論文 88-25-7-4

파이프라인 아키텍춰를 위한 코드 스케쥴링 알고리듬

(A Code Scheduling Algorithm for Pipelined Architecture)

金 殷 成*, 林 寅 七*

(Eun Sung Kim and In Chil Lim)

要 約

본 논문에서는 파이프라인 인터록을 소프트웨어로 해결하는 코드 스케쥴링 알고리듬을 제안한다. 즉 파이프라인 아키텍춰에서 머신 명령어를 수행할 때 파이프라인 인터록이 발생될 경우, 복잡한 하드웨어 인터록 메카니즘을 사용하는 대신 컴파일 시에 소프트웨어로써 명령어들을 재구성해 줌으로서 이를 처리하도록 한다.

상충 자원의 사용 순서를 엄격히 지켜야 제약을 완화시켜 명령어들을 좀 더 자유롭게 선택할 수 있도록 하여 프로그램 코드 크기를 줄이고 전체 수행 시간을 단축할 수 있도록 한다.

Abstract

This paper proposes a code scheduling algorithm which gives a software solution to the pipeline interlock. This algorithm provides a heuristic solution by recordering the instructions, instead of using hardware interlock mechanism when pipeline interlock prevents the execution of a machine instruction in a pipelined architecture.

Program code size and overall execution time can be reduced due to the increased flexibility in the selection of instructions, which is possible from the alleviated ordering restriction on the use of conflict resources.

I. 서 론

오늘날 컴퓨터 아키텍춰는 회로의 고집적화에 따른 하드웨어의 발전과 이를 지원하기 위한 소프트웨어 기술의 급속한 발전에 힘입어, HLL(고급언어, high level language)을 뒷받침하기 위한 호스트로서의 효율성과 아키텍춰를 실현함으로써 얻을 수 있는 수행 성능을 향상시키기 위해 꾸준히 발전되어 왔다.[1,2] 최근 컴퓨터 아키텍춰 설계로 좀 더 간소하고 수행 빈도가 큰 명령어들의 효율적인 이용으로 하드웨어 자원 이용의 효율을 극대화시키고 고성능의 수행 능력을 갖추어 최선의 비용/성능을 얻기 위한 RISC(reduced instruction set computer) 기법을 널리 사용하고 있다.[3,4,5]

RISC 개념을 근간으로 하는 컴퓨터는 하드웨어의 단순화를 지향하여 대부분의 명령어들을 단일 사이클내에 실행 가능토록 하고, 이를 위해 명령어의 크기를 동일하게 하고 명령어의 형식을 고정시켜 명령어 디코드 시간을 줄이고, 대부분의 명령어에 공통적으로 사용되는 데이타 패스외에 특별한 하드웨어 지원이 필요한 명령어와 사용 빈도수가 적은 명령어

*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 4月 18日

를 배제하여 개개의 명령어를 더 빠르게 실행시킨다. 따라서 RISC는 기존의 방식보다 많은 스테이지를 갖는 강력한 파이프라인의 실현을 가능케 하여 컴퓨터의 성능을 대폭 향상시킬 수 있게 된다.

파이프라인 아키텍춰는 몇개의 순차적인 명령어들을 보통 서로 다른 스테이지를 사용함으로써 동시에 수행할 수 있게 한다. 어떤 명령어가 자기의 오퍼랜드로서 자기보다 선행하는 명령어에서 계산된 값을 사용하게 되는 경우가 있는데, 만약 선행하는 명령어가 여전히 수행 중에 있다면 그 결과 값을 이용할 수 없게 되고, 이 두 명령어 간에는 데이타 의존 관계에 따른 파이프라인 타이밍 해저드(pipeline timing hazard)가 존재하게 된다. 하드웨어 실현에 의해 이러한 해저드를 검출하고, 프로그램을 수행한 실제 결과가 프로그래머가 의도했던 것과 일치될 수 있도록 파이프라인 인터록(interlock)을 마련하여 필요한 값이 사용될 수 있을 때까지 후행 명령어의 실행을 지연시켜 타이밍 해저드를 해결해 준다.

파이프라인 아키텍춰에서 인터록 메카니즘의 설계는 복잡할 뿐더러 고성능의 프로세서에 대한 과도한 하드웨어 오버헤드를 초래하게 된다. 또한 해저드 검출을 위한 검사는 명령어에 대한 해저드의 존재 여부에 상관없이 모든 명령어에 오버헤드를 부과하게 된다. 그리고 해저드가 존재하는 경우에는 그 실행이 지연되기 때문에 파이프라인을 효율적으로 이용하지 못하게 되어 프로그램의 수행 속도를 저하시키게 된다.[6, 7, 8]

이러한 복잡한 하드웨어를 사용하는 방법을 대신하여 인터록 메카니즘을 제거시키고, 컴파일 시에 파이프라인 스케쥴링에 의해 명령어들의 인터록 관계를 고려하여 원래의 코드 시퀀스를 재구성해 줌으로써 해저드를 제거하는 방법이 제안되었다.[9, 10, 11, 12]

하드웨어의 단순화와 규칙화는 단일 VLSI 칩으로 프로세서를 실현하기 위한 중요한 요소가 되고, 하드웨어 오버헤드를 감소시킴으로써 동작 속도를 높이는 한편, 파이프라인의 패러랠리즘을 효율적으로 이용할 수 있다. 또한 올바르게 재구성된 코드는 하드웨어 인터록을 갖는 파이프라인에서 수행되는 코드보다 더 좋은 성능을 갖는다. 그러나, 코드 시퀀스에 nop(No-Operation)을 추가 삽입해야 하기 때문에 코드 크기가 커지게 되고 컴파일 시에 코드 스케쥴링을 수행해야 하기 때문에 컴파일러에 부담이 많아지는 단점이 있다.

Gross[11]는 nop의 사용을 줄이기 위해 레지스터 이용 순서에 대한 제약을 완화시킴으로서 더 좋은 효율을 기대하고 있으나, 메모리 참조 명령어의 우선 순위 검사등의 상충 레지스터 이용의 제약 완화를 위한 조건 계산이 복잡하여 기본 블럭의 명령어가 N개일 때 $O(N^4)$의 많은 시간이 걸리게 되며, 서로 상충되어 있는 노드쌍 중에서의 선택 과정이 비효율적이다.

본 논문에서는 코드 선택을 위한 우선 순위와 자원 상충 노드에 대한 가중치를 고려하여 명령어 코드의 전체 수행시간 및 기억 용량을 감소시킬 수 있는 코드 스케쥴링 방식을 제안한다. 선택될 가능성을 갖는 노드들에 대하여 적법한 코드 스케쥴링을 위한 제약 조건의 계산을 최소화시키고, 스케쥴링될 자격을 갖는 자원 상충 노드들에 가중치를 주고 그 순서에 의해 스케쥴링함으로써 차후의 노드 선택 범위를 확장해 주게 되어 효율적으로 코드를 생성할 수 있게 한다. 제안된 알고리듬 수행의 복잡도는 N개의 명령어로 이루어진 기본 블럭에 대해 $O(N^2)$의 차수를 갖는다.

Ⅱ. 기본정의 및 관련문제

(정의1) i를 명령어라 하고 ps를 파이프스테이지라 하자. 명령어 i가 스테이지 ps동안에 자원 r의 값을 읽는다면, 자원 r은 위상(i, ps)에서의 소스 필드(source field)라 한다. 만약 명령어 i의 ps 스테이지 동안에 r이 기입된다면, 자원 r은 위상(i, ps)에서의 데스티네이션 필드(destination field)라 한다.

