 새로운 타겟으로부터 페치되는 명령어의 실행은 사이클 시간 n+1 이후에 개시된다. 하드웨어 인터록을 갖춘 아키텍춰는 분기 결정이 날 때까지 지연 슬로트 만큼의 사이클 시간을 낭비하거나, 분기 결정을 예상하여 예상된 타겟의 명령어를 먼저 수행해 간다. 후자의 경우에서는 예상이 잘못된 경우에만 사이클 낭비가 된다. 지연 분기 방식은 지연 슬로트를 그 이후에 side-effect 를 발생하지 않는 명령어로 대체하고, 그렇지 못한 경우에만 대체되지 않은 사이클 수 만큼의 낭비가 초래된다.

3. 구조적 해저드

구조적 해저드는 동일하거나 상이한 기능이나 동작을 위한 두개의 실행이 최소한 한 파이프스테이지에서 충돌될 때 존재하게 된다. 타이밍 및 시퀀싱 해저드와 마찬가지로 파이프스테이지의 다양한 사용 때문에 발생되는 구조적 해저드도 파이프라인 시스템에서 반드시 해결되어야 한다.

그림2의 파이프라인 시스템을 생각해 보자. 메모리를 사용하지 않는 명령어의 실행은 4단계로 나누어 진다(그림2(a)). 처음 단계에서 명령어를 페치하고, 두번째 단계에서 이를 디코드하고 필요한 오퍼랜드를 읽어 들인 후, E-스테이지에서 오퍼랜드를 갖고 요구되는 계산을 하고, 그 결과를 W-스테이지로 넘긴다. 그림2(b)는 메모리를 로드(load)하는 명령의 실행을 나타낸다. E-스테이지에서 유효 주소를 계산하여 M-스테이지에서 계산된 메모리 위치에서 데이타를 인출하여 W-스테이지로 넘긴다. 그림2(c)는 메모리에 스토아(store)하는 명령의 실행으로써 M-스테이지를 반복 사용한다. 계층적 메모리 구조를 사용하는 시스템으로서 외부 캐쉬를 write-back 캐쉬로 사용하는 경우, 한 사이클은 캐쉬 내의 위치를 검사하고 다른 한 사이클은 스토아를 하기 위해 사용한다면 스토아 명령을 수행할 때 메모리 억세스를 위해 2사이클이 필요하게 된다. 스토아 명령 다음에 오는 메모리를 억세스하는 명령어는 최소한 1사이클 이상 분리되어 있어야 한다. 그렇지 않으면 이들 명령어 간에는 M-스테이지의 사용으로 인한 충돌이 일어나게 된다. 이러한 상충을 구조적 해저드라 한다. 그림3은 명령어 간에 구조적 해저드가 발생한 경우에 대한 예이다.

캐쉬 미스나 버스 충돌과 같은 일반적인 형태의 구조적 해저드는 그 동적 특성 때문에 발생 여부를 예측하기가 매우 어렵다. 따라서 해저드가 검출될 때

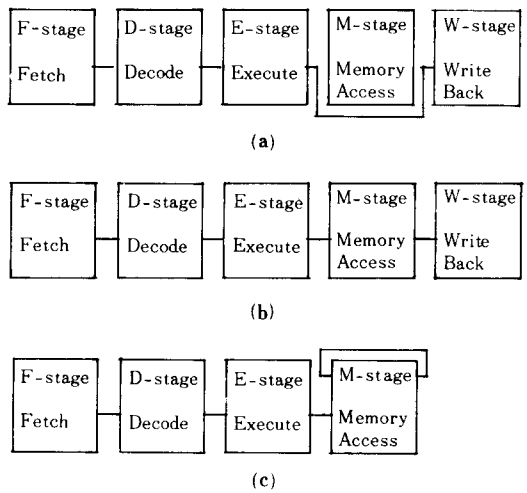


그림 2. 동적으로 제어되는 파이프라인
Fig. 2. Dynamically controlled pipeline.

Instruction	Cycle							
	1	2	3	4	5	6	7	8
store	F-stage	D-stage	E-stage	M-stage	M-stage			
					****			
load		F-stage	D-stage	E-stage	M-stage	W-stage		
store			F-stage	D-stage	E-stage	M-stage	M-stage	
							****	
store				F-stage	D-stage	E-stage	M-stage	M-stage

그림 3. 구조적 해저드의 예
Fig. 3. Example of structral hazard.

마다 모든 파이프라인 수행을 중단시킨 후 해저드를 처리하고 나서 다시 수행해 간다.

본 논문에서는 그림 3의 예와 같이 예측이 가능한 구조적 해저드를 고려하여 스케쥴링을 수행하여 프로그램의 동적 수행 속도를 향상시킬 수 있도록 한다.

Ⅲ. 코드 최적화

인터록을 대신하여 코드 스케쥴링에 의해 해저드를 처리하는 코드 최적화는 머신 코드를 출력하는 코드 생성의 다음 단계에서 수행한다. 따라서 컴파일러에 의해 출력된 코드나 프로그래머가 작성한 어셈블리어 코드에 모두 적용할 수 있다.

(정의 6) 자원 종속 그래프(RDG) $G=\{N, E, H, C\}$는 가중치가 주어진 방향성 비순환 그래프라 한다. 여기서 N은 명령어에 대응하는 노드의 집합이고, E는 데이타 종속 관계를 갖는 노드쌍을 나타내는 에지의 집합이고, H는 에지 E에 대해 계산되는 가중치이고, C는 노드 N에 대해 계산되는 가중치이다.

(정의 7) 시점에서만 진입점(entry point)을 갖고 종점에서만 분기할 수 있고, 내부 루프를 갖지 않는 명령어 시퀀스를 기본 블럭이라 한다.

