ILP 프로세서를 위한 조건실행 지원 스케줄링 알고리듬

유 병 강† · 이 상 정††

요 약

명령어 수준에서 병렬성(Instruction-Level Parallelism, ILP)을 추출하는 것은 슈퍼스칼라 및 VLIW프로세서들의 성능 개선을 위한 효과적인 메커니즘이다. 이를 위하여 여러가지 소프트웨어 기법들이 응용될 수 있다. 이들 기법 중 조건실행(predicated execution)은 명령어의 조건으로 참조되는 부울 소스 오퍼랜드의 값을 기본으로 명령의 조건적 실행 여부를 참조하여 분기명령을 제거함으로서 여러 기본블록의 명령들을 하나의 기본블록으로 구성하여 ILP를 증가시키는 기법이다.

본 논문은 조건실행을 지원하는 ILP프로세서들의 성능 개선을 위하여 기본블록을 넘어선 광역 조건실행 지원 스케줄링 알고리듬(global predicate-sensitive scheduling algorithm)을 제안한다. 또한 C 컴파일러와 시뮬레이터를 개발하고 다양한 벤치마크 프로그램에 대하여 제안된 알고리듬의 성능을 측정하고 타당성을 확인한다. 1, 2, 4 이슈실행에 대한 성능 측정 결과, 평균 20%의 성능 개선이 확인되었다.

A Predicate-Sensitive Scheduling Algorithm in Instruction-Level Parallelism Processors

Byung Kang Yoo† · Sang Jeong Lee††

ABSTRACT

Exploitation of instruction-level parallelism(ILP) is an effective mechanism for improving the performance of modern super-scalar and VLIW processors. Various software techniques can be applied to increase ILP. Among these techniques, predicated execution is the one that increases the degree of ILP by allowing instructions from different basic blocks to be converted to a single basic block by removing branch instructions.

In this paper, a global predicate-sensitive scheduling algorithm is proposed to improve the performance for ILP processors that support predicated execution. In order to examine the performance of proposed algorithm, a C compiler and a simulator are developed. By simulating various benchmark programs with the compiler and the simulator, the performance results of this algorithm are measured and the effectiveness of the algorithm is verified. As a result of measure performance with 1, 2, 4 issue execution, this study was confirmed average performance by 20% or more.

1. 서 론

명령어 수준에서 병렬성(Instruction-Level Parallelism, ILP)을 이용하는 슈퍼스칼라와 VLIW 프로세서들은 불충분한 ILP가 나타날 경우 프로세서들의 잠재적 성능은 극적인 타격을 입는다. ILP를 제한하는 대표적인 요인은 분기명령이다. 분기명령은 기본블록

† 종신회원:순천향대학교 일반대학원 박사과정 수료
†† 정 회 원:순천향대학교 컴퓨터학부 부교수
논문접수:1997년 10월 6일, 심사완료:1997년 11월 10일

(basic block)내에서 ILP의 양을 제한한다. 또한 조건 분기명령인 경우에는 조건의 결과 판정이 지연되어 지속적인 파이프라인 처리를 방해한다. 따라서 분기명령으로 인한 ILP성능의 제한을 극복하고자 하는 많은 연구가 진행되고 있다[3][4][5][6][9][16][17]. 이들 연구 중 분기예측(branch predictuion)은 미리 분기의 조건을 예측하여 지속적인 파이프라인 처리를 보장하는 방법이다[9][17][20]. 이 방식은 분기예측이 잘못되었을 경우 성능손실이 크다는 단점이 있다.

또 다른 방식은 ILP 향상의 주요한 제한 중 하나인 분기명령을 제거하는 조건실행(predicated execution) 기법이 있다[3][4][6][9][18][19][22]. 이 방식은 명령어의 조건으로 참조되는 부울 소스 오퍼랜드의 값을 기본으로 명령의 조건적 실행 여부를 참조하여 분기명령을 제거함으로서 여러 기본블록의 명령들을 하나의 기본블록으로 구성하여 ILP를 증가시키는 기법이다.

본 논문은 조건실행을 지원하는 ILP 프로세서들의 성능 개선을 위하여 기본블록을 넘어선 광역 조건실행 지원 스케쥴링 알고리듬(global predicate-sensitive scheduling algorithm)을 제안한다. 즉 최적화 컴파일러가 조건실행을 지원하는 ILP 프로세서에 대하여 명령어들을 스케쥴할 때 조건실행을 고려하여 스케쥴하는 알고리듬을 개발하여 제안한다. 제안된 알고리듬은 컴파일러가 생성한 중간언어를 입력으로 받아 데이터 흐름 분석(data flow analysis)을 수행한 후 조건 분기를 제거하는 IF-변환(IF-conversion)을 수행한다[1][3][4][18]. IF-변환 수행 후 변환된 중간언어의 조건(predicates)을 나타내는 조건 오퍼랜드를 분석하여 이들 조건 오퍼랜드 간의 포함관계를 분석한 후에 슈퍼블록(superblock)을 구성한다. 슈퍼블록은 프로그램의 흐름그래프 상에서 자주 수행되는 경로상에서의 집합으로 광역 스케쥴링이 적용되는 단위이다[3][5][10][18][22]. 스케쥴링 알고리듬은 슈퍼블럭의 명령어들을 입력으로 받아 데이터 종속 그래프(Data Dependency Graph, DDG)를 구성한다. DDG 구성 시 조건 오퍼랜드 간에 서로 포함관계가 없는 상호 배타적인 명령어(disjoint instructions)들은 연산 오퍼랜드 간에 데이터 종속관계가 있더라도 서로 동시에 실행되지 않으므로 DDG 상에서 데이터 종속관계를 표시하지 않으므로써 스케쥴 시 병렬처리 될 수 있도록 한다. DDG 구성 후에 스케쥴링을 적용한다. 적용되는 스케쥴링 알고리듬은 조건실행으로 이동의 가능성이 많은 상호배타적 명령어들을 최대한 위로 이동함으로써 이후의 명령어들의 스케쥴 가능성을 높인다.

또한 C 컴파일러와 시뮬레이터를 개발하고 다양한 벤치마크 프로그램에 대하여 제안된 알고리듬의 성능을 측정하고 타당성을 확인한다.