(정의2) 명령어 i_2는 명령어 i_1보다 후에 그 수행이 시작되고 (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_2는 데스티네이션-소스 파이프라인 상충된다고 한다.

파이프라인 프로세서에서 명령어들의 수행이 중첩되어, 명령어 i의 결과를 m 사이클 후에야 사용할 수 있다고 하자. $n<m$이라 하면 명령어 i에 의해 기입되는 데이타를 명령어 $i+n$이 참조하려고 할 때 타이밍 해저드가 발생한다. 이러한 데이타 의존 관계가 적법한 모든 스케쥴링에 대한 고려 대상이다. 명령어 i와 $i+n$을 충분히 분리시키지 않으면 명령어들을 수행할 때 타이밍 해저드가 발생되고 주어진 원래의 프로그램의 올바른 수행 결과를 보장할 수 없게 된다. 타이밍 해저드를 좀 더 정확하게 데스티네이션-소스 파이프라인 상충이라 말할 수 있다. 대부분의 파이프라인 인터록은 레지스터 내용의 억세스(access)만을 고려하고 있다. 캐쉬(cache)나 주기억장치 등의 메모리에 대한 인터록은 runtime시에 동적으로 결정되기 때문에 검출할 수가 없다. 따라서 대부분의 아키텍춰는 최소한 store 오퍼레이션에 대해 메모리의 엄격한 순차적 억세스를 유지하도록 하여 메모리에

대한 인터록의 요구를 최소화시키고 있다. 본 논문에서는 레지스터에 대한 인터록만을 고려한다. 이러한 고려는 실제 적용에 아무런 제약도 주지 않는다.

코드 스케쥴링을 수행하기 위해서는 인터록 관계가 고려되지 않은채 생성된 머신 레벨 코드에 대한 표현과 기계 명령어 간의 인터록에 관한 정보가 요구된다. 컴파일러에 의해 생성된 머신 레벨 코드를 나타내기 위해 방향성 비순환 그래프(DAG)를 사용한다. 하나의 노드는 하나의 명령어를 나타내며 각 노드에 대한 라벨은 오퍼레이션의 의미를 명시해 주고, 이 오퍼레이션이 데스티네이션 레지스터를 사용한다면 이 레지스터도 명시해 준다. 리프 노드(leaf node)에 대한 라벨은 기본 블럭의 입구에서의 리프 노드의 값을 명시한다. 즉, load와 같은 오퍼레이션을 명시해 주고 레지스터나 기억 장소 위치를 나타낸다. 내부 노드의 오퍼랜드는 그 자식이 되는 노드의 오퍼레이션의 데스티네이션에 의해 주어진다. DAG 데이타 구조는 자원 상충 관계를 갖는 명령어와 기본 블럭의 입구 및 출구에서 live해야 할 레지스터를 갖는 명령어에 관한 정보 및 배열 원소(array element)나 포인터로 억세스(access)되는 대상과 같이 alias된 대상의 억세스의 순서를 유지하기 위한 정보와 링크되어 진다.

DAG에 대한 평가 순서는 다음을 만족할 때 적법하게 된다.

(1) 부모는 그의 모든 자식들이 스케쥴링된 후에야 스케쥴링될 자격을 갖는다.

(2) 레지스터나 기억 장소의 값을 사용하는 모든 노드는 그 값을 변경시킨 새로운 값을 사용하는 노드보다 먼저 그 스케쥴링이 완결된다.

(3) 메모리에 대한 load나 store는 이들이 동일한 어드레스를 참조할 경우, 원래의 순서를 유지한다.

(4) 레지스터의 값이 원래의 기본 블럭의 출구에서 live하다면, 그 값이 기입된 레지스터를 갖는 노드는 이 레지스터에 대해 다른 값을 기입하려는 노드보다 가장 나중에 선택된다.

(5) 노드 i가 노드 j의 부모가 되는 모든 노드 i, j는 이들 사이에 정의된 인터록 거리 이상만큼 분리된다.

최적의 코드 스케쥴링 문제는 적법한 DAG에 대한 평가 순서를 만족시키면서 최소한의 nop를 사용하여 원래의 결과와 같은 결과를 얻는 명령어 시퀀스를 생성하는 것이다.

(정의3) 2개의 명령어가 모두 동일한 데스티네이션 필드 r을 갖는다는 이 명령어는 자원 r에 대하여 서로 상충된다고 하고, 이 데스티네이션 필드를 상충 자원이라 한다.

(정의4) 2개의 명령어 i, j가 서로 상충되고 j가 자기의 부모로 명령어 j' 를 갖고 있을 때, 명령어 j' 를 스케쥴링할 시점에서 명령어 j보다 명령어 i가 후에 선택되어 있었다면 데드록(deadlock)이 발생한다고 한다.

데드록 발생의 가능성 때문에 다음 명령어를 선택하려고 할 때 스케쥴링 준비가 된 명령어만을 보고 선택하는 것은 적절치 않다. 선택될 노드를 결정하기 위해서는 선택된 후에 데드록 상황에 빠지지 않도록 미리 고려해 주어야 한다. 이러한 문제를 해결하기 위해 다음과 같은 개념을 사용한다. 그림 1은 데드록 발생을 설명하기 위한 예이다. 이 그림에서 명령어 1 다음에 명령어 3을 선택하는 스케쥴링은 데드록 상황을 만나게 될 것이다. 본 논문에서는 알고리듬의 효율성을 고려하여 backtracking의 사용을 허용하지 않고, 이러한 상황이 발생하지 않도록 스케쥴링을 수행해 간다.

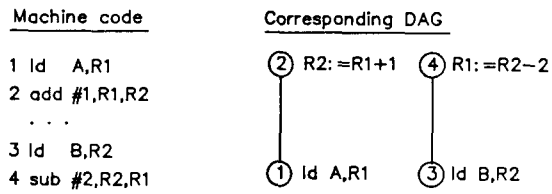

그림 1. 데드록 발생이 가능한 DAG
Fig. 1. DAG with potential deadlock.

(정의5) DAG D에 대해 코드를 생성할 때, 생성된 코드에 대응하는 노드는 커버(cover) 되었다고 한다.

(정의6) 어떤 DAG D에서 한 노드 n이 커버되어 있고, 노드 n의 데스티네이션 레지스터 dest(n)=r을 소스 레지스터로 생성하는 모든 노드가 커버된다면 즉, 노드 n을 자식으로 하는 모든 노드들이 커버된다면 노드 n은 완전히 커버된다고 한다.

(정의7) 데스티네이션 레지스터 r을 갖는 노드 n_1, n_2가 서로 상충되고, 노드 n_1이 커버되어 있다면 노드 n_1이 완전히 커버될 때까지 노드 n_1은 자원 r에 대해 노드 n_2와 배타적 관계에 있다고 한다.

(정의8) DAG D에서 데스티네이션 자원 r을 갖는 노드 n에 대한 배타적 그룹 EXG(n, r)을 다음과 같이 정의한다.

i) dest(n) = r 이 상충 자원이면

EXG(n, r) = {n} ∪ {i | n의 부모가 되는 노드i} ∪ {j | 노드 i의 커버되지 않은 모든 자손 j}

ii) dest(n) = r 이 상충 자원이 아니면

EXG(n, r) = {n}

(정의9) 소스 레지스터를 자신의 데스티네이션으로 다시 사용하는 노드를 귀환 노드라 한다.

(정의10) 스케쥴링 과정중 어떤 한 시점에서 스케쥴링될 자격을 갖는 두개의 노드가 모두 동일한 자원 r에 대한 상충 노드이면 이 두 노드는 서로 중복된다고 한다.

