NOP 명령어 슬롯을 활용하는 VLIW 코드 생성기법

문현주O, 이승수, 김석주, 김석일

충북대학교 컴퓨터과학과

A VLIW Code Generation Technique Utilizing NOP Instruction Slot

Hyun-Ju Moon O Seungsoo Lee Sukju Kim Sukil Kim

Dept. of Computer Science, Chungbuk National University

요 약

본 논문에서는 VLIW 목적코드에 존재하는 NOP 명령어 슬롯에 의미있는 명령어를 중복 삽입하도록 함으로써 원래의 방법에서 존재하였던 자료의존관계를 해소하여 실행시간의 지연을 방지하는 기법을 연구하였다. 이 경우에 하나의 긴 명령어에 동일한 명령어가 둘 이상 포함될 수 있으므로 연산 단계에 이은 쓰기 단계에서 여러개의 명령어가 동일한 레지스터 파일의 주소에 쓰기를 함에 따른 충돌을 피할 수 없다. 본 논문에서는 연산처리기 별로 쓰기 단계에서 연산 결과를 레지스터 파일에 쓰도록 허용할 것인지에 대한 정보를 명령어에 포함하는 TiPS 구조와 TiPS 구조에 적합한 목적코드 생성 알고리즘을 제안하였다. 목적코드 생성 알고리즘은 연산처리기별로 연속적으로 실행되는 명령어간의 자료의존관계를 해소하기 위하여 NOP 대신에 다른 연산처리기에서 실행할 명령어를 수행하도록 동일한 명령어를 복사하여 할당할 수 있다. 실험 결과, 명령어 복사 기법은 기존의 기법에 비하여 전체 실행 사이클을 크게 단축시킬 수 있음을 보여주었다.

1. 서론

최근의 프로세서 구조는 ILP(Instruction-Level Parallelism)를 활용하기 위한 다양한 기술들을 채택하고 있다. 이들 중 각광받는 구조로는 하드웨어에 의해 스케쥴 되는 수퍼스칼라 구조와 컴파일러에 의하여 스케쥴 되는 VLIW 구조등이 있다. 특히 VLIW 구조에서는 정적 스케쥴링 기법을 이용하여 프로그램 전체를 대상으로 병렬성을 추출하므로 보다 높은 병렬성 추출 효과를 얻을 수 있는 장점이 있다[2,3,8].

VLIW 프로세서는 여러 개의 단위 명령어(unit instruction)를 긴 명령어(long instruction word) 단위로 구성하고 긴명령어 단위로 명령어 파이프라인 단계가 진행된다. 따라서 컴파일러는 컴파일 과정에서 긴명령어의 실행 형태를 정확히 판단하여 그 실행 형태를 정확히 유지할 수 있도록 목적코드를 생성해야 한다[4-6]. 이 과정에서 명령어 수준의 병렬성이 충분하지 못하거나 연산처리기의 충돌이 예상될 때에는 긴 명령어를 구성하는 단위명령어 중 일부를 NOP(No Operation)으로 채우거나(수평낭비: horizontal waste), 긴명령어 전체를 NOP으로 채워야(수직낭비: vertical waste) 한다[10]. 목적코드에서 NOP이 차지하는 비중은 결국 응용 프로그램에서 자료의존관계가 복잡해질수록 점차 증가한다[7,8].

VLIW용 목적코드에서 NOP을 대신하여 의미있는 명령어를 복사하여 명령어간의 자료의존관계를 해결할 수 있다면 낭비되는 코드 슬롯을 이용하여 응용 프로그램의 빠른 수행이 가능하다. 그런데 이러한 명령어의 복사는 하나의 긴 명령어내에 동일한 명령어가 둘 이상 포함되도록 하므로 명령어 파이프라인 단계의 쓰기(write-back) 단계에서 이들 명령어를 수행하는 연산처리기들이 동시에 동일한 레지스터 주소를 갱신하는 충돌을 야기하게 된다. 따라서 명령어 복사 기법이 가능하기 위해서는 원래의 명령어를 제외한 모든 복사된 명령어의 연산 결과가 명령어 쓰기 단계에서 비활성화되도록 하는 하드웨어 구조가 필요하다. 본 논문에서는 명령어 별로 명령어 쓰기 단계에서 활성화여부를 결정하는 정보를 코드에 삽입하도록 하고 이 정보에 의하여 각 명령어의 쓰기 여부를 결정하도록 하는 TiPS 구조를 제안하고 TiPS 구조에서 사용할 수 있는 목적 코드 생성 기법을 제안하였다.

