2001년 한국정보처리학회 춘계 학술발표논문집 제8권 제1호

간단한 바이패싱 토폴로지를 보완한 명령어 스케쥴링 방법

김민진, 김석주, 김석일
충북대학교 컴퓨터과학과
e-mail: mjkim1@kaeri.re.kr

An Instruction Scheduling to Compensate Simple Bypassing Topologies

Minjin Kim, Sukju Kim, Sukil Kim
Dept of Computer Science, Chungbuk National University

요 약

바이패싱 회로는 명령어의 파이프라인의 실행 단계가 종료되자마자 연산 결과를 다음 번 파이프라인 단계의 실행 단계의 실행 유니트에서 사용할 수 있어 명령어 실행 시간이 단축된다. 그러나 바이패싱 회로의 복잡도는 연산처리기가 늘어남에 따라 크게 증가하는 단점이 있으므로 명령어 중복 할당 기법을 적용하면 여러 개의 연산처리기에서 동일한 연산을 수행하여 VLIW 구조에서 가상의 바이패싱 회로가 존재하는 것과 같은 효과를 얻을 수 있다.

1. 서론

VLIW(Very Long Instruction Word) 형태는 제어 흐름이 간단하고 각 연산처리기가 동기적으로 동작하기 때문에 광범위하게 연구되고 있다[1,2]. 특히 근래에 들어와 멀티미디어를 빠르게 처리하는 프로세서 구조의 경우에는 미디어 정보가 가지고 있는 서브워드 병렬성(subword parallelism)을 활용하는 구조로 각광을 받고 있다[3]. 이러한 VLIW 구조의 특징을 충분히 이용하기 위해서는 실행시 동시에 실행이 가능한 명령어들을 모아 긴 명령어로 구성되는 목적 코드를 생성할 수 있어야 한다. 또한 목적 코드를 생성하는 과정에서 보다 많은 병렬성을 추출하기 위하여 퍼콜레이션 스케쥴링(percolation scheduling), 추론 스케쥴링(speculation scheduling), 파이프라인 스케쥴링(pipeline scheduling) 등의 기법이 연구되었다[4,5,6].

하드웨어의 측면에서 보다 빠른 계산을 하기 위하여 연산처리기의 수를 늘리거나 명령어 파이프라인의 실행단계가 종료되자마자 연산 결과를 다음 번 파이프라인 단계의 실행 유니트에서 사용할 수 있도록 하는 바이패싱(bypassing)기법이 있다. 특히 Viper 구조[7]는 각 연산처리기별로 명령어 파이프라인의 실행 단계의 연산 결과가 같은 파이프라인의 입력단으로 되돌려질 뿐만 아니라 서로 다른 연산처리기의 명령어 파이프라인의 실행 단계의 입력단으로 연결되어있다. 결국 이러한 바이패싱 회로는 하나의 연산처리기에서 계산된 중간 결과를 다른 연산처리기에서도 사용할 수 있도록 하므로 연산이 여러 개의 연산처리기에서 동시에 수행된 것과 동일한 상태로 간주될 수 있다. 그러나 바이패싱 회로의 복잡도는 연산처리기의 수가 늘어남에 따라 크게 증가하는 단점이 있다.

이러한 단점을 개선하기 위하여 본 논문에서는 바이패싱 대신에 여러 개의 연산처리기에서 동일한 계산을 수행하도록 하여 가상의 바이패싱 회로가 존재하는 것과 같은 효과를 얻을 수 있는 명령어 중복 할당 기법을 제안하였다. 이 방법은 연속된 명령어간의 자료의존성으로 말미암아 불필요하게 추가된 빈 명령어(NOP)대신 의미 있는 명령어를 중복 할당함으로써 바이패싱 회로를 장치한 것과 동일한 효과를 얻을 수 있다.

본 논문의 제 2절에서는 전통적인 Viper 구조에서 목적 코드를 생성하는 기법의 예를 들었으며 제 3절에서는 기존의 복사 기법[8,9]을 Viper 구조에 적용한 명령어 중복 할당 기법을 제안하였다. 제 4절에서는 제안한 기법을 대상으로 여러 가지 명령어 그래프를 대상으로 모의 실험을 수행하였다. 마지막으로 제 5절에서는 본 논문의 결론을 도출하였다.