(정의11) 상충 자원 r을 갖는 어떤 노드 n_0에 대한 접합 집합 JOINT(n_0, r) 을 다음과 같이 정의한다.

$$JOINT(n_0, r) = EXG(n_0, r) \cup \sum_{i=1}^{k} EXG(n_i, r)$$

단, 노드 n_i는 노드 n_{i-1}의 부모 노드이면서 귀환 노드

노드 n이 귀환 노드를 부모로 갖지 않을 때 이 노드에 대한 접합 집합 JOINT(n, r) 은 그 배타적 그룹 EXG(n, r)과 같다. 그리고 위 식에서 노드 n_k를 노드 n_0의 최종 노드라 한다.

(정의12) 어떤 노드 n에 대한 접합 집합 JOINT(n, r)에 속하는 모든 노드 i의 데스티네이션 레지스터 중 상충 자원만의 집합을 COLLECT(n) 이라 한다. 즉,

$$COLLECT(n) = \{r_i \mid \forall i \in JOINT(n, r),\ dest(i) = \text{상충 자원 } r_i\}$$

(정의13) 상충 자원 r_1을 갖고 있는 노드 n_1과 상충 자원 r_2를 갖는 노드 n_2에 대해 $n_1 \notin JOINT(n_2, r_2)$ 이고 $n_2 \notin JOINT(n_1, r_1)$일때 $dest(n_2) = r_2 \in COLLECT(n_1)$이고 $dest(n_1) = r_1 \in COLLECT(n_2)$라면 노드 n_1과 n_2는 맞물림되어 있다고 한다.

(정의14) 노드 n_1이 커버되어 있지만 완전히 커버되지 않았다면 노드 n_2가 노드 n_1의 배타적 관계에 있거나 맞물림되어 있다면 노드 n_2는 봉쇄된다고 한다.

(정의15) 노드 n_2가 노드 n_1의 배타적 관계에 있거나 맞물림되어 있을 때 노드 n_1이 완전히 커버되면 노드 n_2는 노드 n_1에 대해 해제된다고 한다.

그림2에서 모든 레지스터는 이 DAG의 범위 밖에서 dead하다고 가정하자. 맨처음에 DAG의 리프 노드가 되는 노드 1,3,5,7 이 스케쥴링될 자격을 갖게 된다. 노드1의 배타적 그룹 EXG(1,r4) = {1} 이고, 각 노드에 대한 배타적 그룹은 EXG(3,r1) = {3},

Machine code

```
1 sub  #10,R4
2 ld   0[R4],R2
3 add  #1,R12,R1
4 sub  R1,R2,R10
5 ld   A,R2
6 st   R2,0[R10]
7 mov  #20,R3
8 add  R3,R2
9 st   R2,-80[fp]
```

Corresponding DAG

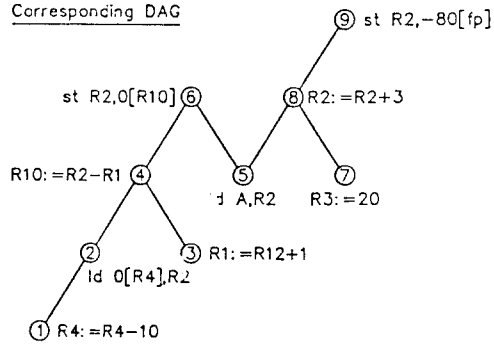

그림 2. DAG의 한 예
Fig. 2. An example of DAG.

EXG(5,r2) = {1,2,3,4,5,6,7,8}, EXG(7,r3) = {6} 과 같이 구해진다. 이제 각 노드의 접합 집합은 노드5를 제외한 노드들이 자신의 부모 노드로서 귀환 노드를 갖지 않으므로 각 노드의 배타적 그룹과 같게 된다. 노드5의 접합 집합은 노드8을 귀환 노드로 갖기 때문에 노드8의 배타적 그룹 EXG(8,r2)= {8,9} 을 구한 후, EXG(5)와 EXG(8)의 합으로 주어진다. 즉, JOINT(5,r2) = EXG(5,r2) + EXG(8, r2) = {1,2,3,4,5,6,7,8,9} 가 된다. 노드 1,3,7은 스케쥴링될 완전한 조건을 갖고 있지만 노드5는 노드2가 완전히 커버되기 전에는 절대로 스케쥴링될 수 없다. 이러한 제약은 미리 검사하여 올바른 스케쥴링이 수행될 수 있도록 해야 하는데, 이를 위하여 위와 같이 접합 집합 JOINT(n, r) 을 구하고 이 접합 집합의 노드 중에서 노드 n의 부모 노드가 아닌 노드가 데스티네이션으로 자원 r을 갖는다면 노드 n은 스케쥴링될 자격을 갖는 대상에서 제외한다. 즉, 노드 2 ∈ JOINT(5,r2) 이고, dest(2) = r2가 되므로 노드5는 스케쥴링될 후보자 노드가 될 수 없다.

그림3은 상충 자원이 되는 각 명령어의 데스티네이션 필드만을 표시한 간단한 DAG이다. 여기서 노드1이 스케쥴링되어 있다고 하자. 그러면 그 다음

스케쥴링의 자격을 갖는 노드는 유일하게 노드 2 가 된다. 먼저 노드 6 은 데스티네이션으로 상충 자원 r2 를 갖고 있기 때문에 노드 1 이 완전히 커버되기 전에는 스케쥴링될 수 없다. 노드 5 는 노드 1 과 맞물림 관계가 소멸되었을 때 비로소 스케쥴링될 자격을 갖게 된다. 즉, 노드 1 의 배타적 그룹 EXG(1, r2) 는 노드 1 이 스케쥴링된 시점에서 {2,3,4} 가 되고, 접합 집합 JOINT(1,r2) 도 마찬가지로 {2,3,4} 가 된다. 접합 집합에 속하는 노드들이 갖는 상충 자원의 집합 COLLECT(1)= {dest(3)} = {r1} 이 된다. 노드 5 의 배타적 그룹과 접합 집합은 EXG(5,r1) = JOINT(5,r1) = {5,6,7} 이 되고 COLLECT(5)= {dest(5), dest(7)} ={r1, r2}가 된다. 맞물림 관계에 의한 데드록 상황이 발생되지 않도록 하기 위해서 이미 스케쥴링된 노드 n_k의 COLLECT(n_k)를 구하고 스케쥴링될 후보자가 되는 노드 n_i의 COLLECT(n_i)를 구하여 dest(n_k) ∈ COLLECT(n_i), dest(n_i) ∈ COLLECT(n_k)가 되면 노드 n_i는 스케쥴링될 자격을 갖는 대상에서 제외시킨다. 따라서 노드 1 이 스케쥴링 되어 있는 시점에서, 그림 3 의 DAG로 부터 dest(1) = r2 ∈ COLLECT(5)이고 dest(5) = r1 ∈ COLLECT(1)인 조건때문에 노드 5 는 스케쥴링될 자격을 갖지 못하게 된다.

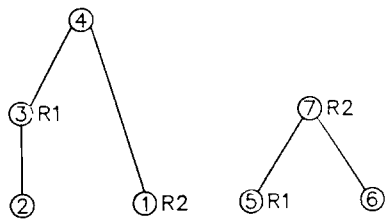

그림 3. 간단한 DAG
Fig. 3. A simple DAG.

Ⅲ. 코드 스케쥴링 알고리즘

명령어 코드의 전체 수행 시간 및 기억 용량을 감소시키기 위하여, 먼저 기본 블럭으로 나누어진 원시 명령어 스퀀스에 대해 방향성 비순환 그래프(DAG)를 구성하고 명령어간의 인터록 거리를 나타낸 다음, 자원 상충 레지스터를 구하고 이를 이용하여 인터록이 해소된 명령어의 시퀀스를 생성하도록 한다. 이 과정에 대한 스케쥴링 알고리듬은 다음과 같다.