2. 코드 복사와 프로세서 구조

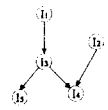
2.1 코드 분석

VLIW 기본 구조는 명령어 캐쉬, 데이터 캐쉬, 다중포트 레지스터 파일과 여러개의 연산처리기(Function Unit: FU)로 구성된다. 명령어를 처리하는 파이프라인은 명령어 인출, 디코딩, 연산 및 쓰기 단계의 네단계로 구성된다. 즉, 프로세서는 명령어 캐쉬로부터 긴 명령어를 인출하여 긴 명령어를 구성하는 단위 명령어별로 명령어를 디코딩하여 필요한 피 연산자를 가져온 다음 지정된 연산처리기에 할당한다. 각각의 연산처리기들은 할당받은 단위 명령어를 독립적으로 실행한다. 이때 각각의 연산처리기가 필요로 하는 피 연산자는 공유하는 다중 포트 레지스터 파일을 참조하여 가져오게 된다. 따라서 자료의존관계가 있는, 서로 다른 연산처리기에 할당되어 수행되어야 하는 한 쌍의 명령어들은 다중포트 레지스터 파일을 통하여 필요한 정보를 교환하여야 한다.

예를 들어, 연산처리기 F_1와 F_2에서 수행되는 명령어 I_1과 I_2간에 자료의존관계가 존재한다면 F_2는 F_1이 명령어 I_1의 연산이 종료된 다음 I_1의 결과가 레지스터 파일에 기록된 후에야 F_2가 I_2를 실행해야 하므로 I_2의 연산은 결국 I_1의 연산이 종료된 후 한 사이클이 더 진행된 후에야 비로소 실행이 가능하다. 따라서 서로 다른 연산처리기간에 할당된 연속된 명령어들간에 자료의존관계가 존재할 경우에는 레지스터 파일을 참조하는데 필요한 지연이 발생한다. 이러한 지연은 결국 NOP으로만 구성된 긴명령어의 삽입이 필요하거나 긴명령어내에 NOP이 삽입되는 것을 피할 수 없다. 그러나 같은 연산처리기에 할당된 연속된 명령어간에 자료의존관계가 존재하는 경우에는 쓰기 단계 이전에 연산 결과를 실행 단계로 되돌려주는 bypass 기법을 이용할 수 있으므로 자료의존관계로 인한 별도의 지연이 발생하지 않는다.

```
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

```
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) VLIW 코드 예제 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) VLIW 코드 예제 2

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

(e) 명령어 복사 결과

그림 1. 컴파일러에 의한 코드의 생성

이러한 가정하에서 그림 1(a)의 프로그램을 토대로 VLIW 목적 코드가 생성되는 과정을 살펴보면 다음과 같다. 그림 1(b)는 명령어 그래프로 모드는 명령어, 화살표로 표시된 간선은 명령어 간의 자료의존관계를 나타낸다. 그림 1(c)과 1(d)는 각각 두 개의 연산처리기로 구성된 VLIW 프로세서에서 가능한 명령어 할당 결과이다. 그림 1(b)에서 I_1과 I_2간에는 자료의존관계가 존재하지 않으므로 서로 다른 연산처리기에 할당되어 하나의 긴 명령어를 구성할 수 있다. I_3과 I_4는 각각 I_1과 I_3와의 자료의존관계로 인하여 동일한 연산처리기에 할당되어야 한

* 이 논문은 한국 과학재단의 연구비 지원에 의한 것임

다. 그림 1(c)와 (d)에서는 첫 번째 연산처리기(F_1)에 할당하였다. 만일 I_4를 두 번째 연산처리기(F_2)에 할당하는 경우에는 F_1에 할당된 I_3과의 자료의존관계로 인하여 I_3가 포함된 긴명령어와 I_4가 포함된 긴명령어 사이에 NOP으로만 이루어진 긴명령어가 삽입되어야 한다. 따라서 I_4를 F_1에 할당하기로 한다.