2. Viper와 목적 코드 생성

Viper 구조는 각 연산처리기별로 명령어 파이프라인의 실행 단계의 입력단으로 되돌려질 뿐 아니라 서로 다른 연산처리기의 명령어 파이프라인의 실행 단계의 입력단으로 연결되는 회로를 추가하여 하나의 연산처리기에서 계산된 중간 결과를 다른 연산처리기에서도 사용할 수 있도록 하고 있다. 그림 1은 네 개의 연산처리기로 구성된 VLIW 구조의 여러 가지 구성을 도시한 것이다.

바이패싱 회로로 연결되지 않은 연산처리기 F_0와 F_1에서 수행되는 명령어 I_1과 I_2간에 자료 의존 관계가 존재한다면 F_1은 F_0가 명령어 I_1의 계산을 종료하고 그 결과를 레지스터 파일에 기록한 후에야 비로소 I_2를 실행 단계로 진입시킬 수 있다. 즉, 바이패싱 회로로 연결되지 않은 한 쌍의 연산처리기에 할당된 명령어들간에 자료 의존 관계가 존재할 경우에는 레지스터 파일을 참조하는데 필요한 지연이 발생한다. 이러한 지연은 결국 NOP으로만 구성된 긴 명령어를 목적 코드에 추가하거나 긴 명령어 내에 부분적으로 NOP이 삽입되도록 한다. 그러나 같은 연산처리기에 할당된 연속된 명령어간의 자료 의존 관계는 그림 1(b)와 같이 F_0와 F_1 간에 바이패싱 회로를 추가한 P_2 구조에서는 지연이 발생되지 않는다.

그림 2(a)의 소스프로그램을 토대로 두 개의 연산처리기로 구성된 P_1 구조용 목적 코드가 생성되는 과정을 살펴보자. 여기서 P_1 구조란 각 연산처리기 자체에만 바이패싱 회로가 있는 구조이다. (b)는 (a)를 토대로 얻은 명령어 그래프이며

노드와 화살표 있는 간선은 각각 명령어와 명령어간의 자료 의존 관계를 나타낸다.

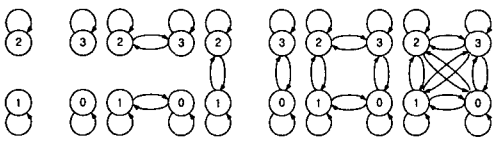
(c)와 (d)는 각각 두 개의 연산처리기로 구성된 Viper 구조에서 가능한 명령어 할당 결과이다. 즉, I_1과 I_2간에는 자료 의존 관계가 존재하지 않으므로 서로 다른 연산처리기에 할당되어 하나의 긴 명령어를 구성한다. I_3과 I_4는 각각 I_1과 I_3와의 자료의존관계로 인하여 동일한 연산처리기(F_0)에 할당된다. 만일 I_4를 두 번째 연산처리기(F_1)에 할당하는 경우에는 F_0에 할당된 I_3와의 자료 의존 관계로 인하여 I_3가 포함된 긴 명령어와 I_4가 포함된 긴 명령어 사이에 NOP으로만 이루어진 긴 명령어가 삽입되어야 한다. 따라서 I_4는 F_0에 할당되어야 한다.

I_4와 I_5간에는 자료의존관계가 없으므로 I_5는 I_4가 할당된 연산처리기에 할당하거나 다른 연산처리기에 할당이 가능하다. (c)는 동일한 연산처리기 F_0에 할당하는 경우를 보인 것이다. 그러나 I_5를 다른 연산처리기(F_1)에 할당하는 경우에는 I_5와 I_3간의 자료의존관계($15)로 인하여 (d)와 같이 한 사이클 늦은 네 번째 사이클에 실행되도록 목적 코드가 생성되어야 한다.

P_2의 경우에는 I_5를 실행하는데 필요한 피연산자 $15가 바이패싱 회로를 통하여 F_0로부터 제공될 수 있으므로 (e)와 같이 I_5와 I_4를 하나의 긴 명령어로 구성할 수 있다. 따라서 P_2에서는 P_1보다 한 사이클 빠른 계산이 가능하다.