2. 조건실행

조건실행은 조건 표현식의 값에 따라 조건적으로 실행되어지는 프로그램 구조를 표현하기 위하여 조건실행 명령들(predicated instructions)을 이용한다. 조건부 명령은 조건실행 오퍼랜드를 첨가한 일반적인 명령이다. 조건실행 명령의 의미는 조건실행 오퍼랜드의 값을 평가하여 만약 조건이 참(true)이면 그 명령을 실행하고 그렇지 않으면(false) 그 명령을 NOP (no-operation)로 처리한다. 다음과 같은 명령의 경우를 고려하자.

```
SUB     dst, src, opr IF p_cond
```

위 명령에서 만약 p_cond의 조건이 참이면 src에서 opr을 뺀 결과를 dst에 저장하고 만약 p_cond가 거짓이면 SUB 명령은 NOP로 처리되어 프로그램 실행에 아무 영향을 주지 않는다. 즉 조건 실행은 조건 분기(conditional branch)를 이용하지 않고도 조건적으로 실행되는 프로그램의 구조를 표현할 수 있으며 조건실행 기술자는 올바른 결과만이 프로세서의 상태에 영향을 미친다.

```
Low = 0;
High = N - 1;
Mid = (Low + High) / 2;
while (Low <= High && Search != Data[Mid]) {
    if (Search < Data[Mid]) High = Mid - 1;
    else Low = Mid + 1;
    Mid = (Low + High) / 2;
}
if (Search == Data[Mid]) return(Mid);
else return(-1);
```

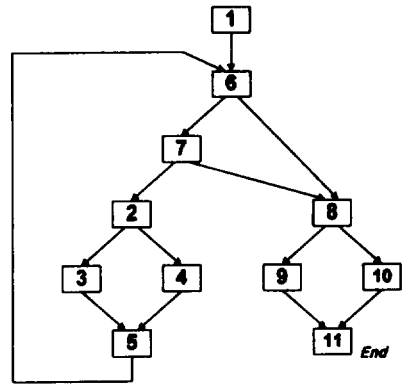
(그림 2.1) 예제 C 소스 프로그램
(Fig. 2.1) An example C source program

컴파일러에 의해서 이용되는 일반적인 프로그램의 표현 형태는 제어 흐름 그래프(Control Flow Graph, CFG)이다. 예제로 사용되는 C 소스 프로그램은 (그림 2.1)이고 이에 대한 어셈블리 코드는 (그림 2.2)에 나타나 있다.

```
                ;function prolog
                ...
                ;.maxoffset=0
[1]    1:       MOV     $35, #0
       2:       STR     $35,Low
       3:       LDR     $5,N
       4:       SUB     $6,$5, #1
       5:       STR     $6,High
       6:       LDR     $7,Low
       7:       LDR     $8,High
       8:       ADD     $9,$7,$8
       9:       DIV     $10,$9, #2
      10:       STR     $10,Mid
      11:       JUMP    L.3
[2]   12:L.2:
      13:       LDR     $11,Search
      14:       LDR     $12,Mid
      15:       SHLA    $13,$12, #2
      16:       LDR     $14,Data($13)
      17:       CMPGE   $36,$11,$14
      18:       JT      $36,L.5
[3]   19:       LDR     $15,Mid
      20:       SUB     $16,$15, #1
      21:       STR     $16,High
      22:       JUMP    L.6
[4]   23:L.5:
      24:       LDR     $17,Mid
      25:       ADD     $18,$17, #1
      26:       STR     $18,Low
[5]   27:L.6:
      28:       LDR     $19,Low
      29:       LDR     $20,High
      30:       ADD     $21,$19,$20
      31:       DIV     $22,$21, #2
      32:       STR     $22,Mid
[6]   33:L.3:
      34:       LDR     $23,Low
      35:       LDR     $24,High
      36:       CMPGT   $37,$23,$24
      37:       JT      $37,L.7
[7]   38:       LDR     $25,Search
      39:       LDR     $26,Mid
      40:       SHLA    $27,$26, #2
      41:       LDR     $28,Data($27)
      42:       CMPNE   $38,$25,$28
      43:       JT      $38,L.2
[8]   44:L.7:
      45:       LDR     $29,Search
      46:       LDR     $30,Mid
      47:       LDR     $32,Data($31)
      49:       CMPNE   $39,$29,$32
      50:       JT      $39,L.8
[9]   51:       LDR     $33,Mid
      52:       MOV     _ret,$33
      53:       JUMP    L.1
[10]  54:L.8:
      55:       MOV     _ret, #-1
[11]  56:L.1:
                ;function epilog
```

(그림 2.2) 예제의 어셈블리 코드 부분
(Fig. 2.2) An assembly code of an example

(그림 2.2)의 좌측 숫자는 기본블록의 순열을 나타내며 그 다음은 라인 번호를 나타낸다. 예제에 대한 CFG는 (그림 2.3)에 표현된 것과 같이 각 노드가 하나의 기본블록이며 각각의 에지가 그 노드에서 가능한 제어흐름과 상응함을 나타내는 방향성 그래프이다.



(그림 2.3) 예제 어셈블리 코드의 제어흐름도
(Fig. 2.3) A control flow graph of a example assembly code

한 프로그램의 컴파일 동작은 명령들이 재정렬되어지는 CFG상의 경유 경로로서 생각될 수 있다. 코드 생성동안 CFG상의 노드들은 프로그램 메모리에 선형적으로 위치되어 진다. 한 프로그램의 실행은 프로세서가 실행되어져야 한다고 결정한 명령들인 CFG상의 경유 경로로서 생각할 수 있다. CFG상에서 프로그램의 조건적 구조는 제어 종속(control dependency)관계를 일으킨다. 즉 CFG상의 노드 X가 CFG상의 노드 Y와 제어 의존관계에 있다고 할 때에 첫째로 노드 Y가 두 개 또는 그 이상 외부로의 연결선을 갖으며 둘째로 노드 Y에 이르는 제어 흐름이 노드 Y에서 선택된 조건에 따라 노드 X를 실행할 수도 있고

아닐 수도 있다는 것을 말한다.

만약 명령집합이 조건 실행을 지원한다면 IF-변환(IF-conversion)이라고 알려진 처리를 통하여 나중에 설명되는 (그림 2.4)와 같은 조건부 명령을 통하여 CFG상의 조건적 실행 부분을 표현할 수 있다. IF-변환에서 조건적으로 실행되는 부분을 표현하는 일반적인 단계는 첫째로 각각의 조건적으로 실행되는 노드에 조건 레지스터를 할당하고 둘째로 조건 레지스터를 할당하고 둘째로 조건 레지스터를 각각의 노드에 따라 어떻게 설정(set)할 것인가를 고려한다. 마지막으로 IF-변환은 조건적으로 실행되는 노드의 모든 명령들을 상응하는 조건 레지스터를 포함한 형태의 조건명령들로 변환한다. 위의 단계를 거치면 제어종속 관계는 데이터종속 관계로 변환된다.