입력 : 인터록이 고려되지 않고 생성된 기본 블럭의 명령어의 시퀀스

출력 : 인터록을 제거하고 원시 명령어 시퀀스와 동일한 결과를 갖도록 최소한의 nop 이 삽입된 명령어의 시퀀스

절차 : 모든 기본 블럭들에 대해 다음 과정을 반복 수행한다.

(1) 기본 블럭을 분석하여 방향성 비순환 그래프 구성

(2) 생성될 수 있는 명령어 집합을 결정

(3) 주어진 선택 순서에 의한 명령어 선택

(4) 선택된 명령어에 대한 적법성 검사

1. DAG 의 구성

DAG 의 한 노드는 하나의 명령어를 나타내며 노드 번호는 한 기본 블럭내의 명령어 시퀀스의 순서 번호로 주어진다. 즉, 기본 블럭의 k 번째 명령어는 노드 k 로 모델화한다. 노드 간의 에지는 부모 노드가 자식 노드의 데스티네이션 레지스터를 자신의 소스 레지스터로 사용할 경우에 만들어 진다. 각 노드에는 부모 노드와의 인터록 거리가 주어지며 루트로 부터의 깊이 d 가 계산되어 주어진다.

2. 코드 스케쥴링

병렬적인 DAG $D_1, D_2, \cdots, D_n$이 i 번째의 노드를 선택하기 위해 스케쥴링을 수행할 시점 t 에서 각각 커버된 집합 $C_1, C_2, \cdots, C_n$을 갖는다고 하자. 그리고

$$D = \sum_{k=1}^{n} D_k, \qquad Ct = \sum_{k=1}^{n} C_k$$

라 정의한다. 즉, D 는 기본 블럭의 모든 노드를 포함하는 전체 DAG 가 되고 C_t는 시점 t 에서 이미 커버되어 있는 노드의 집합이다. 그러면 C_t 를 갖는 DAG D 가 해제되어야 할 상충 노드들의 집합 CONFLICT = {n_k | 해제되지 않은 상충 노드 $n_k \in C_t$} 를 갖는다고 하자.

〔단계 1〕 $n_i \in (D - C_t)$ 이고, n_i의 모든 지식 $B = \sum_{j=1}^{m} b_j$에 대하여 $b_j \in C_t$ 이며, n_i와 이미 스케쥴링된 b_j간의 거리 ℓ이 b_j와 n_i간에 주어진 인터록 길이보다 크거나 같으면 노드 n_i를 선택될 가능성을 갖는 후보자 노드 집합 NODE에 속하도록 한다.

〔단계 2〕 해제된 자원을 데스티네이션으로 하는 노드를 NODE 에 편입시킨다.

〔단계 3〕 후보자 노드 집합 NODE 에 새로 추가되는 모든 노드 n_i에 대해

1) dest(n_i)가 상충 자원이 아니면
배타적 그룹 EXG(n_i) = {n_i}로 한다.

2) dest(n_i)가 상충 자원이면

a) 배타적 그룹 EXG(n_i)와 접합 JOINT(n_i)를 구하고, $n_j \in$ JOINT(n_i)이며 n_j가 n_i의 부모 노드가 아

널 때, dest(n_j)=dest(n_i)이면 노드 n_i를 NODE에서 제외한다. 그렇지 않으면

b) COLLECT(n_i)를 구하고, 노드 n_k∈CONFLICT에 대해

dest(n_k) ∈COLLECT(n_i),
dest(n_i)∈COLLECT(n_k)

이면 노드 n_i를 NODE에서 제외한다.

〔단계4〕 메모리 참조 우선 순위를 검사한다.

1) 노드 n_i가 load/store로서 메모리 우선 순위를 지키지 못할 경우에는 NODE에서 n_i를 제외한다.

2) dest(n_i)=상충 자원인 노드 n_i에 대해 n_a∈{집합 그룹 JOINT(n_i)−n_i}인 노드 n_a가 load/store이고 메모리 참조 우선 순위가 n_a<n_b인 노드 n_b가 커버되어 있지 않다면 노드 n_b의 모든 자손들의 집합 SUCC(n_b)를 구하여, n_c∈SUCC(n_b)에 대해 dest(n_c)=dest(n_i)이면 노드 n_i를 제외시킨다.

〔단계5〕 후보자 노드 집합 NODE에 속한 임의의 2개의 노드쌍(n_p, n_q)가 서로 중복되거나 맞물림될때

1) JOINT(n_p)와 JOINT(n_q)가 동일한 메모리 위치를 참조하는 명령어를 포함하고 있다면, 노드 n_p와 n_q중에서 메모리 참조 우선 순위가 낮은 명령어를 JOINT에 포함하고 있는 노드를 봉쇄한다.

2) 노드쌍의 각 노드의 깊이를 비교하여, DEPTH(n_p)>DEPTH(n_q)이면 노드 n_q를 봉쇄한다.

3) 두 노드의 깊이가 같은 경우에는, 각 노드에 대한 여유를 구하여 여유가 더 작은 노드를 봉쇄하고 NODE에서 제외시킨다. 즉, MARGIN(n_p)> MARGIN(n_q)이면 노드 n_q가 봉쇄된다. 그렇지 않고,

4) MARGIN(n_p)=MARGIN(n_q)이면 상충 노드 MARGIN의 수가 작은 노드를 봉쇄한다.

〔단계6〕

1) NODE가 NULL이면 nop를 삽입시키고, 그렇지 않으면

2) 다음의 순서에 따라 선택한다.

a) 상충 노드이면서 완전히 커버되지 않은 배타적 그룹에 속하는 노드 중 깊이가 가장 큰 노드

b) 상충 노드가 아니고 완전히 커버되지 않은 배타적 그룹에 속하는 노드 중 깊이가 가장 큰 노드

c) 배타적 그룹에 속하지 않는 상충 노드 중 깊이가 가장 큰 노드

d) 나머지 노드들 중 깊이가 가장 큰 노드

e) 위 경우에서 깊이가 같은 경우에는 MARGIN이 큰 노드를 선택하고, 부모와의 인터록 거리가 0인 노드는 항상 가장 나중에 선택한다.

〔단계7〕 최종 선택된 n_i이 상충 노드이면 NODE에 속한 상충 노드 n_i와

dest(n_i) ∈COLLECT(n_i),
dest(n_i) ∈COLLECT(n_k)

이면 노드 n_i를 NODE에서 제외한다.

〔단계8〕 D−C_t=NULL이 될 때까지 알고리듬을 반복 수행한다.

(정리1) 코드 스케쥴링 알고리듬은 절대로 맞물림에 의한 데드록을 발생시키지 않는다.

(증명) 스케쥴링된 코드 시퀀스의 수를 NS라 하자. NS=0이면, 즉 스케쥴링된 명령어가 하나도 없는 경우에는 어떠한 명령어를 선택하더라도 데이타 종속관계가 존재하지 않기 때문에 맞물림되지 않는다.

NS=L이라 가정하자. 스케쥴링될 자격을 갖는 코드가 지금 2개만 남아 있고 이 노드들이 맞물림이 되는 경우를 생각하자. 맞물림이 되지 않는 노드는 스케쥴링될 때 데드록에 아무런 영향을 미치지 않기 때문에 위 경우에 대한 고려만으로 충분하다. 따라서 코드 스케쥴링 알고리듬은 이러한 상태에 도달하지 않음을 보이면 된다.