만일 (d)에서 두 번째 긴 명령어의 NOP의 위치에 중복해서 I_3를 할당하고 I_5를 F_1에 할당하면 I_3과의 자료의존관계가 해결되므로 (f)와 같이 목적 코드를 구성하는 긴 명령어의 수가 하나 적어지게 되므로 P_2에서와 같이 전체적으로 1 사이클 빠른 계산이 가능하다. 즉, NOP이 포함될 코드 슬롯에 다른 연산처리기가 실행할 명령어를 중복 할당하는 경우에 전체적으로 목적 코드의 길이가 짧아져 프로그램의 실행 사이클을 줄일 수 있다.

(e)와 (f)를 비교하면 P_2에서의 할당 결과는 P_1 구조의 NOP에 의미 있는 명령어를 중복 할당한 결과와 동일한 결과를 얻을 수 있음을 알 수 있다. 따라서 명령어 중복 할당 스케쥴링을 적용하면 바이패싱 회로를 추가하거나 완전 연결과 같은 바이패싱 토폴로지의 채용을 하지 않고도 명령어의 중복 할당만으로도 비슷한 효과를 얻을 수 있음을 알 수 있다.

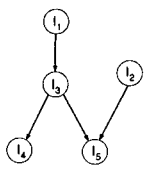


(a) P_1 (b) P_2 (c) P_3 (d) P_4 (e) P_5

그림 1. 바이패싱 회로의 연결 토폴로지

```
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) P_1 목적 코드의 예 1

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

(d) P_1 목적 코드의 예 2

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

(e) P_2 목적 코드

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

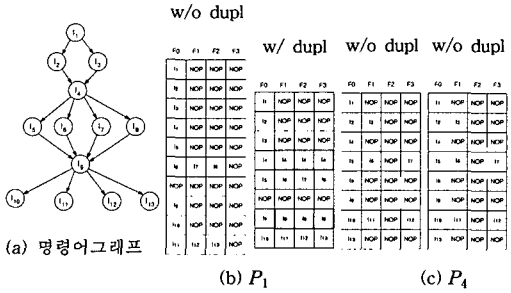
(f) 명령어를 중복 할당한 P_1 목적 코드

그림 2. Viper 구조와 목적 코드 생성

3. Viper 구조에서의 명령어 중복 할당 기법

응용 프로그램 내에 명령어 수준 병렬성이 충분하지 않은 경우에는 긴 명령어 형태를 구성하기 위해서는 미처 채워지지 않은 명령어 영역을 빈 명령어(NOP)로 채워야 한다. 또한 이렇게 구성된 긴 명령어간에 내재되어 있는 자료 종속 관계나 자원 충돌을 제거하기 위해서는 빈 명령어로만 채워진 긴 명령어(LNOP)을 삽입하여야 한다. 이런 빈 명령어들의 삽입으로 인해 목적 코드의 길이가 길어지며 메모리의 낭비가 초래된다. 특히 캐시 메모리 내에 실제 실행되는 명령어에 비해 이런 빈 명령어들이 차지하고 있는 비율이 커지면 캐시의 이용율이 떨어져 전체 시스템의 성능이 저하될 수 있다. 목적 코드에서 NOP이 차지하는 비중은 응용 프로그램에서 자료의존관계가 복잡해질수록 점차 증가한다.

Viper 구조용 목적 코드에서 NOP을 대신하여 의미 있는 명령어를 중복 할당하여 명령어간의 자료의존관계를 해결할 수 있다면 낭비되는 코드 슬롯을 이용하여 응용 프로그램의 빠른 수행이 가능하다. 다음의 예제를 통하여 명령어 중복 할당 기법의 장점을 살펴보자.



(a) 명령어그래프 (b) P_1 (c) P_4

그림 3. 각 바이패싱 토폴로지별 명령어 중복 할당 결과

그림 3(a)의 명령어 그래프에서 각 노드는 명령어를 표시하며 화살표 있는 간선은 각 명령어간의 자료 의존 관계를

표시하고 있다. (b)의 왼쪽 도표는 P_1 구조에서 명령어의 중복 할당을 실행하지 않고 각 연산처리기 자체에서 실행 결과가 바로 바이패싱 되어 다음 명령어의 실행 단계에서 사용할 수 있도록 한 것이다. 이때의 실행 시간은 총 10 사이클이 걸렸다. (b)의 오른쪽 도표에서는 P_1 구조에서 명령어 중복 할당을 실행한 것으로 음영이 진 곳은 빈 명령어 슬롯(NOP)에 부모 노드에 해당되는 명령어가 중복 할당된 것이다.

P_1 구조에는 각 연산처리기 사이에는 바이패싱 회로가 없으므로 명령어 I_4와 자료 의존 관계가 있는 명령어 I_5, I_6, I_7, I_8은 명령어 I_4를 중복 할당함에 의하여 1 사이클 지연 없이 실행될 수 있다. 또 명령어 I_9을 중복 할당하면 I_9과 자료 의존 관계가 있는 명령어 I_{10}, I_{11}, I_{12}, I_{13}이 1 사이클 지연 없이 실행된다. 이와 같이 명령어 중복 할당으로 인하여 실행 시간이 8 사이클로 감소되었다.

(c)의 왼쪽은 P_4 구조에서 명령어 중복 할당을 실행하지 않은 결과이다. 이 경우는 자료 의존 관계가 있는 명령어가 할당된 연산처리기 사이에 바이패싱 회로가 존재하므로 명령어가 지연 없이 실행되어 P_1 구조에서 명령어 중복 할당을 실행한 결과와 같은 8 사이클의 실행 시간을 보여준다.