I_5의 경우에는 I_5를 할당하는 경우에는 비록 I_4와는 자료의존관계가 없으므로 I_4가 할당된 긴명령어의 F_2에 할당될 수 있으나 I_3과의 자료의존관계로 인하여 그림 1(c)와 같이 I_4와는 별개의 명령어로 구성되어야 한다. 따라서 그림 1(a)에 보인 프로그램은 항상 4개의 긴 명령어로 구성된 목적코드를 생성하게 된다.

만일 그림 1(e)에서 밑줄 친 부분과 같이 NOP이 삽입될 부분에 I_3를 F_2에 또다시 할당한다고 가정하면, I_5를 F_2에 할당할 경우에 I_3과의 자료의존관계가 해결되므로 긴명령어의 수가 하나 적어지게 되므로 전체적으로 1 cycle 빠른 계산이 가능하다. 즉, NOP이 포함될 코드 슬롯에 다른 연산처리기가 실행할 명령어를 복사하는 경우에 전체적으로 목적코드의 길이가 짧아져 프로그램의 실행 사이클을 줄일 수 있다.

2.2 TiPS 프로세서 구조

명령어를 복사하는 경우에 기존의 VLIW 구조에서는 쓰기 단계에서 레지스터 파일의 동일한 주소를 동시에 여러 개의 연산처리기가 참조를 하게되는 문제점이 발생하게 된다. 따라서 명령어를 복사하기 위해서는 쓰기 단계에서의 레지스터 파일 충돌을 방지하는 하드웨어가 필요하다. 본 논문에서는 명령어 별로 연산의 결과를 레지스터 파일에 저장하는 쓰기 단계의 활성화 여부를 알려주는 정보를 추가하고 이 정보에 따라 쓰기 단계를 활성화하거나 비활성화할 수 있는 프로세서인 TiPS(Tiny Processor Structure)구조를 제안하였다. TiPS 구조는 그림 2와 같이 여러개의 정수처리기(IPU : Integer Processing Unit)와 레지스터 파일의 갱신 여부를 제어하는 제어회로를 포함하고 있다.

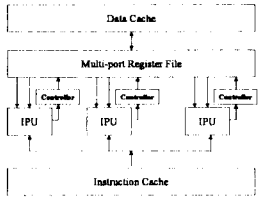


그림 2. TiPS 프로세서 구조

2.3 TiPS 명령어 형식

그림 2에 보인 TiPS 구조를 활용하기 위해서는 쓰기 단계에서 연산결과의 유효 여부를 결정하는 정보가 필요하다. 본 논문에서는 이를 위하여 명령어 별로 1bit의 정보를 추가하여 쓰기 단계에서 활성화 여부를 판단하도록 하였다. TiPS 구조를 위한 긴 명령어의 형태는 그림 3과 같다. 즉, 각 단위 명령어는

wb_en . Instruction

와 같이 기존의 명령어 *Instruction*이 쓰기 단계에서 활성화될 것인지를 결정하는 정보(*wb_en*)와 명령어 *Instruction*의 조합으로 구성된다. 여기서 *wb_en*의 값이 1이면 쓰기 단계에서 쓰기 동작이 활성화되며, *wb_en*이 0이면 비활성화된다. 그림 1의 예제에서 TiPS 구조용 목적코드의 형태는 다음과 같이 구성된다.

```
I1 : 1.lw   $14, 20($sp)    I2 : 1.lw   $8,  22($sp)
I3 : 1.mul  $15, $14, 4     I3 : 0.mul  $15, $14, 4
I5 : 1.addiu $24, $15, 1    I5 : 1.addu $25, $15, $8
```

여기서 밑줄친 명령어는 쓰기 단계에서 비활성화되는 것이 명시적으로 표현하고 있다.

TiPS 구조의 명령어가 기존의 명령어와 비교하여 *wb_en*이 추가됨으로 인하여 비록 데이터 버스의 넓이가 넓어지기는 하지만 실제로 명령어 체계는 기존의 명령어 체계를 그대로 이용할 수 있는 장점이 있다.

3. TiPS 코드 생성 기법

3.1 명령어 복사와 자료의존관계

정리 1, 2는 긴명령어에서 NOP이 차지하고 있는 슬롯에 의미 있는 명령어를 복사하는 경우에도 원래의 긴명령어가 수행되었을 때의 연산 결과와 동일한

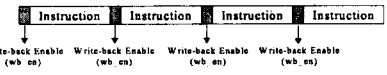


그림 3. TiPS 명령어 구조

연산 결과를 얻을 수 있음을 알려준다.