조건실행의 지원으로 (그림 2.2)의 원래 코드는 (그림 2.4)의 IF-변환 처리 후의 코드처럼(그림자 부분)

```
        ;function prolog
        ...
        ;.maxoffset=0
        MOV     $35, #0
        STR     $35,Low
        LDR     $5,N
        SUB     $6,$5, #1
        STR     $6,High
        LDR     $7,Low
        LDR     $8,High
        ADD     $9,$7,$8
        DIV     $10,$9, #2
        STR     $10,Mid
        JUMP    L.3
L.2:
        LDR     $11,[illegible]
        LDR     $13,[illegible]
        SHLA    $12,$13, #2
        LDR     $14,[illegible](13)
        CMPGE   $50,$11,$14
        CMPLT   $51,$11,$14
        LDR     $15,Mid IF $51
        SUB     $16,$15, #1 IF $51
        STR     $16,High IF $51
;
        LDR     $17,Mid IF $50
        ADD     $18,$17, #1 IF $50
        STR     $18,Low IF $50
;
        LDR     $19,Low IF $50 OR $51
        LDR     $20,High IF $50 OR $51
        ADD     $21,$19,$20 IF $50 OR $51
        DIV     $22,$21, #2 IF $50 OR $51
        STR     $22,Mid IF $50 OR $51
```

```
L.3:
        LDR     $23,Low
        LDR     $24,High
        CMPGT   $37,$23,$24
        JT      $37,L.7
        LDR     $25,Search
        LDR     $26,Mid
        SHLA    $27,$26, #2
        LDR     $28,Data($27)
        CMPNE   $38,$25,$28
        JT      $38,L.2
L.7:
        LDR     $29,Search
        LDR     $30,Mid
        SHLA    $31,$30, #2
        LDR     $32,Data($31)
        CMPNE   $39,$29,$32
        JT      $39,L.8
        LDR     $33,Mid
        MOV     _ret,$33
        JUMP    L.1
L.8:
        MOV     _ret, #-1
L.1:
        ;function epilog
```

(그림 2.4) IF-변환된 어셈블리 코드 부분
(Fig. 2.4) IF-converted assembly code pant

if-then-else에 해당하는 조건 분기가 제거되었으며 이로 인하여 ILP 처리를 위한 기본블록이 이전보다 상당히 커졌음을 알 수 있다. 즉 ILP가 조건실행을 통하여 대단히 증가한다.

(그림 2.4)의 코드는 조건 오퍼랜드가 정적 단일지정문(static single statement) 형식으로 정의된 중간언어의 코드이다. 정의된 중간코드는 RISC 명령어 형식의 3 주소 코드(3-address code)에 조건 오퍼랜드를 부가하였다[24]. 예를들어 (그림 2.2)와 (그림 2.4)등에 표현된 어셈블리 코드중 CMPGE 명령의 중간언어 표현 형식은 다음과 같다.

op부분에는 EQ, NE, LT, LE, GT 및 GE의 하나가 응용되며 type에는 unsigned, float 및 double중 하나를 지정한다. 이 오퍼레이션의 의미는 operand2와 operand3에 대한 op의 연산 결과가 참이면 operand1에 1의 값이 그렇지 않으면 0의 값이 설정된다.

3. 스케쥴링

조건실행을 지원하는 ILP 프로세서들의 성능 개선을 위하여 기본블록을 넘어선 광역 조건실행 지원 알

고리듬(global predicatesensitive scheduling algorithm)은 다음과 같은 과정으로 수행된다.

- 조건실행 데이터 흐름분석
- 슈퍼블록 구성
- 스케쥴링

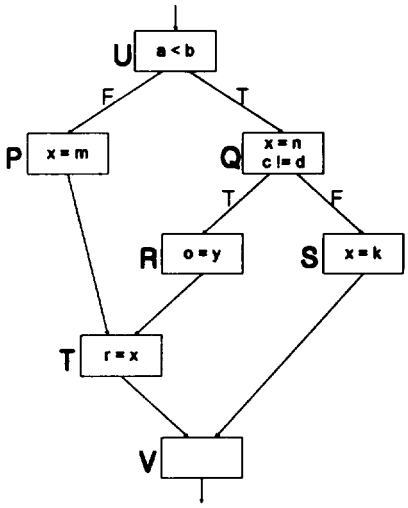
3.1 조건실행 데이터 흐름분석

조건실행 데이터 흐름분석(predicate data flow analysis) 단계에서는 컴파일러가 생성한 중간언어를 입력으로 받아 데이터 흐름분석(data flow analysis)을 수행한 후 조건 분기문을 제거하는 IF-변환(IF-conversion)을 하고 조건 오퍼랜드의 포함관계 분석을 수행한다.

IF-변환은 제어 종속(control dependent)을 데이터 종속(data dependent)으로 변환하는 과정으로 조건실행을 사용하여 제어 흐름을 제거한다. IF-변환과정은 먼저 선택된 기본 블록들에 따라서 국지적인 제어종속 정보를 계산한다. 일단 제어종속 정보가 계산되면, 하나의 조건 레지스터가 제어종속들의 각각의 단일 집합을 나타내기 위하여 할당된다. 그러므로 종속들의 일반적인 집합을 공유하는 모든 블록들은 동일한 조건하에서 실행되어 질 것이다. 조건 비교 명령들은 특별한 조건에 관련된 종속제어 에지들의 소스인 모든 기본블록들내에 삽입된다. 조건부 비교 조건은 특별한 제어종속 에지에 의해 표시된 분기 조건에 따라서 결정된다. 조건부 비교 명령들이 삽입된 후에는 새롭게 삽입된 조건부 비교 명령을 포함한 각 선택된 블록내의 모든 명령들은 그들 블록에 할당된 조건에서 조건화 된다. 최후에 선택된 블록들로부터 다른 선택된 블록들 까지의 모든 조건 분기와 무조건 분기들은 제거된다.

(그림 3.1)은 임의의 프로그램에 대한 CFG 예이고 (그림 3.2)는 IF-변환 전후의 코드를 보여주는 예이다. (그림 3.2(b))에 나타난 바와 같이 IF-변환 수행 후에는 4개의 분기명령이 제거되어 7개의 기본블록이 2개의 블록으로 통합됨을 알 수 있다.

IF-변환 수행 후 변환된 중간언어의 조건(predicates)을 나타내는 조건 오퍼랜드를 분석하여 이들 조건 오퍼랜드 간의 포함관계를 분석한다. 조건 오퍼랜드 간의 포함관계는 조건실행 분할그래프(predicate partition graph)에 근거하여 계산한다[3][18]. 즉 IF-변환되어 구해진 조건 오퍼랜드를 노드로하고, 에지 (p, q)는 q가 p에 포함되는 관계를 표시한다. 즉 포함관계가 있는 두 조건 실행은 함께 참이 되는 가능성이 존재하여 서로 배타적으로 실행되지 않는 관계를 나타낸다.