노드 n_1과 n_2는 각각 데스티네이션 r1과 r2를 갖고 그 자식으로 각각 c_1과 c_2를 갖고 있고 노드 c_1과 c_2의 데스티네이션은 각각 r2, r1이라 하자. c_1은 이미 스케쥴링되어 있고 알고리즘이 c_2를 선택하려는 시점에 와 있는 상태라고 걱정하자.

c_2를 n_1보다 먼저 선택하려고 할 때 c_2는 상충노드이고 dest(c_2)=r1∈COLLECT(c_1)인데 이미 스케쥴링되어 있는 노드 c_1이 배타적 그룹 EXG(c_1)={n_1}에서 dest(n_1)=r1 즉, dest(n_1)=dest(c_2)가 된다. 또한 EXG(c_2)={n_2}에서 dest(n_2)=r2이고 완전히 커버되지 않은 노드 c_1의 데스티네이션 dest(c_1)=r2가 되므로 dest(c_1)=dest(n_2)가 된다. 따라서 c_2는 n_1이 커버되기 전에는 절대로 선택될 수 없다. 그러므로 이 코드 스케쥴링 알고리듬은 스케쥴링 되어 있는 L개의 노드에 대해 다음 스케쥴링 될 노드에 의한 맞물림은 절대로 발생되지 않는다.
(증명끝)

코드 스케쥴링 과정 중 서로 중복되는 노드쌍이나 맞물림이 발생된 노드쌍에서 어느 노드를 선택할 것인지를 결정하는 것은 중요한 문제이다. 서로 상충되는 노드쌍은 어느 한 노드가 선택되면 그 노드가 완전히 커버될 때까지 다른 노드는 절대로 선택될 수 없다. 따라서 어느 노드를 선택하느냐에 따라 스케쥴링 결과에 많은 영향을 미치게 된다. 본 논문에서는 이러한 문제를 고려하여 다음과 같은 웨이트 함수 DEPTH와 MARGIN을 사용하여 선택 결정을

한다.

$$DEPTH(n_0)=\begin{cases}\text{루트로부터 노드 } n_0\text{까지의 패스상의 에지수} \\ \quad (n_0\text{의 부모 노드가 귀환 노드가 아닌 경우}) \\ \text{루트로부터 최종 귀환 노드 } n_k\text{까지의 패스상의 에지수} \\ \quad (n_0\text{의 부모 노드가 귀환 노드인 경우})\end{cases}$$

$$MARGIN(n)=NODE-JOINT(n)$$

커버된 상충 노드는 가급적 빨리 해제해 주어야 코드 선택 폭을 크게해 줄 수 있다. 상충 노드의 배타적 그룹에 속한 노드를 전부 커버하여야 그 상충 노드를 해제해 주게 되는데 스케줄링 과정에서 배타적 그룹내의 노드 간에 인터록이 발생할 때 가급적이 배타적 그룹에 속하지 않은 노드를 사용하여 해결하면 스케줄링 효율을 높일 수 있다.

구성된 DAG에 대해 코드 스케줄링을 수행하기 위해서는 다음과 같은 제약이 고려된다.

(제약1) 기본 블럭의 입구와 출구에서 live한 레지스터를 고려한다. 입구에서 live한 레지스터 집합 중 상충 레지스터를 소스 레지스터로 하는 노드들은 이에 대한 다른 상충 노드보다 먼저 커버되어야 하며, 출구에서 live한 레지스터를 데스티네이션 레지스터로 갖는 상충 노드는 이와 상충되는 노드들 중 가장 나중에 선택되어야 한다.

(제약2) 기본 블럭의 최종 명령어인 분기 명령어는 스케줄링된 명령어 시퀀스의 제일 마지막에 그대로 있어야 한다.

(제약3) 동일한 기억 장소의 내용을 후에 변경시키는 명령어에 대해서는 반드시 원래의 순서를 유지하도록 한다. 이를 위해서 스케줄링하게 될 노드의 배타적 그룹을 구하여 명령어 $j \in EXG(n)$가 load나 store일 경우에는 메모리 억세스의 적법한 순서를 갖는지 확인한다.

메모리 참조로 인한 데드록을 피하기 위해 코드 스케줄링을 수행할 때 알고리즘의 단계4-1과 단계5-1에서 제약3을 고려해 주어 원래의 코드 시퀀스의 출력 결과와 동일한 결과를 갖도록 보장해 주어야 한다. 따라서 다음과 같은 명령어 수행 순위를 지켜야 한다.

1. load 명령의 메모리 인출 장소(memory reading location)와 store 명령어의 메모리 기입 장소 (memory writing location)가 같은 경우에는 원래의 명령어 수행 순서를 유지한다.

2. 동일한 메모리 기입 장소를 갖는 store 명령어들은 그대로 원래의 명령어 수행 순서를 유지하여야 한다.

3. 동일한 메모리 인출 장소를 갖는 load 명령어들은 위의 두가지 조건을 만족시키기만 하면 그 수행 순서는 어떠한 순서가 되더라도 허용 된다.

주어진 코드 시퀀스에 대한 DAG의 구성은 레지스터 자원의 사용에 관한 종속 관계만을 표현하기 때문에 메모리와 같은 레지스터 이외의 자원 사용의 관계가 포함되어 있지 않다. 따라서 동일한 메모리 장소를 사용하는 명령어를 찾아 반드시 지켜야 할 명령어 수행 순위를 명시해 주어야 한다. 메모리 억세스 순서에 대한 제약이 코드 스케줄링 수행에 주는 영향을 알아보기 위해 그림4의 코드 시퀀스와 이에 대한 DAG를 생각해 보자.

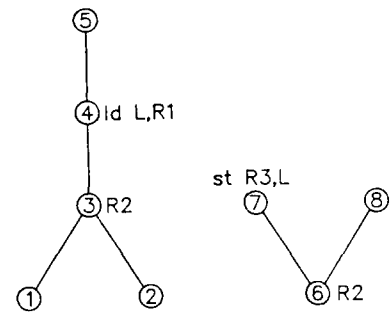

그림 4. 메모리 억세스 순서에 의한 스케줄링 제약
Fig. 4. The scheduling constraint by memory access ordering.

그림4의 DAG에서 노드 4와 7은 동일한 메모리 위치를 참조하는 명령어이기 때문에 메모리 참조 순서를 적법하게 유지시키기 위해 노드4는 노드7보다 어떠한 경우라도 먼저 스케줄링되어야 한다. 이제 노드6을 스케줄링하려는 시점에서 이 노드에 대한 접합 집합을 구하면 $JOINT(6,r2)=\{6,7,8\}$이 되고, 메모리 참조 순서의 제약을 받는 명령어를 포함하고 있고, 이 노드의 데스티네이션 dest(6)=r2는 상충 노드이기 때문에 알고리듬 단계 4-2를 적용하여 메모리 참조 우선 순위를 검사한다. 노드4는 노드7보다 메모리 참조 우선 순위가 높기 때문에 노드4의 모든 자손들의 집합을 구하면 $SUCC(4)=\{3\}$이 되고 dest(3)=r2=dest(6)이므로 노드6은 스케줄링될 수 없게 된다.

Machine code

```
1 ld   -4[fp],R1
2 add  #1,R1,R2
  . . .
3 ld   F,R2
4 st   R2,-4[fp]
5 sub  #1,R2,R1
```

Corresponding DAG

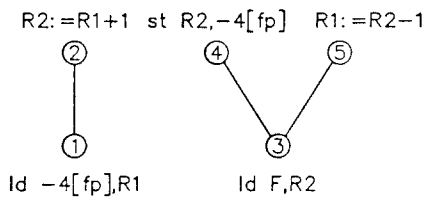

그림 5. 스케쥴링에 대한 메모리 억세스의 영향
Fig. 5. The effect of memory access for scheduling.