정리 1. 하나의 긴 명령어에 NOP이 아닌 어떤 명령어가 복사되어 둘 이상의 연산처리기에 의해 동시에 실행되어도 이로 인하여 새로운 자료의존관계가 생성되지 않는다.

<증명> 생략

정리 2. 하나의 긴 명령어에 NOP이 아닌 어떤 명령어가 복사되어 둘 이상의 연산처리기에 의해 동시에 실행되어도 복사하지 않은 경우와 동일한 계산 결과를 얻는다.

<증명> 생략

정리 1과 2로부터 긴명령어 내에서의 명령어 복사는 연산 결과에 영향을 미치지 않음을 알 수 있다. 따라서 주어진 프로그램으로부터 긴명령어를 생성하는 과정에서 NOP 슬롯에 명령어 복사가 연속적으로 이루어지더라도 계산 결과에는 변화가 없을 것이다.

3.2 코드 복사를 이용한 코드 생성 기법

3.1절에서 NOP을 대신하여 '의미있는 명령어'를 삽입하기 위하여 의미있는 명령어를 선정하는 과정에서 연속한 다음 명령어간의 자료의존관계를 제거할 수 있는 명령어를 선택하여 삽입한다면 결국 그림 1에서 살펴본 바와 같이 목적코드의 길이를 줄일 수 있다. 따라서 코드 복사가 가능한 TiPS에서는 NOP 슬롯을 이용한 코드 복사 기법이 매우 유용하다. 코드 복사 기법은 태스크 수준의 스케쥴링 기법[1,9]에 적용되어 왔으나 본 논문에서는 VLIW 구조의 일종인 TiPS 구조의 목적코드를 생성하는데 적용하였다.

TiPS 코드를 생성하는 과정은 다음과 같다. 첫 번째 단계에서는 명령어 그래프를 기반으로 동시에 할당이 가능한 명령어들의 그룹을 결정하여 같은 레벨로 분류한다. 각 명령어별로 레벨은 명령어 그래프에서 뿌리 노드(root node)로부터 각 명령어에 이르는 최장 경로의 길이로 결정하였다.

레벨이 결정되면 낮은 레벨에 포함된 명령어부터 긴명령어를 구성하도록 한다. 코드 생성시에는 각 명령어들이 가용한 연산처리기 중에서 어떤 연산처리기에 할당될 것인가를 결정하여야 한다. 이 때, 현재 처리중인 명령어는 모든 부모노드와의 자료의존관계가 해결되는 시점에 할당될 수 있다. 즉, 어떤 명령어 I_i를 연산처리기 F_j에 할당한다고 가정할 때, I_i 명령어의 부모노드 I_i^p가 다른 연산처리기 F_k에 할당되어 있다면 I_i^p의 수행이 끝난 후 자료전달을 위한 지연시간이 소요된 후에 I_i를 할당할 수 있다. 그런데 만일 F_j에 이미 할당되어 있는 명령어 중 수직낭비 슬롯이 존재하고, 이 슬롯에 I_i^p가 복사될 수 있다면 I_i^p를 복사하지 않는 경우보다 빠른 시점에서 수행을 시작할 수 있다. 단, F_j에 I_i^p를 복사하기 위해서는 복사 시점에서 I_i^p와 이것의 부모노드들간의 자료의존관계가 모두 해결되었다는 조건을 만족해야 한다.

이와 같이 어떤 명령어의 할당시, 모든 부모노드의 복사 여부를 검사하여 가장 빠르게 명령의 수행을 시작할 수 있는 연산처리기에 명령어를 할당하므로써 효과적인 코드를 생성할 수 있다. 그림 4는 코드 복사를 이용한 코드 생성기법을 정리한 알고리즘이다.