(그림 3.1) 제어 흐름 그래프의 예
(Fig. 3.1) An example of CFG

```
U : CMPGE   p, a, b
    JT   p, Q
P : MOV   x, m
    JUMP   T
Q : MOV   x, n
    CMPNE   q, c, d
    JT   q, R
S : MOV   x, k
    JUMP   V
R : MOV   o, y
T : MOV   r, x
V : ...
```

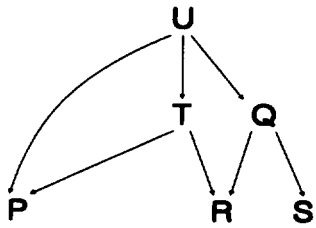
(a)

```
U : CMPGE   p, a, b
    CMPLT   q, a, b
    MOV   x , m   IF   p
    MOV   x , n   IF   q
    CMPNE   r, c, d   IF q
    CMPEQ   s, c, d   IF q
    OR   t, p ,r   IF q
    MOV   x, k   IF s
    MOV   o, y   IF r
    MOV r, x   IF t
V : ...
```

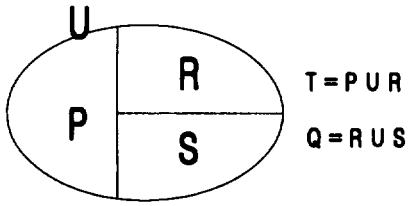
(b)

(그림 3.2) (a) 일반적인 순차코드 (b) IF-변환 코드
(Fig. 3.2) (a) Sequential code (b) IF-conversion

(그림 3.3)은 (그림 3.2)의 IF-변환 코드에 대한 조건실행 분할그래프와 포함관계를 나타낸 그림이다. 분할그래프에서 파생된 에지의 노드는 서로 포함관계를 갖는다. 즉 그림 예에서 q로 부터 r, s가 파생되었으므로 r, s는 q에 포함되고 q=rUs 관계가 성립된다. 또한 p와 q는 서로 포함관계가 없으므로 서로 동시에 참이 되어 실행되는 일이 없다. 따라서 변수 x에 대하여 데이터 종속관계가 있더라도 같이 스케쥴 할 수 있게 된다.



(a) 조건실행 분할 그래프



(b) 서술 영역간의 포함관계

(그림 3.3) (그림 3.2)의 IF-변환 코드에 대한 (a) 조건실행 분할 그래프와 (b) 포함관계

(Fig. 3.3) (a) A partition graph for the example and (b) Domains of predicates in (Fig 3.2)

3.2 슈퍼블록 구성

슈퍼블록은 프로그램의 흐름그래프 상에서 자주 수행되는 경로 상에서의 집합으로 광역 스케쥴링이 적용되는 단위이다[3][4][5][18]. 코드 최적화 및 스케쥴링의 목적은 프로그램의 의미(semantics)를 유지하면서 실행시간을 최소화하는 것인데 이것이 전역적으로 이루어 질 때 어떤 부분에 대한 최적화와 스케쥴링의 결정은 다른 부분의 실행 시간이 증가하는 반면 선택된 하나의 제어 경로에 대한 실행 시간을 감소 시킬 수 있다. 좀 더 자주 실행되는 경로를 선택하므로서 전체적인 성능 향상을 얻을 수 있다.

슈퍼블록들은 두 단계로 구성되어 지는데 첫번째 단계에서 추적들은 실행 프로파일 정보를 사용하여 판정되며, 두번째 단계에서 후부복제(tail duplication)라고 하는 처리가 추적에 대한 임의의 진입점들을 소거하기 위하여 실행된다. 복제(copy)는 첫번 진입점에서 끝 진입점에 이르는 추적의 후위 부분을 만든다. 그러면 추적의 모든 진입점은 상응하는 복제 기본블록들로 이동된다. 슈퍼블록내의 기본블록들은 코드내에서 연속적일 필요가 없다. 그러나 구현은 슈퍼블록내의 모든 블록들이 더 나은 캐쉬 성능을 위하여 연속적인 순서를 갖도록 하기 위하여 코드를 재구성한다.

3.3 스케쥴링

스케쥴링 알고리듬은 슈퍼블럭의 명령어들을 입력으로 받아 데이터 종속 그래프(Data Dependency Graph, DDG)를 구성한다. DDG는 슈퍼블록 내의 각 명령어에 대하여 데이터 종속관계를 조사하여 각 명령어가 노드가 되고 각 에지가 데이터 종속관계를 나타내는 비순환형 그래프이다. 데이터 종속관계는 각 오퍼레이션의 소스 레지스터와 목적 레지스터를 조사하여 같은 레지스터인 경우 발생한다. DDG 구성시 조건 오퍼랜드 간에 서로 포함관계가 없는 상호 배타적인 명령어(disjoint instructions)들은 연산 오퍼랜드 간에 데이터 종속관계가 있더라도 서로 동시에 실행되지 않으므로 DDG 상에서 데이터 종속관계를 표시하지 않으므로서 스케쥴 시 병렬처리 될 수 있도록 한다.

DDG를 구성한 후 각 노드에 대하여 자손노드의 수를 세어서 자손노드의 수가 많은 순서로 각 명령어에 대하여 스케쥴링 우선순위를 부여한다. 자손노드의 수가 같으면 소스 코드상에서 먼저 기술된 순서대로 우선순위를 부여한다. 우선순위 부여 후에는 우선순위가 높은 오퍼레이션 부터 스케쥴될 명령어로 선택하고 각 명령어의 데이터 종속관계 및 자원충돌 관계를 조사하여 (그림 3), (그림 4)의 알고리듬을 적용하여 스케쥴한다. 적용되는 스케쥴링 알고리듬은 조건실행으로 이동의 가능성이 많은 상호 배타적 명령어들을 최대한 위로 이동함으로써 이후의 명령어들의 스케쥴 가능성을 높이기 위해 기존 리스트 스케쥴