주어진 명령어 스트림은 먼저 기본 블럭들로 나누어지고, 각 기본 블럭의 명령어 시퀀스에 대응하는 자원 종속 그래프가 생성된다. 에지 e_{ij}는 노드 n_i에서 노드 n_j로의 방향성이 주어진 노드쌍 (n_i, n_j)이라 할 때, 에지 e_{ij}에는 노드 n_i, n_j 간의 지연 슬로트 수에 근거한 가중치 $H(e_{ij})$가 주어진다. 노드 n_i에는 리프 노드로부터 노드 n_i에 이르는 패스상에 속하는 에지들의 가중치의 누계된 값인 임계 가중치 C_i가 주어진다.

코드 최적화를 위한 알고리듬은 다음과 같다.

〔알고리듬 1〕 코드 최적화

```
Code Optimization (명령어 스트림) {
  for(각 기본 블럭에 대해) {
    자원 종속 그래프의 구성 ;
    각 노드에 가중치 할당 ;
    가중치에 의한 노드 선택 ;
  }
}
```

알고리듬 1은 다음과 같은 여러 개의 알고리듬으로 구성된다.

〔알고리듬 2〕 자원 종속 그래프의 구성

```
Build RDG(기본 블럭) {
  for(코드 시퀀스의 순서에 따른 각각의 명령어에 대해) {
    이 명령어에 대응하는 노드를 생성하여 RDG에 추가 ;
    생성된 노드에 노드 번호 부여 ;
    if(추가된 명령어에서 정의되는 레지스터가 존재하면) {
      추가된 명령어를 출발 노드로 하여 이전에 생성된 노드중 최종
        정의된 레지스터를 사용하는 노드들과 에지로 연결 ;
      연결된 에지에 가중치 H(e_ij) ←0 을 할당 ;
    }
    for(추가된 명령어가 사용하는 각각의 레지스터에 대해) {
      이 레지스터를 최종 정의한 명령어와 에지로 연결 ;
      연결된 에지에 가중치 H(e_ij)←(두 노드간의 지연슬로트수+1)
        를 할당 ;
    }
  }
}
```

〔알고리듬 3〕 각 노드에 대한 가중치 할당

```
Weight Assignment(RDG) {
  for(각 노드 n_i 에 대해)
    노드 n_i 의 임계 가중치 C(n_i)←0 으로 초기화 ;
  RDG에 대한 토폴로지칼 정렬 (sorting) 을 수행 ;
  for(토폴로지칼 정렬의 역순에 의한 각 노드 n_i 에 대해)
    for(에지의 가중치 H(e_ij)를 갖는 각각의 에지 e_ij 에 대해)
      C(n_i)←MAX(C(n_i), C(n_j)+H(e_ij)) ;
}
```

토폴로지칼 정렬에 역순을 사용하는 이유는 노드 n_i의 자식이 되는 노드 n_j에 대한 임계 가중치가 구해진 후에 노드 n_i에 대한 임계 가중치를 구할 수 있기 때문이다.

(정의 8) 주어진 자원 종속 그래프 RDG에 대한 코드를 생성할 때, 생성된 코드에 대응하는 노드는 마크(mark) 되었다고 한다.

(정의 9) 자원 종속 그래프 RDG에서 자기의 선행자들이 모두 마크된 노드를 후보자 노드라고 한다.

(정의10) 명령어 i_m은 명령어 i_n의 선행 노드이고, i_n의 모든 선행자들이 마크되어 있고 i_n을 스케쥴링하려고 할 때, i_m 후에 마크된 명령어의 수를 L이라

하면 L-H(e_{mm})+1≧0 일 때 함수 Timing(i_n, i_m, L)은 true가 된다.

함수 Timing(i_n, i_m, L-1)=false 이고 Timing(i_n, i_m, L)=true 이면, 명령어 i_n과 i_m 사이의 지연 슬로트 수는 L 이라고 말할 수 있다. 즉, 명령어 i_n과 i_m은 명령어 스트림에서 최소한 L사이클 이상 떨어져 있어야 한다.

(정의11) 후보자 노드로서 이전에 이미 마크되어 있는 자신의 모든 선행 노드와 함수 Timing을 만족시키는 후보자 노드들의 집합을 후보자 그룹이라 한다.

(정의12) 명령어 i_m은 명령어 i_n의 선행 노드이고, i_n의 모든 선행자들이 마크되어 있고, 구조적 해저드에 의한 이전 명령어와의 지연 슬로트 수가 S인 i_n을 스케줄링하려고 할 때, i_m 후에 마크된 명령어의 수를 L이라 하면 L-S≧0 일때 함수 Structral(i_n, i_m, L) 은 true가 된다.

명령어 i_n과 i_m간에 구조적 해저드가 존재할 때 해저드의 영향을 받지 않을 만큼 두 명령어를 떨어지게 하여 해저드를 해결할 수 있다. 그러나 코드의 재구성시 지연 슬로트에 들어갈 명령어가 없는 경우에, nop을 삽입해 줄 필요가 없다. 하드웨어 메카니즘에 의해 해저드가 해결될 때까지 프로세서가 임시 중단되기 때문이다. 즉, 코드 스케줄링을 수행할 때 구조적 해저드가 존재하는 두 명령어는 가급적 지연 슬로트 만큼 멀리 떨어지도록 한다.