Machine code

```
1 ld   -4[fp],R1
2 add  #1,R1,R2
  . . .
3 ld   F,R2
4 st   R2,-4[fp]
5 sub  #1,R2,R3
```

Corresponding DAG

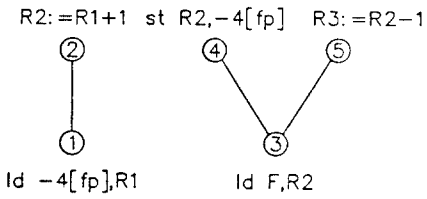

그림 6. 메모리 억세스 순서를 고려한 스케쥴링
Fig. 6. Scheduling considering the effect of memory access order.

그림 5에서 노드 1과 4는 동일한 기준 레지스터(base register)와 변위(offset)를 갖기 때문에 동일한 메모리 위치를 참조하는 명령어들이므로 노드1은 노드 4보다 언제나 먼저 커버되어야 한다. 이 DAG에 대해 스케쥴링을 수행 할 때 노드 1과 3은 자기 자식을 갖고 있지 않으므로 선택될 후보가 될 수 있다. 노드 3을 먼저 스케쥴링하게 되면 상충 자원 r1, r2에 대한 맞물림 관계 때문에 노드 4,5가 커버되기 전에 노드 1은 커버될 수 없게 되고, 따라서 메모리 참조의 우선 순위를 위반하게 되므로 적법한 스케쥴링이 될 수 없다. 그러므로 스케쥴링을 시작할 때 노드 1이 노드 3보다 반드시 먼저 선택되어야 한다.

그림 6의 DAG를 생각해 보면 위와 마찬가지로 최초로 선택될 자격을 갖는 노드들은 1과 3이 된다. 노드 3을 먼저 스케쥴링한다고 하면 노드 1과 3은 서로 맞물림 관계가 아니기 때문에 노드 4를 노드 1보다 나중에 스케쥴링해 주기만 하면 된다. 즉, 노드 시퀀스 3-4-5-1-2 혹은 3-5-4-1-2의 스케쥴링은 적법하다.

Ⅳ. 적용 예 및 검토

1. 대상 머신

본 논문에서 제안한 스케쥴러의 효율성을 보이기 위해 하드웨어 인터록이 없는 MIPS(microprocessor without interlock pipe stages)에 적용한다.[1],[11] RISC 프로세서인 MIPS는 load/store 아키텍춰로서 모든 오퍼랜드는 사용되기 전에 load 명령어에 의해 레지스터에 load 되어야 하고 메모리의 내용은 store 명령어에 의해서만 변경될 수 있다. MIPS는 서로 다른 스테이지를 점유하여 3개의 명령어를 동시에 수행하는 5-스테이지 파이프라인을 사용한다. 그림7은 파이프라인의 각 스테이지의 기능을 나타내고, 그림 8은 파이프라인의 명령어 수행 과정을 나타낸다.

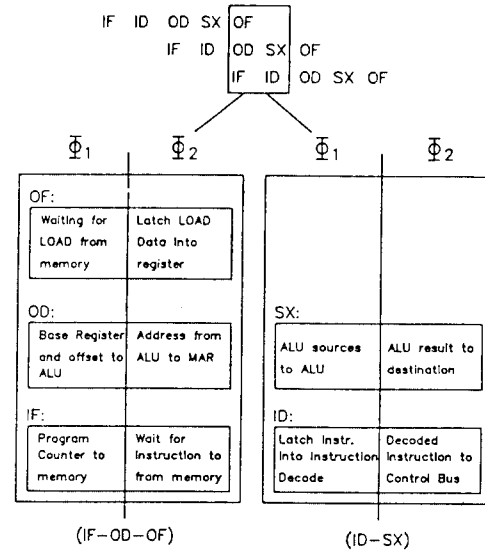

그림 7. 명령어 수행 시퀀스
Fig. 7. Instruction execution sequence.

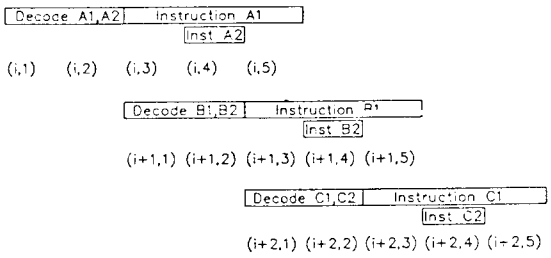

그림 8. 파이프라인의 명령어 수행
Fig. 8. Pipelined execution of instructions.

수행되는 각 명령어는 ALU를 OD-스테이지에서 SX-스테이지에서 각각 한번씩 사용할 수 있다. 명령어는 OD-스테이지에서 ALU를 어드레스 계산을 위해 사용하고 SX-스테이지에서는 ALU 오퍼레이션을 위해 사용하거나 OD와 SX를 모두 ALU 오퍼레이션을 위해 사용할 수 있다. 즉, 각 명령어는 2개의 독립적인 명령어를 합하여 구성된다. 그림 8은 실제의 명령어간의 관계를 보여준다. 2사이클로 한 명령어 워드의 페치와 디코드가 이루어져 2개의 명령어가 수행될 준비를 끝마치게 된다. 엔코딩 공간은 고정되어 있기 때문에 2개의 명령어를 합칠 때에는 두번째 오퍼레이션에는 제약이 주어진다. 어드레스 계산을 위한 long immekliate field를 갖는 명령어에는 두번째 ALU 오퍼레이션을 엔코드할 만한 공간이 없게 된다. 이러한 경우에는 SX-스테이지 동안 ALU는 유휴 상태가 된다. 산술 오퍼레이션의 결과는 파이프스테이지 3(OD) 와 4(SX)에서 레지스터에 기입될 수 있다. Load 명령어는 파이프 스테이지 5(OF) 동안에 레지스터에 기입한다. 레지스터는 어드레스 계산을 위한 소스로서 OD-스테이지에서 사용되거나 산술 오퍼레이션을 위한 소스로서 OD 및 SX-스테이지에서 사용된다.

메모리 시스템의 구성 때문에 메모리 억세스 명령어 사이에 인터록이 발생한다. 표 1은 2개의 명령어 사이에 존재할 수 있는 인터록 거리를 나타낸 것이다. 스케쥴러는 인터록 거리 만큼의 지연 슬로트를 자원 의존 관계가 없는 명령어나 nop을 채워줌으로서 코드 시퀀스의 재구성을 시도하게 된다.

2. 예제 및 검토

알고리즘의 수행 과정을 설명하기 위해 한 예로 다음과 같은 한 기본 블록의 명령어 시퀀스에 대해 적용한다. 이 명령어 시퀀스에 대해 작성된 방향성 비순환 그래프는 그림 9와 같이 된다.

표 1. 명령어 쌍에 대한 인터록 거리
Table 1. Pipelined execution of instructions.