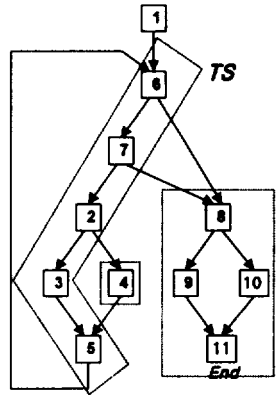
링(list scheduling) 알고리듬을 보완 개선한 알고리듬이다[10][14][25]. 즉 현재 스케쥴 될 명령어와 이미 스케쥴된 명령어들과 데이터 종속관계가 없는 경우에는 바로 스케쥴하지 않고 최대한 위로 이동한다.

```
predicated-sensitive-schedule()
{
S = 슈퍼블록의 오퍼레이션 집합;
last_cycle = 0;

  while (집합S가 공집합이 아니라면) {
   op = 집합S의 가장 높은 순위 OP;
   S = S - {op};
   cycle = last_cycle;

   while (cycle이 0이 아니면서
         op가 INST[cycle]의 OP들과
         데이터 종속관계가 없으면) {
    --cycle;
/*스케쥴될 가장 빠른 클럭사이클 결정*/
   }

   ++cycle;

   while (cycle != last_cycle+1) {
/*최후 스케쥴 명령어 도달 때 까지*/
    if (op 가 INST[cycle]의 OP와
       자원충돌 관계가 없으면) {
     INST[cycle] = INST[cycle] + {op};
/*스케쥴*/
      break;
    }else –cycle;
   }

   if (cycle == last_cycle + 1) {
/*OP가스케쥴된 명령어와 병렬처리못하면*/
     INST[cycle] = {op};
     last_cycle = cycle;
   }

  }

}
```

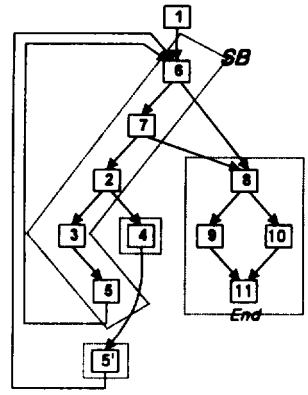
(그림 3.4) 조건실행 지원 알고리듬
(Fig. 3.4) A predicate-sensitive scheduling algorithm

4. 적용 예

본 논문에서 제안되는 알고리듬에 대한 설명에 사용되는 프로그램은 (그림 2.1)의 이진탐색을 위한 C 소스 프로그램이다. C 컴파일러에 의해 생성된 어셈블리 코드의 실제 처리 부분은 (그림 2.2)에 나타내었다. 이 코드를 기본블록들로 구분하여 제어흐름도를 나타낸 것이 (그림 2.3)이다. (그림 4.1)은 예제 프로그램의 TS부분에 대한 슈퍼블록형성을 나타낸 그림이다.



(a) after trace selection



(b) after tail duplication

(그림 4.1) 슈퍼블록 형성절차
(Fig. 4.1) Superblock formation

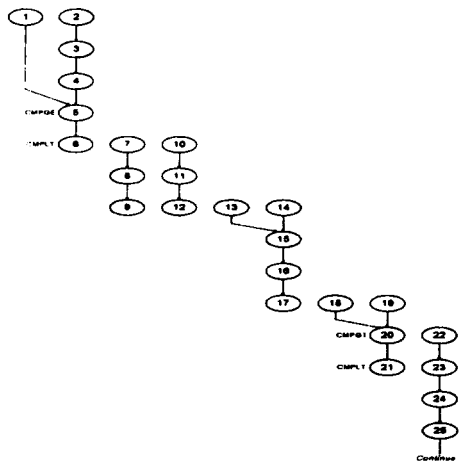
(그림 4.2)는 IF-변환된 후의 스케쥴링 대상이 되는 슈퍼블록에 대한 코드 부분이다. (그림 4.3)은 (그림

4.2)에 대한 데이터 종속 그래프를 나타낸다. 그림에서 점선은 병렬로 처리가 가능한 명령들을 나타내며 화살표 실선은 종속의 흐름을 나타내고 이중 실선은 IF-변환되어 조건 오퍼랜드가 고려되는 동일한 레벨을 나타낸다.

```
    L.2:
 1:         LDR     $11,Search
 2:         LDR     $12,Mid
 3:         SHLA    $13,$12, #2
 4:         LDR     $14,Data($13)
 5:         CMPGE   $50,$11,$14
 6:         CMPLT   $51,$11,$14
 7:         LDR     $15,Mid IF $51
 8:         SUB     $16,$15, #1 IF $51
 9:         STR     $16,High IF $51
    ;L.5:
10:         LDR     $17,Mid IF $50
11:         ADD     $18,$17,#1 IF $50
12:         STR     $18,Low IF $50
    ;L.6:
13:         LDR     $19,Low IF $50 OR $51
14:         LDR     $20,High IF $50 OR $51
15:         ADD     $21,$19,$20 IF $50 OR $51
16:         DIV     $22,$21, #2 IF $50 OR $51
17:         STR     $22,Mid IF $50 OR $51
    L.3:
18:         LDR     $23,Low
19:         LDR     $24,High
20:         CMPGT   $53,$23,$24
21:         CMPLT   $54,$23,$24
22:         LDR     $25,Search IF $54
23:         LDR     $26,Mid IF $54
24:         SHLA    $27,$26, #2 IF $54
25:         LDR     $28,Data($27) IF $54
26:         CMPNE   $55,$25,$28
27:         CMPEQ   $56,$25,$28
            ...                 ...
```

(그림 4.2) 스케쥴링 대상이 되는 코드부분
(Fig. 4.2) Scheduling code parts

(그림 4.3)의 노드 1, 2, 3, 4, 5 및 6에서 노드 5는 노드 1에 종속인 동시에 노드 2, 3, 4에도 종속적이다. 한편 노드 1과 노드 2는 비종속 관계이므로 첫번째 스케쥴링 단계에서 동시에 병렬처리 되도록 놓여질 수 있다. 그러나 노드 2에 더 많은 자손 노드가 존재하므로 스케쥴링 우선순위가 먼저이다. 다음 노드 1과 노드 3에서 노드 3이 자손 노드의 수가 더 많으므로 노드 3의 스케쥴링 우선순위가 먼저이다. 그 다음에 노드 1과 노드 4의 경우에 두 노드가 모두 자손 노드를 한 개씩 갖으나 노드 4가 노드 3에 종속적임을 고려하여 먼저 스케쥴링된다.



(그림 4.3) 데이터 종속 그래프
(Fig. 4.3) Data dependency graphs

한편 노드 5와 노드 6은 IF-변환된 부분으로 주어진 오퍼랜드에 대한 조건 오퍼랜드 값을 계산하는 부분으로 동일한 우선순위를 갖으며 노드 10과 같이 종속에 독립적인 것들은 병렬처리 가능한 최상위 위치에 배치된다. (그림 4.3)은 (그림 4.2)에 대해 병렬 파이프라인 아키텍처가 한 사이클 당 두개 이상의 오퍼레이션을 동시에 수행한다고 가정할 때 이루어지는 스케쥴링 과정을 나타낸다.

스케쥴이 완료되면 (그림 4.4)의 우측 부분의 순서와 같이 코드의 배치 순서가 결정된다. 아래의 (그림 4.5)는 예제 프로그램에 대하여 스케쥴링된 어셈블리 코드를 나타낸다.