〔알고리듬 4〕 가중치에 의한 노드 선택

```
Select Node (RDG) {
  후보자 그룹을 구한다 ;
  if(후보자 그룹에 분기 명령어 n_b가 속해 있으면) {
    if({후보자 그룹-n_b}==NULL) {
      n_b를 선택 ;
      후보자 그룹에서 n_b를 제거 ;
      RDG에서 n_b를 마크 ;
    }
    else
      분기 지연 최적화 ;
  }
  while(RDG의 모든 노드가 마크될 때까지) {
    후보자 노드이면서 Timing(n_j, n_k, L)=true인 노드 n_j를
      후보자 그룹에 추가 ;
    if(후보자 그룹==NULL)
      nop을 삽입 ;
    else {
st haz sol :
      if(후보자 그룹에 H(e_ij)가 가장 큰 노드 n_i가 하나이면)
        n_i를 선택 ;
      else if(n_i가 여러개이면)
        임계 가중치 C(n_i)가 가장 큰 노드 n_i를 선택 ;
      if(Structral(n_i, n_h, L)==true) {
        후보자 그룹에서 n_i를 제거 ;
        RDG 에서 n_i를 마크 ;
      }
      else if({후보자 그룹-n_i}==NULL) {
        후보자 그룹에서 n_i를 제거 ;
        RDG 에서 n_i를 마크 ;
      }
      else{
        후보자 그룹에서 n_i를 제거 ;
        구조적 그룹에 n_i를 추가 ;
        struct←1 ;
        goto st haz sol ;
      }
      if(struct==1) {
        struct←0 ;
        후보자 그룹←후보자 그룹+구조적 그룹 ;
      }
    }
  }
}
```

(정의13) 어떤 한 명령어가 후속 명령어와의 해저드 발생으로 갖을 수 있는 최대 지연 슬로트를 잠재성 슬로트라 한다.

〔알고리듬 5〕 분기 지연 최적화

```
Branch Optimization(n_b) {
  분기 명령어 n_b의 지연 슬로드 d를 결정 ;
  for(i=1 ; i<=d ; i++) {
    {후보자 그룹-n_b}에서 잠재성 슬로트 1<i인 노드들을 찾아
      집합 SLOT에 넣는다 ;
    if (SLOT==NULL)
      break ;
    else {
      if (SLOT 에서 H(e_ij)가 가장 큰 노드 n_i가 하나이면)
        n_i 를 선택 ;
      else if (n_i가 여러개이면)
        C(n_i)가 가장 큰 노드 n_i를 선택 ;
      후보자 그룹과 SLOT 에서 n_i를 제거 ;
      RDG 에서 n_i를 마크 ;
    }
    후보자 노드이면서 Timing(n_j, n_k, L)=true 인 노드 n_j를
      후보자 그룹에 추가 ;
  }
  n_b를 선택하여 후보자 그룹에서 제거 ;
  RDG에서 n_b를 마크 ;
}
```

코드 재구성을 위한 스케쥴링은 기본 블럭을 범위로 하여 수행되기 때문에 재구성된 기본 블럭의 마지막 부분에 있는 명령어들은 인접된 다음 기본 블럭의 처음 부분의 명령어들과 후에 해저드가 발생할 가능성이 있다. 스케쥴링을 수행할 때 인접된 블럭들과의 관계를 고려하여 블럭간의 해저드 발생 가능성을 감소시킬 수 있으나 그 효율성이 높지 않고, 고려할 이웃 기본 블럭이 아직 재구성되지 않은 경우에는 많은 어려움이 따르게 된다. 따라서 본 논문에서는 블럭간에 발생할 수 있는 해저드 문제를 명령어의 잠재성 슬로트를 고려해 줌으로써 해결하고, 수행중인 기본 블럭의 재구성이 이웃 기본 블럭에 영향을 주지 않도록 한다.

Ⅳ. 코드 모션(Motion)

전장에서 제안된 코드 최적화를 위한 코드 스케쥴링은 분기 지연 처리를 위해 분기 명령어가 갖게 되는 시퀀싱 해저드를 고려하고 있다. 기본 블럭을 범위로 하여 처리된 분기 지연은 분기 명령어의 분기 타겟과 다음 명령어가 속한 기본 블럭을 고려한 코드 모션에 의해 좀 더 효율적인 코드를 생성할 수 있다.

(정의14) 분기 명령어의 수행 결과로 분기 타겟에 있는 명령어가 수행되게 된다면 이 분기 명령은 성취(taken) 된다고 하고, 분기되지 않고 바로 다음의 후속 명령어를 수행하게 될때 이 분기 명령어는 비성취(not taken) 된다고 한다.

(정의 15) 명령어 i를 분기 명령어라 할때 분기 타겟을 T 라 하고, 분기 명령어가 비성취되었을때 실행되는 후속 명령어의 위치를 N 이라 한다.

코드 모션에 의한 분기 지연 처리는 분기 명령어의 지연 슬로트에 이동된 명령어가 분기 명령어 이후 수행되어 가는 프로그램의 의미를 변경시키게 되는 영향을 줄 수 있기 때문에 기본 블럭의 시점(entry)에서 레지스터들의 사용되는 상태를 고려해 주어야 한다. 각 기본 블럭에 대해 블럭의 시점으로부터 시작되는 데이타 흐름에서 정의되기 전에 그 블럭이나 후행 블럭에서 먼저 참조되는 레지스터의 집합, 즉, 기본 블럭의 시점에서 live 한 레지스터 집합들을 고려해 준다.

(정의16) 한 분기 명령어에 대해 분기 명령이 성취되었을 때 수행하게 되는 기본 블럭의 시점에서 live 한 레지스터 집합을 IN(BT) 라 하고, 비성취 되었을 때 수행되는 기본 블럭의 시점에서 live 한 레지스터 집합을 IN(BN) 이라 한다.