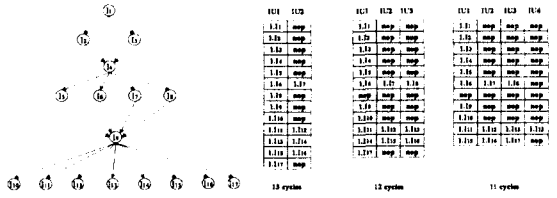
그림 5는 주어진 명령어 그래프에 대하여 IPU의 수가 2~4개일 경우 기존의 코드 생성기법과 본 논문에서 제안한 코드 복사 기법을 이용하여 생성한 코드를 비교한 것이다. 그림 5(c)에서 음영으로 표시한 부분은 긴명령어에서 복사된 명령어들을 나타내며, 이들의 쓰기 가능(*wb_en*) 비트가 0으로 설정되어 있다. 그림 5에서 복사기법이 기존의 기법에 비하여 긴명령어의 길이가 줄어든 것을 알 수 있다.

4.실험 및 고찰

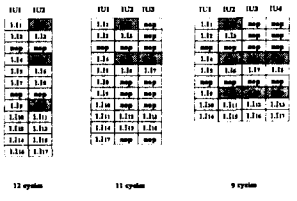
본 논문에서 제안한 코드 생성 기법의 성능을 평가하기 위하여 TiPS를 1~6개의 정수형 유니트(IPU)로 구성하는 것으로 가정하였다. 즉, 모든 명령어는 실행 사이클이 한 사이클 필요하며, 매 사이클마다 하나의 명령어가 완료되는 것으로 간주하였다. 또한, 캐시미스는 일어나지 않는 것으로 간주하였으며, 명령어 그래프는 하나의 기본 블록으로 구성되어 있는 것으로 간주하였다. 또한, 공정한 실험을 위하여 실험에서는 임의의 자료의존관계를 가지는 명령어 그래프(instruction graph)를 임의로 발생시키도록 하였다. 각 명령어 그래프의 노드 수는 각각 100개인 경우와 1,000개인 경우로 제한하였으며, 다양한 응용 프로그램

Algorithm : Code Generation
Input : Instruction Graph (basic block)
Output : VLIW object code
Begin
  For every instruction I_i
    Set Level(i) as their critical-path length
  EndFor
  For every instruction I_i in ascending order of Level(i)
    S_i - set of parent nodes of I_i
    For every functional unit IPU_j
      t_j^c - time at which I_i could assigned under assumption that every node in S_i communicated through register files
      t_j^d - time at wnich I_i could assigned under assumption that some node of S_i can be duplicated to IPU_j
    EndFor
    $T_i = \min(t_j^c, t_j^d)$ for all IPU_j
    IPU_p - IPU which satisfying T_i
    If $T_i = t_p^d$
      Assign available node of S_i to IPU_p
      Assign I_i to IPU_p
    Else
      Assign I_i to IPU_p
    EndIf
  EndFor
End

그림 4. 복사 기법 알고리즘



(a) 명령어 그래프 (b) 기존 기법



(c) 복사 기법

그림 5. 코드 복사에 의한 코드 생성 예제

을 모의하기 위하여 명령어 그래프에 포함된 간선의 수를 변화하면서 실험하였다. 즉, 명령어 그래프가 완전연결형(fully-connected)인 경우의 간선의 수를 100%라고 하였을 때에 간선의 수를 20%~50%까지 10%간격으로 변화시키면서 각 경우별로 50개씩의 명령어 그래프를 생성하여 실험하였다.

그림 6과 7은 각각 100개와 1,000개로 구성된 명령어 그래프에 대하여 기존의 VLIW 목적 코드 생성 기법(GCG : General Code Generation)과 본 논문에서 제안하는 코드 생성기법(DCG: Duplication Code Generation)을 사용하였을 경우에 생성된 목적코드의 길이를 비교한 것이다. 그림 6(a)~(d)는 그래프 생성시 간선의 비율이 각각 20%, 30%, 40% 및 50%인 경우에 실험한 결과이다. 그림에서 그래프의 x축은 TiPS를 구성하는 정수형 유니트의 수를 나타내며 y축은 50개의 그래프에 대한 실행 사이클의 평균치이다. 또한 각 그래프에서 막대 그래프는 50개의 명령이 그래프 별로 실험한 결과의 표준편차를 나타낸 것이다. 그림에서 알 수 있듯이 DCG 기법이 GCG 기법에 비하여 목적코드의 길이가 짧아지 전체 수행사이클이 단축시킬 수 있음을 보여준다. 복사 기법의 효과는 명령이 그래프의 간선의 비율이 클수록 뚜렷해지는 것도 보여준다.