Data Ready Set	Instructions	
	I	II
1 2	2	1
1 3	3	

1 4	4	
1 5	5	6
5	10	7
6	11	8
7	12	9
8	13	18
9	14	19
10	15	20
11	16	21
12	17	22
13 14		23
14		24
15		25
16	27	26
17		
18		
19		
20		
21		
22		
23		
24		
25		
26		
27		

(그림 4.4) 스케쥴링 과정
(Fig. 4.4) Scheduling

```
2:    LDR     $12,Mid
1:    LDR     $11,Search
3:    SHLA    $13,$12, #2
4:    LDR     $14,Data($13)
5:    CMPGE   $50,$11,$14
6:    CMPLT   $51,$11,$14
10:   LDR     $17,Mid IF $50
7:    LDR     $15,Mid IF $51
11:   ADD     $18,$17, #1 IF $50
8:    SUB     $16,$15, #1 IF $51
12:   STR     $18,Low IF $50
9:    STR     $16,High IF $51
13:   LDR     $19,Low IF $50 OR $51
18:   LDR     $23,Low
14:   LDR     $20,High IF $50 OR $51
19:   LDR     $24,High
15:   ADD     $21,$19,$20 IF $50 OR $51
20:   CMPGT   $53,$23,$24
16:   DIV     $22,$21, #2 IF $50 OR $51
21:   CMPLT   $54,$23,$24
17:   STR     $22,Mid IF $50 OR $51
22:   LDR     $25,Search IF $54
23:   LDR     $26,Mid IF $54
24:   SHLA    $27,$26, #2 IF $54
25:   LDR     $28,Data($27) IF $54
26:   CMPNE   $55,$25,$28
27:   CMPEQ   $56,$25,$28
```

(그림 4.5) 스케쥴링된 어셈블리 코드
(Fig. 4.5) A scheduled assembly code

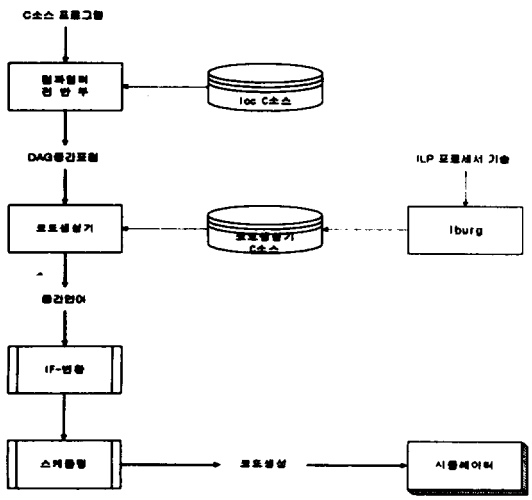
5. 시뮬레이션 및 결과분석

본 장에서는 조건반응 실행지원 알고리듬을 적용하여 ILP프로세서의 성능을 측정하고 평가하기 위하여 제작된 C 컴파일러 및 시뮬레이터에 관하여 설명한다[25]. 컴파일러는 벤치마크용 C 소스 프로그램을 입력받아 3-주소 코드형태의 중간언어를 생성한다. 생성된 중간언어는 설계 사양에 따른 적절한 매개변수와 함께 시뮬레이터의 입력 데이터로 사용되며 시뮬레이터는 성능 측정에 대한 여러가지 종합정보를 생성한다. 시뮬레이션 결과에 따라 생성된 정보(profiling 정보 등)들은 측정 목적에 맞게 중간언어를 적절히 수정하여 다시 시뮬레이터의 입력으로 사용된다. 개발된 도구들은 ILP프로세서를 위한 컴파일러 및 프로세서 설계시에 명령수준의 상위 레벨에서 설계 사양의 선택기준 및 성능검증의 수단으로 사용될 수 있다. 이 장의 중반부는 설명된 C 컴파일러 및 시뮬레이터를 사용할 수 있도록 설계로 DDG를 구성하고 스케쥴링하는 단계를 설명하며 하반부는 벤치마크 프로그램에 대한 측정값을 비교 검토함으로서 성능 개선의 효과를 판단한다.

5.1 컴파일러 및 중간언어

알고리듬의 성능측정을 위하여 개발된 C 컴파일러의 구성은 (그림 5.1)과 같다. 컴파일러의 전반부는 Fraser와 Hansen이 개발한 lcc[2]를 사용하여 C 언어로 작성된 소스 프로그램의 어휘분석 및 구문분석을 처리한다. 코드 생성기는 code-generator generator인

lburg[2]가 사용되는데, 이것은 lcc의 중간표 현인 DAG (Directed Acyclic Graph) 정보를 입력으로 하여 대상 머신의 C 프로그램을 생성한다.



(그림 5.1) C 컴파일러의 구성
(Fig. 5.1) The structure of C compiler

중간언어는 3-주소 코드의 RISC 명령어 형태를 취하면서 컴파일러가 지원하는 정적 정보를 표현할 수 있도록 머신 독립적으로 광범위하게 정의하였다. 이전의 (그림 2.1)과 (그림 2.2)는 이진탐색을 처리하는 C 프로그램의 소스와 컴파일러가 생성한 중간언어를 나타낸다. 중간언어에서 사용되는 변수에는 C 언어에서 사용되는 변수명, 변수형 및 범위 그리고 각 함수마다 운영되는 프레임상의 위치정보, 정적으로 참조되는 수 및 변수의 크기등이 나타난다. 이러한 중간언어의 표현은 코드의 디버깅과 최적화를 위한 정보를 활용할 수 있도록 고려하였으며 표기법은

[의사코드][변수명][변수형][범위][오프셋][참조][크기]

와 같은 순서로 표시된다.

5.2 시뮬레이터

본 논문의 성능 측정을 위하여 GNU의 Flex, Bison [7][8]을 사용하여 구현된 시뮬레이터는 컴파일러에 의해 생성된 중간언어와 ILP 프로세서의 환경 매개변수를 함께 입력으로 받아 시뮬레이션한다. 시뮬레이터는 수행 후의 메모리 내용, 클럭스, 명령 트레이스, 사용된 명령들의 정적빈도수, 동적 빈도수, 분기명령의 예측률 및 프로파일 정보등을 생성한다. 시뮬레이터는 2패스로 수행되는데 패스 1에서는 중간언어의 각 행을 읽어서 명령 및 심볼에 대한 자료를 구축하고, 패스 2에서는 패스 1에서 구축된 자료를 기본으로 파이프라인 단계별로 시뮬레이션을 수행한다.