〔알고리듬 6〕 코드 모션

```
Code Motion{
  for(각 기본 블럭에 대해) {
    if(기본 블럭내에 분기 명령어가 존재하면) {
      분기 명령어의 지연 슬로트 d를 결정 ;
      채워지지 않은 슬로트 s←d-(분기 명령어 이후 블럭의 끝까
          지의 분기 명령어 수)로 할당 ;
      if (s==0)
        continue ;
      switch(분기 명령어의 타입) {
        case 무조건 분기문 :
          T위치로부터 시작되는 s개의 명령어를 복사 ;
          분기 타겟을 T+s 위치로 수정 ;
          break ;
        case 조건 분기문 :
          if(분기 타겟==backward) {
            i←0 ;
            do {
              타겟 T+i 의 명령어를 선택 ;
              if(선택된 명령어의 데스티네이션
                레지스터∈IN(BN)) break ;
              선택된 명령어를 지연 슬로트에 복사 ;
              s--;   i++ ;
            } while(s>=1) ;
            분기 타겟을 T+i 위치로 수정 ;
            if(s>0) {
              s개 만큼의 nop 을 삽입 ;
              i←0 ;
              do{
                if(타겟 N+i 의 명령어의 데스티네이션
                  레지스터∈IN(BT)) break;
                삽입된 nop 중 하나를 제거 ;
                s--;     i++;
              } while(s>=1) ;
            }
          }
          else{
            i←0 ;
            do{
              if(타겟 N+i의 명령어의 데스티네이션
                레지스터∈IN(BT)){s개의 nop 을 삽입 ;
                break ;
              }
              s--;   i++;
            } while(s>=1) ;
            if(s>0) {
              i←0 ;
```

```
        do{
          타겟 T+i 의 명령어를 선택 ;
          if(타겟 N+i의 명령어의 데스티네이션
            레지스터∈IN(BN)) break ;
          선택된 명령어를(분기명령어의 위치+
              (d-s)+(i+1))의 위치에 복사 ;
          nop 하나를 제거 ;
          s--;  i++;
        } while(s>=1) ;
          분기 타겟을 T+i 위치로 수정 ;
        }
      }
    default :
      s 개의 nop을 삽입 ;
  }
 }
}
```

무조건 분기문과 조건 분기문으로 대별될 수 있는 분기 명령어 중에서 타겟이 되는 블럭의 정보를 사용하기가 어렵거나 불가능하기 때문에 코드 모션을 적용하지 못하는 명령어들이 있다. 트랩(trap) 이나 감독자 호출(supervisor call) 은 운영 체계에 이미 정의되어 있는 위치에서 그 수행을 재개하지만 이 분기 타겟에서 수행되는 명령어들에 대한 정보를 사용할 수 없기 때문에 고려 대상에서 제외된다. 서브루틴으로의 복귀나 간접 점프는 그 분기 타겟 값이 레지스터나 메모리에 저장되어 있어서 정적인 방법으로는 분기 타겟을 결정할 수 없기 때문에 코드 모션은 이 명령어들에 대해서 적용되지 않는다.

전장의 분기 최적화에 의해 수행된 지연 분기의 처리로 지연 슬로트에 채워진 명령어들은 분기가 성취되거나 비성취되거나에 상관없이 항상 수행되어야 하는 명령어이기 때문에 nop의 삽입만으로 지연 분기를 처리하는 방식에 비해 지연 분기 처리를 코드 모션에 의해 수행했을 때 얻을 수 있는 이점은 코드를 이동시켜 온 위치와 프로그램의 동적 특성에 따라 다르게 된다.

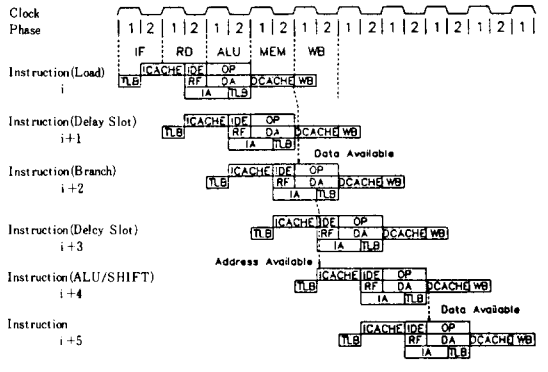
분기 타겟으로부터 명령어들을 이동시켜 온 경우는 분기가 성취되었을 때에만 수행속도가 향상되고 코드 크기는 코드를 복사하기 때문에 줄어 들지 않는다. 이러한 코드 모션은 분기 타겟이 후진(back-ward) 방향일 때 적용하게 되는데, 후진 방향의 분기는 거의 프로그램의 루프(loop) 구조를 제어하기 위한 분기가 되고 루프 분기 명령어는 거의 항상 성취되기 때문이다.

그러나 전진(forward) 방향의 분기는 거의 같은 확률로 성취되거나 비성취된다.[9] 따라서 이 경우에는 비성취 방향에서 코드를 이동시켜 옴으로써 코드 크기를 줄이고, 분기가 성취되지 않을 경우에는 수행 속도까지도 감소시킬 수 있다.

V. 검 토

본 논문에서 제안한 코드 최적화 과정을 MIPS Computer System에서 개발한 하드웨어 인터록이 없는 RISC 프로세서인 R2000에서 수행될 수 있는 프로그램 코드에 적용하여 그 결과를 검토한다.

R2000 프로세서는 RISC 아키텍춰로 설계되어 있어 모든 명령어들을 단일 사이클로 수행하며, 5개의 명령어를 중복되게 하여 동시에 처리할 수 있는 5-스테이지 파이프라인으로 설계되어 있다. 그림 4 는 파이프라인의 각 스테이지의 기능과 명령어 수행 과정을 나타낸다.



TLB	Translation Lookaside Buffer
ICACHE	Instruction Cache Access
IDEC	Instuction Decode
RF	Register Operand Fetch
IA	Instruction Address Calculation and Translation
OP	Operation(ALU/Shift)
DA	Data Address Calculation and Translation
DCACHE	Data Cache Access
WB	Write-back to register

그림 4. 명령어 파이프라인
Fig. 4. Instruction pipeline.