Instruction Pair (Inst. 1-Inst. 2)	Interlock Length	Comment
Load-Compute	2	
Load-Address	2	load된 값을 address로 사용
Load-Data	1	load된 값을 data store 를 위해 사용
Compute-Compute	0	
Compute-Address	0	계산된 값을 address 로 사용
Compute-Data	0	계산 결과를 data store 를 위해 사용
Data-Compute	1	data store 후 계산 결과를 그 레지스터에 기입

Sample instruction sequence

```
1 ld   A,R2
2 ld   B,R3
3 add  R3,R2,R4
4 st   R4,X[R3]
5 ld   C,R5
6 sub  R2,R5,R3
7 ld   -1[fp],R4
8 add  #2,R4
9 st   R4,Y[R3]
```

Corresponding DAG

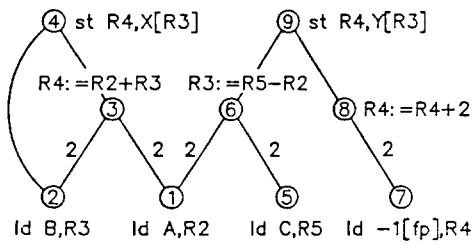

그림 9. 코드 시퀀스 및 이에 대한 DAG
Fig. 9. Code sequence and resulting DAG.

문제를 간단히 하기 위해서 기본 블럭의 출구에서의 모든 레지스터는 dead하고 이 기본 블럭내에서는 메모리를 억세스하는 명령어들 간에 aliasing이 존재하지 않는다고 하자. 따라서 제약 1과 제약 3은 항상 만족되는 것으로 생각하면 된다. 그림 6의 DAG에서 초기에는 커버된 노드의 집합 $C_t = \{\}$ 이다. 이 DAG

의 리프 노드가 되는 노드 1,2,5,7이 맨 먼저 선택될 수 있는 자격을 갖는데 노드 1을 가장 먼저 스케쥴링한다고 하자. 이것은 코드 스케쥴링을 하기 위한 여러제약 조건에 대한 고려를 할 필요를 없게 한다. 노드 1이 스케쥴링이 되면 $C_t = \{1\}$ 이 되고 단계1,2에서 NODE = {2,5,7} 이 되고 단계 3에서 각 노드에 대한 배타적 그룹을 구하면 EXG(2)= {2,3,4} EXG(5)= {5}, EXG(7)= {7,8} 이 되고, 접합 집합을 구하면 JOINT(2)= {2,3,4}, JOINT(5)= {5}, JOINT (7)=EXG(7)+EXG(8)= {5,6,7,8,9} 가 된다. 단계 6에서 노드 2와 7은 상충 노드이고 깊이가 모두 2이므로, MARGIN(2)=NODE−JOINT(2)= {2,5,7} − {2, 3,4} ={5, 7}, MARGIN(7)=NODE − JOINT(7)= {2,5,7} − {5,6,7,8,9} = {2} 이므로 MARGIN이 큰 노드2가 선택되어 지고 커버된 노드의 집합 Ct= {1,2} 가 된다. 노드 2가 선택이 되었기 때문에 단계 7에서 노드 7은 노드 2에 대해 맞물림 되어 있으므로 NODE에서 제외되어 NODE= {5} 가 되고 노드 7은 노드 2가 완전히 커버될 때까지 선택될 수 없게 된다.

다시 단계 1에서 노드 3은 노드 1과의 인터록이 여전히 발생하여 후보자 노드가 될 수 없으므로 NODE = {5} 가 된다. 마지막 단계 6에서 노드 2가 선택되고 $C_t = \{1,2,5\}$ 가 된다.

$D - C_t$ = NULL 이 될 때까지 이러한 방식으로 계속 수행하면 명령어들을 모두 스케쥴링할 수 있다. 그림 9의 코드 시퀀스에 대한 스케쥴링 최종 결과는 그림 10(b)와 같이 된다. 그림 10(a)는 스케쥴링하지 않고 생성한 명령어 시퀀스를 나타낸다.

```
1 ld   A,R2          1 ld   A,R2
2 ld   B,R3          2 ld   B,R3
3 nop                3 ld   C,R5
4 nop                4 nop
5 add  R3,R2,R4      5 add  R3,R2,R4
6 st   R4,X[R3]      6 st   R4,X[R3]
7 ld   C,R5          7 ld   -1[fp],R4
8 nop                8 sub  R2,R5,R3
9 nop                9 nop
10 sub R2,R5,R3      10 add #2,R4
11 ld  -1[fp],R4     11 st  R4,Y[R3]
12 nop
13 nop
14 add #2,R4
15 st  R4,Y[R3]
      (a)                  (b)
```

그림10. (a) 스케쥴링하지 않은 명령어 시퀀스
(b) 알고리즘 수행 결과

Fig. 10. (a) Legal instructions without scheduling.
(b) The result of scheduling execution.

표 2는 여러 프로그램에 대한 스케쥴링 수행 결과를 비교한 것이다. 이 표는 스케쥴링을 하지 않았을 때 필요한 nop의 백분율 및 알고리즘의 수행 결과로 요구되는 nop의 백분율을 보인다. 테스트 프로그램으로 다음과 같은 benchmark 프로그램을 사용하였다.

- Sieve : prime number를 구하는 프로그램
- Bsearch : binary search 프로그램
- Fibinacci 1 : fibonacci 급수를 계산하는 재귀적 프로그램
- Fibonacci 2 : fibonacci 급수를 계산하는 비재귀적 프로그램
- Bubble : bubble sort 프로그램
- Shell : shell sort 프로그램
- Median : 중앙값을 계산하는 통계 프로그램

표 2. 알고리듬 수행 후 삽입된 nop 수의 비교
Table 2. The comparison of nops inserted after executing algorithm.

프로그램명	명령어수	요구되는 nop(%)		
		스케쥴링하지 않을 경우	Gross방법	제안된 방법
Sieve	46	24.6	17.9	14.8
Bsearch	38	24.0	19.1	15.6
Fibonacci_1	60	10.4	3.2	3.2
Fibonacci_2	83	17.8	2.4	2.4
Bubble	26	27.8	25.7	23.5
Shell	38	20.8	17.4	15.6
Median	147	26.1	17.7	14.5

3. 알고리듬의 복잡도

본 알고리듬을 사용하여 명령어 시퀀스를 재구성하는데 요구되는 시간 복잡도를 C_t라 하면 다음과 같이 표현될 수 있다.

$$C_t = C_b + C_d + C_n + C_e$$

여기서

C_b는 방향성 비순환 그래프(DAG)를 구성하기 위한 시간 복잡도

C_d는 각 노드의 루트로부터의 깊이를 할당하기 위한 시간 복잡도

C_n는 스케쥴링될 후보자 노드 집합 NODE를 구하기 위한 시간 복잡도

C_e는 각 노드의 배타적 그룹을 구하기 위한 시간 복잡도

먼저 한 기본 블럭에 속한 명령어 시퀀스에 대한 DAG를 구성하기 위해 필요한 시간 복잡도를 알아보자. DAG의 표현은 한 명령어를 한 노드로 나타내고, 레지스터 사용에 관한 정보에 따라 노드간의 에지가 만들어진다. 즉, 노드간의 에지는 한 명령어에 대응하는 한 노드가 생성되고, 그 노드가 소스로 사용하는 레지스터를, 그 노드가 생성된 시점에서 가장 최근에 데스티네이션으로 정의한 노드와 에지로 연결 되므로 시간 복잡도 C_b는 형성되는 에지수에 비례하게 된다. 한 기본 블럭이 N개의 명령어로 이루어져 있고, 한 명령어가 사용하는 레지스터의 수를 U라고 하고 가장 최근에 정의된 레지스터를 갖는 노드를 찾는데 걸리는 시간을 T_d라고 하면, DAG를 구성하기 위해 걸리는 수행 시간은 $U*T_d*N$이 된다. U는 실제로 작은 값을 갖는다. 전형적인 3-주소 형식의 명령어를 사용할 경우에 U의 값은 2가 되고 T_d도 일정시간(costant time)이라 할 수 있으므로 DAG를 구성하기 위해 요구되는 시간 복잡도는 O(N)이다.