동시에 다중 이슈되는 명령들은 시뮬레이터가 자동으로 명령간의 데이터 종속관계를 조사하여 종속관계가 없는 명령에 대해서만 동시에 실행될 수 있도록 스케쥴된다. 시뮬레이터에서 스케쥴은 각 명령이 이

```
<# of static instructions>
total # : 55
          dataop    :   12, 0.22
          movop     :    4, 0.07
          cmpop     :    4, 0.07
          cvop      :    0, 0.00
          ldrop     :   19, 0.35
          strop     :    6, 0.11
          jump      :    3, 0.05
          jt, jf    :    4, 0.07
          retop     :    1, 0.02
          call      :    0, 0.00
          push,pop  :    2, 0.04
          nop       :    0, 0.00
          misc.     :    0, 0.00
<segment address>
text segment => start address : 00000000,
                length : 220 bytes
data segment => start address : 00010000,
                length : 60 bytes
-------------------------------------------------
<executed clocks>
executed clocks : 34 clocks
<# of dynamic instructions>
total # : 36
          dataop    :    7, 0.19
          movop     :    3, 0.08
          cmpop     :    3, 0.08
          cvop      :    0, 0.00
          ldrop     :   12, 0.33
          strop     :    3, 0.08
          jump      :    2, 0.06
          jt, jf    :    3, 0.08
          retop     :    1, 0.03
          call      :    0, 0.00
          push,pop  :    2, 0.06
          nop       :    0, 0.00
          misc.     :    0, 0.00
```

(그림 5.2) 예제의 시뮬레이션 측정 결과
(Fig. 5.2) A sample of simulation result

슈 수에 따라 이슈와 수행순서(in-order issue, in-order execution)를 유지하면서 스케쥴된다. (그림 5.2)는 예제 프로그램의 일반 코드에 대하여 2 이슈로 처리된 후 시뮬레이터가 생성한 실행 측정 결과이다.

5.3 시뮬레이션

본 논문에서 제안된 알고리듬의 성능 평가를 위하여 〈표 5.1〉과 같은 C 언어로 작성된 5개의 프로그램을 선택하여 시뮬레이션 하였다. 시뮬레이션 구분은 〈표 5.1〉에서와 같이 4가지 유형에 대하여 각각 1, 2 및 4 이슈에 대한 수행 성능에 대한 측정치를 조사하였다. 모드 Ⅰ에서는 에서는 C소스 프로그램을 컴파일한 결과 그대로를 시뮬레이션 하였고 모드 Ⅱ에서는 모드 Ⅰ의 어셈블리 코드에 리스트 스케쥴링 처리를 하여 시뮬레이션 하였으며 모드 Ⅲ에서는 모드 Ⅱ의 어셈블리 코드에 대하여 조건반응 지원 알고리듬을 적용하지 않고 병렬성만을 처리한 노드에 대하여 시뮬레이션하였다. 마지막으로 모드 Ⅳ에서는 모드 Ⅱ의 어셈블리 코드에 대하여 본 논문에서 제안하는 조건반응 지원 알고리듬을 적용한 ILP코드에 대한 시뮬레이션을 실행하였다.

〈표 5.2〉는 5개의 프로그램에 대한 4가지 모드에 대한 시뮬레이션의 결과를 종합한 표이고 (그림 5.3)은 〈표 5.1〉의 프로그램에 대한 〈표 5.2〉의 측정치를 명령 이슈 및 클럭수에 따라 나타낸 그림이다.

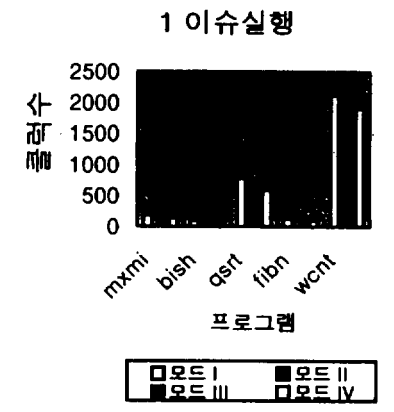
〈표 5.1〉 시뮬레이션 프로그램 및 모드
〈Table 5.1〉 Simulation programs and modes

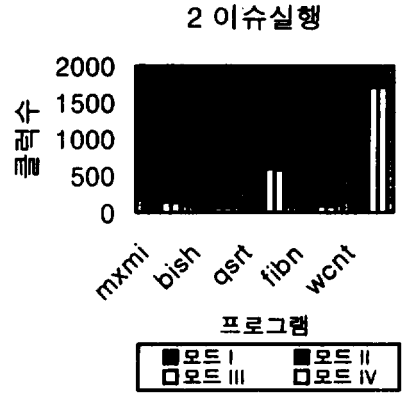
내용구분	내 용 설 명	
시뮬레이션 프로그램 (C 소스)	mxmi	max-minmum
	bish	binary search
	qsrt	quick sort
	fibn	fibonacci sequence
	wcnt	word counter
시뮬레이션 모드	Ⅰ	스케쥴 없는 일반코드 처리
	Ⅱ	모든 Ⅰ에 리스트 스케쥴링 처리
	Ⅲ	Ⅱ에 조건반응을 고려하고 ILP 처리
	Ⅳ	Ⅱ에 조건반응 스케쥴링을 처리하고 ILP 처리

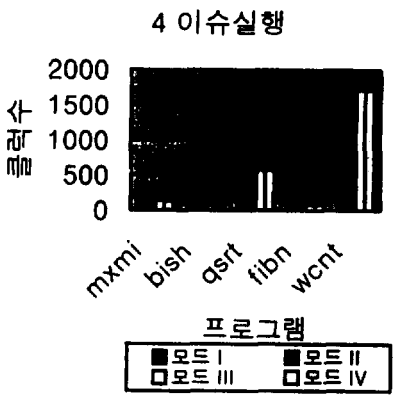
〈표 5.2〉 시뮬레이션 모드에 따른 측정값
〈Table 5.2〉 Results of simulation

모드	Ⅰ			Ⅱ		
이슈	1	2	4	1	2	4
Mxmi	181	160	159	161	148	146
Bish	93	80	79	86	72	71
Qsrt	784	712	698	621	594	592
Fibn	127	102	100	108	92	81
Wcnt	2120	1802	1798	1980	1790	1788

모드	Ⅲ			Ⅳ		
이슈	1	2	4	1	2	4
Mxmi	150	142	141	142	138	131
Bish	80	68	67	71	67	66
Qsrt	614	592	590	593	578	576
Fibn	108	98	97	94	82	81
Wcnt	2002	1711	1708	1910	1708	1706







(그림 5.3) 명령 이슈 및 클럭수에 따른 성능비교 그래프
(Fig. 5.3) Graphs with instruction issue and clock cycle