그림 4 에서 i번째 명령어를 load 명령어라 하자. Load 명령어에 의해 적재되는 데이타는 명령어 i의 MEM 사이클이 끝날때까지 유효하지 않기 때문에 명령어 i+1의 ALU 사이클 동안에는 사용될 수 없게 되고, 명령어 i+2의 ALU 사이클에서 사용될 수 있다. 따라서 load 명령어에 의해 레지스터에 적재되는

데이타는 한 사이클의 지연후에 사용될 수 있으므로 load 명령어는 타이밍 해저드에 의한 슬로트 1을 갖는다.

명령어 i+2를 분기 명령어라 하자. 명령어 i+2는 분기 타겟 어드레스를 계산해야 하고 이 어드레스는 명령어 i+2의 ALU 사이클의 첫 위상(phase)전에는 사용될 수 없기 때문에 명령어 i+3의 명령어 캐쉬 엑세스를 하기에는 너무 늦게 되고, 따라서 명령어 i+4의 명령어 캐쉬 액세스를 위해 사용하게 된다. 지연 슬로트 (i+3)내의 명령어는 분기나 점프가 발생하기 전에 항상 수행되기 때문에 시퀀싱 해저드가 존재하게 된다.

또한 메모리 시스템의 구조적 특성에 의해 발생되는 구조적 해저드가 존재한다. 외부 캐쉬는 write-back 캐쉬로서 store 동작을 위해 2개의 메모리를 사이클을 필요로 한다. 한 사이클은 캐쉬내의 위치를 검사하기 위한 것이고, 다른 한 사이클은 데이타를 저장하기 위해 사용한다. 따라서 2개의 메모리 사이클을 아무런 장애없이 사용할 수 있도록 하기 위해 store 명령어의 바로 다음에 메모리 비참조 명령어가 오도록 한다. 예를 들어서 계산 명령어는 MEM 사이클을 사용하지 않기 때문에 계산 명령어가 store 명령어의 바로 다음에서 수행될 수 있도록 해주는 것이 좋다.

코드 최적화의 수행 과정을 설명하기 위해 한 예로 그림5와 같은 한 기본 블럭의 명령어 시퀀스에 대해 먼저 코드 스케줄링을 수행해 보자.

이 명령어 시퀀스에 대해 알고리듬2를 적용하여 자원 종속 그래프를 구성하면 그림6과 같이 된다. 여기서 레지스터 r0는 하드-와이어드 0으로 되어 있어 이 레지스터에 어떠한 값이 기입되는지에 상관없이 항상 0값을 갖는다. 그리고 각 에지에는 두 노드간의 해저드에 의한 지연 슬로트에 1을 더한 값이 가중치로 주어지며, 상충 자원의 사용에 의한 프로그램의 의미 변경을 막기 위해 생성된 에지에는 0을 할당한다. 자원 종속 그래프가 구성되고 에지 가중치가 할당된 후에, 알고리듬3에 의해서 각 노드에 대한 임계 가중치를 계산한다. 각 노드의 임계 가중치는 리프 노드로부터 그 노드에 이르는 패스중, 임계 경로(critical path)에 해당하는 에지들의 가중치의 합으로 주어진다. 리프 노드의 임계 가중치는 항상 0으로 주어진다.

모든 노드에 임계 가중치가 주어진 자원 종속 그래프에 대해 알고리듬4와 알고리듬5를 적용하여 코드를 재구성한다. 최초의 후보자 그룹에 속하는 노드들은 자원 종속 그래프의 루트 노드들이 되므로 후

simple instruction stream

1	lw	r27, −4(fp)
2	lw	r26, −8(fp)
3	add	r1, r27, r26
4	sw	r1, _x(r26)
5	lw	r24, _a(r0)
6	sub	r2, r27, r24
7	sw	r2, −4(fp)
8	lw	r23, 0(ap)
9	lw	r1, _arr(r23)
10	addi	r22, r22, #1
11	slt	r1, r22, r1
12	beq	r1, r0, L27

그림5. 코드 시퀀스의 한 예
Fig. 5. An example of code sequence.

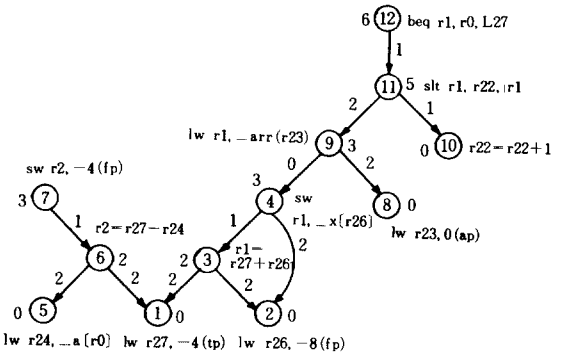


그림6. 그림5의 코드 시퀀스에 대응하는 자원 종속 그래프
Fig. 6. Resource depependent graph corresponding the code sequence of Fig. 5.

보자 그룹은 {7, 12}가 되고, 노드 12가 분기 명령어이므로 알고리듬5에 의한 분기 지연 최적화를 수행한다. 분기 명령어인 노드 12의 지연 슬로트는 1이므로 먼저 노드 7이 선택된 후에 노드 12가 선택되고, 선택된 노드들은 자원 종속 그래프에서 마크된다. 분기 지연 최적화가 완료된 후에 알고리듬4로 복귀되어 자원 종속 그래프의 모든 노드가 마크될 때까지 타이밍 해저드와 구조적 해저드를 처리하기 위한 코드 재구성을 계속해 나간다. 이제 후보자 그