5. 결론

본 논문에서는 VLIW 목적코드에서 많은 부분을 차지하는 NOP 슬롯에 의미있는 코드를 복사하여 할당함으로써 낭비된 코드 슬롯을 활용하여 프로그램의 실행 사이클을 단축시키는 명령어 복사 기법을 제안하였다. 제안된 기법에 의하여 생성되는 목적 코드에서는 하나의 긴 명령어내에 동일한 명령어가 둘 이상 포함될 수 있으므로 명령어 파이프라인의 쓰기 단계에서 레지스터 파일의 충돌을 야기하게 되므로 이를 방지할 수 있는 TiPS 구조도 제안하였다. TiPS 구조에서는 여러 개의 연산처리기가 같은 위치의 레지스터 파일 주소에 쓰기 동작을 수행할 경우, 이들 중 지정된 명령어가 수행되는 연산처리기의 연산 결과만이 쓰기 동작이 활성화 되도록 한다.

제안된 기법에 의하여 생성되는 TiPS 목적 코드에서는 NOP 슬롯에 할당된 명령어가 명령어간의 자료의존관계를 해결하므로써 목적코드의 길이가 기존의 VLIW 목적코드에 비하여 짧아짐으로써 전체적으로 프로그램의 실행 사이클이 단축된다. 여러 가지 명령어 그래프에 대한 시뮬레이션 결과도 명령어 복사 기법에 의하여 생성된 목적코드를 실행하는 경우가 기존의 기법에 의한 경우보다 프로그램의 수행시간이 단축되는 것을 확인시켜 주었다.

향후에는 TiPS 구조를 구성하는 연산처리기에 실수 유니트를 포함할 경우에 대한 연구를 계속할 계획이다.

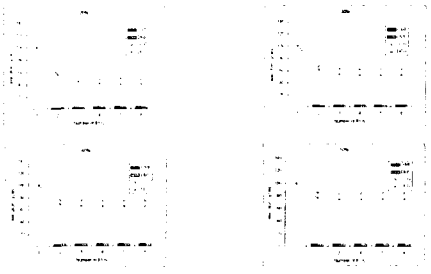


그림 6. 코드 생성 기법에 따른 실행 사이클 비교 (명령어 수 : 100개)

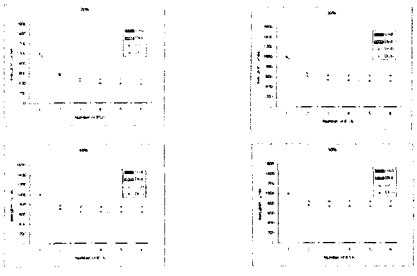


그림 7. 코드 생성 기법에 따른 실행 사이클 비교 (명령어 수 : 1,000개)

참고 문헌

[1] B. Kruatrachue and T. Lewis, "Grain Size Determination for Parallel Processing," *IEEE Software*, pp.23-32, January, 1988.
[2] 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, April 1997
[3] Sung-Hyun Jee, No-Kwang Park and Sukil Kim, "Performance analysis of caching instructions on SVLIW processor and VLIW processor." *Journal IEEE Korea Council*, Vol. 1, No. 1, December 1997
[4] Joseph A. Fisher, "Trace Scheduling: A technique for global microcode compaction,", *Trans. Comp.*, Vol. C-30, No. 7, pp. 478-490, July 1981.
[5] Monica Lam, "softwure Pipelining: An effiective scheduling technique for VLIW," PhD thesis, Canegie Mellon, May 1987.
[6] Albert Y. Zomaya, Selim G. Akl, et. al., *Parallel and Distributed Computing Handbook*, McGraw-Hill, 1996
[7] Thomas M. Conte and Sumedh W. Sathaye, "Dynamic Rescheduling: A technique for object code compatibility in VLIW architecture", *Proc, 28th Inter. Symp. Micro.*, 1995.
[8] Shyh-Kwei Chen, W. Kent Fuchs and Wen-Mei W.Hwn, "An analytical approach to scheduling code for superscalar and VLIW architectures," Proc. Inter. conf. Para. Pro., pp. 1258-1292, 1994.
[9] S. Manoharan, "Augmenting work greedy assignment schemes with task duplication," ICPADS '97, pp. 772-777, 1997.
[10] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy, "Simultaneous Multithreading: Maximizaing On-Chip Parallelism," *ISCA '95*, pp. 392-403, 1995.