DAG가 구성된 다음, 각 노드의 루트로 부터의 깊이를 계산하기 위해 DAG에 대한 토폴로지칼 정렬을 한 후에 토폴로지칼 정렬의 역순대로 각 노드 n_i의 깊이는 노드 n_i에서 노드 n_j로의 각각의 에지 e_{ij}에 대해 $DEPTH(n_i)=Max(DEPTH(n_i), DEPTH(n_i)+1)$을 반복 계산함으로서 주어진다. 토폴로지칼 정렬을 수행하는데 걸리는 시간은 에지의 수 E와 노드의 수N에 비례하게 되어 O(E+N)이 되는데 에지의 수는 각 노드의 사용되는 레지스터의 수U와 노드의 수 N의 곱의 형태가 되므로 그 시간 복잡도는 $O(N+E)=O((U+1)*N)$ 즉, O(N)이 된다.[17] 한편 각 노드의 깊이의 계산은 에지수 만큼 수행되므로 깊이 계산을 위한 시간 복잡도는 $O(E)=O(U*N)=O(N)$이 되고, 모든 노드에 대해 깊이를 할당하기 위한 시간 복잡도 C_d는 토폴로지칼 정렬과 모든 노드에 대한 깊이 계산을 위한 시간 복잡도의 합으로 표현되므로 O(N)이 된다.

이제 알고리듬의 단계 1과 단계 2의 수행으로 스케줄링될 후보자 노드 집합 NODE를 구하기 위한 수행 시간은 자신의 모든 자식 노드들이 이미 커버된 노드를 발견하기 위한 DAG의 탐색 시간이 된다. 이미 i개의 노드가 커버되어 있는 시점에서 NODE를 구하기 위한 수행 시간은 N-i개의 노드들과 이 노드들이 갖는 에지의 수 E_{N-i}의 합에 비례하게 되고, 또한 후보자 노드 집합 NODE를 구하는 과정은 모든 노드가 선택될 때까지 반복 수행된다. 따라서 시간의 복잡도는 $O((N-i)+E_{N-i})$의 형태로 계산될 수 있고, $N-i<=N$이고, $E_{N-i}<=E$가 되므로 $O(N*(N+E))$로 bound될 수 있으므로 스케줄링될 후보자 노드 집합을 구하는 시간 복잡도 C_n은 $O(N^2)$이 된다.

이제 단계 3에서 NODE에 추가되는 노드에 대한 배타적 그룹을 구하게 되는데, 한 노드에 대한 배타적 그룹은 그 노드의 부모 노드들과 아직 커버되지 않은 자식 노드들이 되므로 DAG의 각 노드를 방문하여 이를 찾기 위한 시간은 BFS(breadth first search) 알고리듬을 사용하면 최악의 경우를 가정하여 $O(N+E)=O(N)$의 시간 복잡도를 갖게 된다. 한 노드의 배타적 그룹은 최초에 DAG를 한번만 탐색하여 얻을 수 있으며 어떤 노드가 스케줄링되었을 때는 단지 스케줄링된 노드를 배타적 그룹에서 제외해 주기만 하면 되기 때문에 거의 일정한 시간으로도 가능하다. 따라서 모든 노드N에 대한 배타적 그룹을 구하기 위한 시간 복잡도 C_e는 $O(N^2)$이 된다.

단계 4 에서 단계 8 까지의 오퍼레이션들은 위 단계에서 구해진 정보들을 사용한 비트 오퍼레이션이 대부분이고 일정 시간으로 수행될 수 있음은 명확하다.

따라서 명령어 시퀀스를 스케줄링하기 위한 알고리즘의 수행에 필요한 시간 복잡도C_t는 $O(N^2)$이 된다.

V. 결 론

본 논문에서는 하드웨어 인터록을 사용하지 않는 파이프라인 아키텍춰의 타이밍 해저드를 소프트웨어 스케줄링으로 해결하여 파이프라인의 패러랠리즘을 효율적으로 이용할 수 있는 코드 스케줄링 방식을 제안하였다.

코드 재구성을 수행할 때 선택될 가능성을 갖는 노드들에 대하여 적법한 코드 스케줄링을 위한 제약조건의 계산을 최소화시키고 스케줄링될 자격을 갖는 자원 상충 노드들에 가중치를 주고 그 순서에 의해 스케줄링을 함으로써 차후의 노드 선택 범위를 확장해 주게 되어 효율적으로 코드를 생성할 수 있게 하였다.

본 논문에 스케줄링 알고리듬은 C언어로 스케줄러를 구성하였고, MIPS에 적용시켜 효율성을 보였다.

본 논문에서 제안한 알고리듬은 파이프라인 인터록이 있는 CISC에 적용하여 프로그램 실행 시간을 단축하는데 이용할 수 있다.

參 考 文 獻

[1] J.L. Hennessy, "VLSI Processor Archit-

ecture," *IEEE Trans. on Computers,* vol. C-33, no. 12, pp. 1221-1246, Dec. 1984.

[2] A. Silbey, V. Milutinovic, V. Mendoza-Grado, "A Survey of Advanced Microp rocessors and High-Level Language Compute Architecture," Tutorial on Advanced Microprocessor & HLL Computer Architecture, pp. 118-141, *IEEE Computer Society Press,* 1986.

[3] M.E. Hopkins, "A Definition of RISC," Proc. of Int. Workshop on High-Level Architecture, pp. 3.8-3.11, May 1984.

[4] G. Radin, "The 801 minicomputer," IBM J. Res. Development, vol. 27, no. 3, pp. 237-246, May 1983.

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

[6] D.A. Patterson, "Reduced Instruction Set Computer," Communications of ACM, vol. 28, no. 1, pp. 8-21, Jan. 1985.

[7] 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.

[8] P.M. Kogge, "The Architecture of Pipelined Computers," McGraw-Hill, New York, 1981.

[9] J.R. Goodman, et al, "PIPE: A VLSI Decoupled Architecture," *Proc. of IEEE/ 12th ISCA,* June 1985.

[10] R. Sites, "Instruction Ordering for the Cray-1 Computer," Tech. Rep. 78-CS-023, Dept. of C.S., Univ. of California, San Diego, July 1978.

[11] 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.

[12] J. Mousouris, et al, "A CMOS RISC Processor with Integrated System Functions," Proceedings COMPCON, pp. 126-131, Mar. 1986.

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

[14] M.G.H. Katevenis, "Reduced Instruction Set Computer Architecture for VLSI," Ph.D. Thesis Univ. of California at Berkeley, Oct. 1983.

[15] H.S. Stone, "High-Performance Computer Architecture," Addison Wesley, pp. 155-164, 1987.

[16] P. Chow, "MIPS-X Instruction Set and Programmer's Manual," Technical Report no. 86-289, pp. 7-11, May 1986.

[17] J. Welsh, J. Elder, D. Bustard, "Sequential Program Structures," Prentice-Hall, pp. 282-289, 1984.

[18] 김은성, 임인칠, "파이프라인 아키텍춰를 위한 코드 스케쥴링 알고리즘," 한국정보과학회87봄 학술발표 논문집, pp. 165-169, 4. 1987. *

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 대우 교수 University of Illinois at Urbana-Champaign의 Computer Science학과 Visiting Profossor역임. 1964年 한양대학교 강사. 조교수, 부교수. 1977年~현재 한양대학교 교수 재직, 현재 동대학교 전자계산소 소장. IEEE Computer Society Korea Chapter Chairman. 주연구분야는 Computer Architecture, RISC Processor 및 Compiler설계, VLSI Testing/Testable Design, Simulation, Layout, AI기법을 이용한 VLSI 설계 및 Machine-Translation등임.

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.