룹은 {6,11}이 되고, 노드 11을 시작 노드로 한 에지의 가중치 $H(e_{11,9})=2$가 노드 6을 시작 노드로 한 에지의 가중치 $H(e_{6,5})=H(e_{6,1})=2$와 같기 때문에 임계 가중치가 큰 노드 11을 먼저 선택하여 마크한다. 새로운 후보자 그룹은 {6,10}이 되고, 선택 기준에 따라 노드 6이 선택되어 마크되고, 후보자 그룹은 {9,10}으로 계산된다. 노드 1이나 5는 마크된 노드 6과 지연 슬로트 1이상 떨어져 스케쥴링되어야 하기 때문에 후보자 그룹에 추가될 수 없다. 노드 9가 마크된 후 후보자 그룹은 {4,5,8,10}이 되어 이 중에서 에지의 가중치가 가장 큰 노드 4가 선택되나 구조적 해저드를 고려하기 위한 함수 Structural(4,9,1)=false이기 때문에 그 대신에 노드 10이 선택되어 마크된다. 이러한 방법으로 모든 노드들이 마크될때까지 수행하면 마크된 노드의 순서 리스트는 7,12,11,6,9,10,4,3,8,5,2,1이 된다. 실제 수행하게 되는 재구성된 코드 시퀀스는 구해진 순서 리스트의 역순으로서, 1,2,5,8,3,4,10,9,6,11,12,7이 된다. 그림 5의 코드 시퀀스에 대한 코드 최적화의 결과는 그림7(b)와 같이 된다. 그림 7(a)는 해저드가 존재할 때마다 nop을 삽입하여 처리한 코드 시퀀스의 결과이다. 이 코드가 수행될 때에는 9번째 load명령어의 수행 후에 load 명령어를 수행하기 전에 1사이클 만큼의 프로세서의 일시 중단이 발생하게 된다.

그림 8은 nop의 삽입만으로 해저드를 해결하여 생성된 여러 개의 기본 블럭들로 구성되어 있는 코드 시퀀스이다. 그림9(a)는 그림 8에 대한 원시 코드 시퀀스의 각 기본 블럭에 대해 코드 최적화를 수행한 결과이다. 모든 타이밍 해저드가 해결되어 있고 시퀀싱 해저드는 유일하게 11번행의 분기 명령에 대해 지연 슬로트에 채워질 명령어가 없어 해결되지 못했다. 그림 9(b)는 알고리듬 6의 코드 모션에 의해 좀 더 효율적인 분기 최적화를 수행하여 얻은 최종 결과이다. 11번행의 분기 명령어의 타겟은 후진 방향이므로 라벨 L15이후의 기본 블럭의 흐름에서 레지스터 r25가 live하지 않다고 가정하면, 라벨 L3의 기본 블럭에서 시작되는 명령어를 지연 슬로트에 복사할 수 있게 되고, 분기 타겟은 L3+1 위치로 수정된다.

다음과 같은 6개의 테스트 프로그램을 사용하여 본 논문에서 제안한 알고리듬과 Gross의 알고리듬 및 default 방법을 적용한 결과를 비교 검토한다.

- Ackerman : ackerman의 함수를 계산하는 재귀적(recursive) 프로그램
- Hanoitower : "hanoi의 탑" 문제를 푸는 프로그램

```
1  lw    r27, -4(fp)
2  lw    r26, -8(fp)
3  nop
4  add   r1, r27, r26
5  sw    r1, _x(r26)
   **********
6  lw    r24, _a(r0)
7  nop
8  sub   r2, r27, r24
9  sw    r2, -4(fp)
   **********
10 lw    r23, 0(ap)
11 nop
12 lw    r1, _arr(r23)
13 addi  r22, r22, #1
14 slt   r1, r22, r1
15 beq   r1, r0, L27
16 nop
```

(a)

```
1  lw    r27 -4(fp)
2  lw    r26, -8(fp)
3  lw    r24, _a(r0)
4  lw    r23, 0(ap)
5  add   r1, r27, r26
6  sw    r1, _x(r26)
7  add   r22, r22, #1
8  lw    r1, _arr(r23)
9  sub   r2, r27, r24
10 slt   r1, r22, r1
11 beq   r1, r0, L27
12 sw    r2, -4(fp)
```

(b)

그림 7. (a) nop의 삽입으로 해저드를 처리한 코드 시퀀스
(b) 코드, 최적화 수행 결과

Fig. 7. (a) Hazard-free code sequence by default solution.
(b) The execution result of code optimization.

```
L3:
   addi  r25, r0, #20
      ⋮
1  addi  sp, sp, #-52
2  addi  r1, r0, #10
3  lw    r27, -8(fp)
4  slt   r1, r1, r27
5  bne   r1, r0, L15
   nop
L13:
6  lw    r26, -4(fp)
7  add   r24, r0, r26
8  j     L30
   nop
9  add   r1, r23, r25
10 slt   r1, r1, r25
11 beq   r1, r0, L3
   nop
L15:
      ⋮
```

그림 8. 원시 코드 시퀀스

Fig. 8. Original code sequence.

```
L3 :                          addi r25, r0, #20
  addi r25, r0, #20           L3 :
    ⋮                           ⋮
3  lw    r27, -8(fp)        3  lw    r27, -8(fp)
1  addi  sp, sp, #-52       1  addi  sp, sp, #-52
4  slt   r1, r1, r27        4  slt   r1, r1, r27
5  bne   sp, sp, #-52       5  bne   sp, sp, #-52
2  addi  addi r1, r0, #10   2  addi  r1, r0, #10
 L13 :                       L13 :
6  lw    r26, -4(fp)        6  lw    r26, -4(fp)
8  j     L30                8  j     L30
7  add   r24, r0, r26       7  add   r24, r0, r26
9  add   r1, r23, r25       9  add   r1, r23, r25
10 slt   r1, r1, r22       10 slt   r1, r1, r22
11 beq   r1, r0, L3        11 beq   r1, r0, L3
L : 15                        addi  r25, r0, #20
  ⋮                           L15 :
                                ⋮
        (a)                          (b)
```

그림 9. (a) 코드 최적화의 중간 수행 결과
(b) 코드 모션후의 최종 결과

Fig. 9. (a) Intermidiate execution result of code optimization.
(b) Final result after code motion.