6. 결 론

본 논문에서는 조건실행을 지원하는 ILP 프로세서들의 성능 개선을 위하여 기본블록을 넘어선 조건실행 지원 스케쥴링 알고리듬을 제안하였다. 제안된 알고리듬은 컴파일러가 생성한 중간언어를 입력으로 받아 데이터 흐름 분석을 수행한 후 조건 분기문을 제거하는 IF-변환을 수행하였다. IF-변환 수행 후 변환된 중간언어의 조건을 나타내는 부울 오퍼랜드를 분석하여 이들 조건 오퍼랜드 간의 포함관계를 분석한 후에 슈퍼블록을 구성한 후 슈퍼블럭 내의 명령어들을 입력으로 받아 데이터 종속 그래프(DDG)를 구성한다. DDG 구성 시 조건 오퍼랜드 간에 서로 포함관계가 없는 상호배타적인 명령어들은 연산 오퍼랜드 간에 데이터 종속관계가 있더라도 서로 동시에 실행되지 않으므로 DDG 상에서 데이터 종속 관계를 표시하지 않으므로써 스케쥴 시 병렬처리 될 수 있도록 적용하였다. 적용된 스케쥴링 알고리듬은 조건실행으로 이동의 가능성이 많은 상호배타적 명령어들을 최대한 위로 이동함으로써 이후의 명령어들의 스케쥴 가능성을 높였다.

제안 개발된 스케쥴링 알고리듬의 성능 평가를 위해 C 컴파일러와 시뮬레이터를 개발하고 다양한 벤치마크 프로그램에 대하여 시뮬레이션하여 성능을 측정하였다.

성능측정 결과 1 이슈실행에서는 약 26%의 성능 개선이 있었고 2 이슈실행에서는 약 17%의 성능 개선을 측정하였다. 4 이슈실행에서는 프로그램내의 충분한 병렬성이 부족하여 2 이슈실행과 비교하여 그다지 성능이 개선되지 않았다.

제안된 알고리듬의 성능으로 1, 2 이슈실행에서 약 20% 정도의 성능 개선이 측정되었으므로 제안된 알고리듬의 타당성을 확인하였다.

참 고 문 헌

[1] A. V. Aho, R. Sethi, J. D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing ©, 1988.

[2] C. Fraser and D. Hanson, *A Retargetable C Compiler : Design and Implementation*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[3] D. M. Gillies, D. R. Ju, R. Johnson and M. Schlansker, "Global predicate analysis and ints application to register allocation", IEEE Micro, pp. 114-125, December 1996.

[4] D. N. Pnevmatikatos, "Incorporating Guarded Execution into Existing Instruction Sets", Ph.D. Thesis, Dept. of Computer Science, University of Wisconsin-Madison, 1996.

[5] D. M. Lavery, W. W. Hwu, "Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs", IEEE Macro, pp. 126-137, December 1996.

[6] D. C. Lin, "Compiler support for predicated execution in superscalar processors", Master's thesis, Univ. of Illinois at Urbana-Champaign, 1990.

[7] Free Software Foundation, *GNU FLEX Manual*, 1995.

[8] Free Software Foundation, *GNU BISON Manual*, 1995.

[9] G. Tyson, "The Effects of Predicated Execution on Branch Prediction", IEEE Micro-27, pp. 196-206, November 1994.

[10] J. A. Fisher, "Trace Scheduling : A Technique for Global Microcode Compaction", IEEE Transactions on Computers, C-30(7), July 1981.

[11] J. Hennessy and D. Patterson, *Computer Archi-*

tecture a Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[12] J. Hennessy and D. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers Inc., 1994.

[13] J. P. Bennet, *Introduction to Compiling Techniques ; A First Course using ANSIC, LEX and YACC*, McGraw-Hill, 1990.

[14] J. R. Ellis, *Bulldog: A Compiler for VLIW Architecture*, The MIP Press, 1986.

[15] K. Hwang, *Advanced Computer Architecture Parallelism, Scalability, Programmability*, McGraw-Hill, 1993.

[16] M. Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, 1991.

[17] N. Gloy, M. D. Smith, C. Young, "Performance Issues in Correlated Branch Prediction Schemes", IEEE Micro-28, pp. 3-14, Nov 1995.

[18] R. Johnson and Michael Schlansker, "Analysis Techniques for Predicated Code", IEEE Micro-29, pp. 100-113, December 1996.

[19] S. A. Mahlke, "Exploiting Instruction Level Parallelism in the presence of Conditional Branches", Ph.D. Thesis, Electrical Engineering, University of Illinois at Urbana-Champaign, 1996.

[20] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, W. W. Hwu, "Characterizing the Impact of Predicated Execution on Branch Prediction", IEEE Micro-27, pp. 217-227, November 1994.

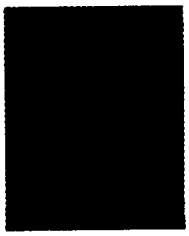
[21] V. Kathail, M. Schlansker, B. R. Rau, *HPL Play Doh Architecture Specification: Version 1.0*, HPL-93-80, February, 1994.

[22] Yamana H., Sato M., Kodama Y., sakane H., Sakai S., Yamaguchi Y.,: Survey of Speculative Execution and the Effect of Task-level Speculation; 51th Nat.Conv.IPSJ, Vol. 6, No. 1 P-3, pp. 75-76, 1995.

[23] 경종민, "최신 마이크로프로세서 구조", 대한전자공학회 전자계산연구회 컴퓨터기술, 제8권 제1호, pp. 9-24, 1994. 12.

[24] 심현규, 김유신, 이상정, "ILP프로세서를 위한 성능 측정 도구의 개발", 대한전자공학회 논문집, 제24권 제1호, pp. 3-6, 1997. 4.

[25] 유병강, 이상정 "명령어수준 병렬처리 프로세서를 위한 소프트웨어 스케쥴링 알고리듬", 대한전자공학회 논문집, 제17권 제2호, pp. 826-829, 1994. 11.



유 병 강

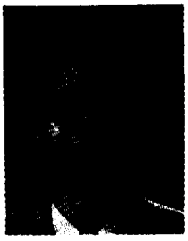
1992년 서울산업대학교 전자계산과(공학사)

1995년 순천향대학교 대학원 전산학과(공학석사)

1998년 순천향대학교 대학원 전산학과(공학박사)

1991년~현재 충남산업대학교 전자계산소

관심분야: VLIW/슈퍼스칼라 프로세서, ILP 스케쥴링, S/W 및 H/W 스케쥴링, OS 스케쥴링



이 상 정

1983년 한양대학교 전자공학과(공학사)

1985년 2월 한양대학교 대학원 전자공학과(공학석사)

1985년 8월 한양대학교 대학원 전자공학과(공학박사)

1988년~현재 순천향대학교 컴퓨터학부 교수

관심분야: 프로세서설계, 최적화 컴파일러 설계, 마이크로프로세서 응용