- Bubblesort : bubble sort 프로그램
- Shakersort : shaker sort 프로그램
- Mergesort : merge sort 프로그램
- Quicksort : quick sort 프로그램

표 1은 테스트 프로그램으로 사용한 여러 개의 프로그램 코드에 대해 코드 최적화를 수행하여 얻은 결과와 다른 방법들에 의해 얻은 결과와의 비교이다. 해저드가 해결되지 않은 원시 프로그램 코드에 대해 각 방법들을 적용하여 얻은 코드 크기의 증가율과 주어진 코드 시퀀스의 순서를 그대로 유지하면서 해저드가 존재할 때마다 nop의 삽입으로써 해결해 주는 default 방법보다 코드 크기에 있어서 향상된 백분율을 나타낸다.

표 2는 구조적 해저드를 고려하여 코드 최적화를 수행한 결과로서 프로그램을 수행했을 때 정적인 수행 사이클 수를 줄임으로서 얻을 수 있는 동작 속도의 향상을 보여준다. 원시 프로그램에 대해 다른 해저드 처리의 결과에 따른 영향을 배제하고, 구조적 해저드의 영향에 의해서만 증가되는 정적인 프로그램 수행 사이클 수를 비교한 것이다. 명령어들이 반복 수행되는 루프를 고려해 볼 때, 실제 수행되는 프로그램의 동적 사이클 수의 향상은 다른 방법들에 비해 본 논문에서 제안된 방법을 사용함으로서 더욱 커질 수 있을 것이다.

표 1. 종래의 방법과의 코드 크기의 비교
Table 1. Comparison with conventional method for code size.

프로그램명	명령어수	원시 프로그램 코드에 대한 코드 크기의 증가율(%)/Default 방법에 대한 향상율(%)		
		Default 방법	Gross 방법	제안한 방법
Ackerman	97	9.3/0.0	4.0/60.0	2.0/80.0
HanoiTower	80	13.0/0.0	4.8/66.7	1.2/91.7
Bubblesort	26	21.2/0.0	7.1/71.4	3.7/85.7
Shakersort	44	20.0/0.0	6.4/72.7	6.4/72.7
Mergesort	105	20.5/0.0	1.9/92.6	1.9/92.6
Quicksort	97	19.2/0.0	10.2/52.2	9.3/56.5

표 2. 정적인 수행 사이클 수의 비교
Table 2. Comparison of static execution cycle time.

프로그램명	원시프로그램에 존재하는 구조적 해저드	원시프로그램에 대한 정적 수행사이클 수의 증가율(%)/Default 방법에 대한 수행사이클수의 감소율(%)		
		Default 방법	Gross 방법	제안한 방법
Ackerman	15	14.3/0.0	14.3/0.0	3.1/80.0
HanoiTower	11	13.8/0.0	18.8/-36.4	8.8/36.4
Bubblesort	1	3.9/0.0	3.9/0.0	0.0/100.0
Shakersort	2	2.1/0.0	0.0/100.0	0.0/100.0
Mergesort	0	0.0/0.0	0.0/ --	0.0/—
Quicksort	11	11.3/0.0	11.3/0.0	2.3/90.9

표 3은 각 방법에 의해 수행된 분기 명령어에 대한 지연 분기 처리의 결과이다. 분기 지연 처리는 코드 크기와 수행 속도에 이바지하는 특성에 따라 4가지 형태로 구분된다. 형태 1은 코드 크기와 수행 속도를 동시에 향상시키는 경우이고, 형태 2는 코드 크기는 감소되지만 수행 속도는 분기가 성취 혹은 비성취 됨에 따라 어느 한 방향에서만 향상되는 경우이고, 형태 3은 지연 슬로트에 명령어가 복사되어 코드 크기가 감소되지 않고 분기의 어느 한 방향에서만 수행 속도가 향상되는 경우이며, 마지막으로 형태 4는 지연 슬로트를 유효한 명령어로 채울 수 없기 때문에 nop을 삽입해 주는 경우이다.

V. 결 론

본 논문에서는 인터록이 없는 RISC 파이프라인 아키텍춰에서 발생하는 타이밍 해저드와 시퀀싱 해

표 3. 분기 명령어의 지연 분기 처리의 비교
Table 3. Comparison of delayed branch optimization of branch instruction.

프로그램명	분기 명령	분기 지연 처리의 형태에 따른 백분율(%)											
		형 태 1			형 태 2			형 태 3			형 태 4		
		제안한 방 법	Gross 방 법	Patterson 방 법	제안한 방 법	Gross 방 법	Patterson 방 법	제안한 방 법	Gross 방 법	Patterson 방 법	제안한 방 법	Gross 방 법	Patterson 방 법
Ackerman	10	80.0	50.0	40.0	0.0	0.0	0.0	0.0	0.0	0.0	20.0	50.0	60.0
HanoiTower	7	85.7	42.9	42.9	0.0	0.0	0.0	14.3	14.2	14.2	0.0	42.9	42.9
Bubblesort	5	40.0	20.0	20.0	40.0	40.0	40.0	20.0	20.0	20.0	0.0	20.0	20.0
Shakersort	7	28.6	28.6	28.6	42.8	42.8	42.8	14.3	14.3	14.3	14.3	14.3	14.3
Mergesort	20	50.0	40.0	30.0	25.0	55.0	60.0	0.0	0.0	5.0	5.0	5.0	5.0
Quicksort	19	47.3	42.1	42.1	31.6	31.6	31.6	15.8	15.8	15.8	5.3	10.5	10.5

저드를 동시에 고려하여 해결하는 파이프라인 스케줄링에 의한 코드 최적화 알고리듬을 제안하였다. 동시에 수행되는 명령어들간의 자원 상충에 기인하는 타이밍 해저드와 분기 타겟 결정을 위한 지연 시간 때문에 발생하는 시퀀싱 해저드를 동시에 고려하여, 코드 최적화의 수행으로 생성된 재배열된 코드 시퀀스의 크기를 감소시킬 수 있도록 하였다. 기본 블럭에 속한 명령어 시퀀스를 읽어 자원 종속 관계를 나타내는 방향성 비순환 그래프 RDG(resource dependent graph)를 작성하고, 노드와 에지에 각각 인터록 가중치 I와 누적 가중치 C를 주고 이 가중치에 의해 생성될 코드 시퀀스를 역방향으로 선택하여 구한다. 또한 코드 최적화를 수행할 때, 동적 특성을 갖는 구조적 해저드를 함께 고려하여 프로그램의 동적 수행 속도를 향상시켰다.

본 논문에서 제안한 코드 최적화 과정은 C 언어로 구성하였고, MIPS의 R2000 프로세서에서 수행되는 코드에 적용시켜 효율성을 보였다.

參 考 文 献

[1] T. Manuel, "The Frantic Search for More Speed," Electronics, pp.59-62, Sep. 1987.

[2] J.L. Hennessy, "VLSI RISC Processors," VLSI System Design, pp. 22-32, Oct. 1985.

[3] J. Hennessy, N. Jouppy, F. Baskett, T. Gross, J. Gill, "Hardware/Software Tradeoffs for Increased Performance," Proc. of ACM Symp. on Architectural Support for Programming Lang. and Operating Systems, pp. 2-11, Mar. 1982.

[4] B. Randell, "Hardware/Software Tradeoffs: A General Design Principle," ACM Computer Architecture, pp. 19-21, June 1985.

[5] J. Walacki, J.D. Laughlin, "Operation Scheduling in Reconfigurable, Multifunction Pipelines," MICRO 20. pp. 80-87, Dec. 1987.

[6] H.S. Stone, "High-Performance Computer Architecture," Addison Wesley, 1987.

[7] J.K.F. Lee, A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," Computer vol.17, no.1, pp.6-22, Jan. 1984.

[8] J. Hennessy, T. Gross, "Postpass Code Optimization of Pipeline Constraints," ACM Trans. on Programming Lang. and Systems, vol. 5, no. 3, pp. 422-448, July 1983.

[9] L.J. Shustek, "Analysis and Performance of Computer Instruction Sets," Ph.D Dissertation, Stanford Univ., May 1977.

[10] A.G. Liles, B.E. Willner, "Branch Prediction Mechanism," IBM Technical Disclousre Bull., vol. 22, no. 7, pp. 3013-3016, 1979.

[11] G.S. Rao, "Technique for Minimizing Branch Delay Due to Incorrect Branch History Table Predictions," IBM Technical Disclosure Bull., vol. 25, no. 1, pp. 97-98, June 1982.

[12] J.J. Losq, "Generalized History Table for Branch Prediction," IBM Technical Disclosure Bull., vol. 25, no. 1, pp. 99-101, June 1982.

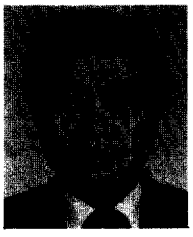
[13] E. Smith, "A Study of Branch Prediction Strategies," Proc. Eighth Symp. Computer Architecture, pp. 135-148, May 1981.

[14] D. A. Pattren, C. H. Sequin, "A VLSI RISC," *IEEE Computer*, vol. 15, no. 9, pp. 8-21, Sep. 1982.

[15] G. Radin, "The 801 Minicomputer," IBM J. Res. Development, vol. 27, no. 3, pp.

237-246, May 1983.
[16] T.R. Gross, J.L. Hennessy, "Optimizing Delayed Branch," *Proc. Micro-15, IEEE*, pp. 114-120, Oct. 1982.
[17] 임동규, 박종득, 김은성, 임인칠, "파이프라인 아키텍춰의 효율적인 분기 지연 처리," 한국정보과학회 '88 봄 종합학술발표논문집, 제15권 1호, pp. 379-382, 1988. 4.
[18] 박기호, 박종득, 김은성, 임인칠, "RISC의 파이프라인 인터록을 위한 코드 스케쥴링 알고리듬," 한국정보과학회 '88 봄 종합학술발표논문집, 제15권 1호, pp. 383-386, 1988. 4.

著 者 紹 介



林 寅 七(正會員)
1962年 한양대학교 공과대학 졸업. 1970年 와세다대학 전자공학과 박사학위 취득. 일본 후지쯔(주) 정보처리시스템연구소 연구원, 한국과학기술원 대우교수, University of Illinois at Urbana-Champaign의 Computer Science 학과 Visiting Professor 역임. 1964年 한양대학교 강사, 조교수, 부교수. 1977年~현재 한양대학교 교수 재직, 현재 동대학교 전자계산소 소장. IEEE Computer Society Korea Chapter Chairman. 주관심분야는 Computer Architecture, RISC Processor 및 Compiler 설계, VLSI Testing / Testable Design, Simulation, Layout, AI 기법을 이용한 VLSI 설계 및 Machine-Translation 등임.

金 殷 成(正會員)
1957年 8月 12日生. 1980年 2月 한양대학교 전자공학과 졸업. 1985年 2月 한양대학교 대학원 전자공학과 졸업 공학석사 학위취득. 1985年 3月~현재 한양대학교 대학원 전자공학과 박사과정 재학중 주관심분야는 RISC 및 Parallel Computer Architecture, Algorithm, Optimizing Compiler, VLSI Circuit Design등임.