(c)의 오른쪽에서는 P_4 구조에서 명령어 중복 할당을 실행하였으나 그 효과가 나타나지 않았는데 이는 현재의 알고리즘이 단지 부모 노드를 중복 할당할 때의 단축되는 시간과 레지스터 파일이나 바이패싱 회로를 통하여 자료를 전달받을 때 단축되는 실행시간을 비교하여 이득이 없으면 명령어 중복 할당을 실행하지 않기 때문에 이런 결과가 발생된 것으로 짐작된다. 이는 부모 노드의 부모 노드까지도 중복 할당을 하면 좀 더 개선된 효과를 얻을 수 있을 것이다.

P_4 구조와 P_5 구조로 진행되며 바이패싱 회로가 복잡하여짐에 따른 하드웨어적 지원으로 인해 명령어 중복 할당의 효과가 나타나지 않았다.

위에서 기술한 명령어 중복 할당 기법의 알고리즘을 다음과 같이 그림 4에 제시하였다.

Algorithm : Redundant Instruction Scheduling
input : Instruction set $I=\{i_1, i_2, \ldots, i_n\}$
Data dependency set $D=\{d_{ij} \mid 1\le i,j\le n\}$
Functional unit set $F=\{F_1, F_2, \ldots, F_q\}$
output : VLIW codes
BEGIN
Let M_a , B_a be queues for every functional unit. a =1,2, . . . , q
for every instruction i_j **do**
Determine the level of instruction. $l_j = \max\{l_k \mid \text{for } \forall\ i_k,\ d_{kj}\neq 0\} + 1$
end for
Let L_m to be the set of instruction i_j such that $l_j = m$.
for each group L_m **do**
for every instruction $i_i \in L_m$, $1 \le i \le |L_m|$ **do**
Reset B_a , a =1,2, . . . , q
// Queue B_k is an interim temporary queue.
for every functional unit F_k **do**
Find $i_f \in G_i$ such that $\beta_i = \psi_f^{\mu(f)} + \delta(F_{\mu(f)}, F_k)$ is the latest.
if there exists bypassing between $F_{\mu(f)}$ and F_k
then $\delta(F_{\mu(f)}, F_k) = 0$
else $\delta(F_{\mu(f)}, F_k) = 1$
// β_i is the time when the communication
// between i_f and i_i is completed
// τ_k is the completion time of the instructions already
// allocated to F_k
if $\tau_k \ge \beta_i$ // if i_i can not be allocated to F_k before β_i
then
// duplicating i_f to F_k does not make any good.
Put instruction i_i to queue B_k .
$\psi_i^k = \tau_k + ex_i$
else // check whether duplicating instruction i_f to F_k shortens
// the completion time.
if $\beta_i \ge \max_{i_w \in G_f} \{\tau_k, \psi_w^{\mu(w)} + \delta(F_{\mu(w)}, F_k)\} + ex_f$
then
Put instruction i_f to queue B_k .
Recalculate β_i
Put instruction i_i to queue B_k .
$\psi_i^k = \beta_i + ex_i$
else
// Duplicating instruction i_f to F_k does not make any good.
Put instruction i_i to queue B_k .
$\psi_i^k = \beta_i + ex_i$
end if
end if
end for
Find functional unit F_b with the min$\{\psi_i^k \mid k=1,\ldots,q\}$.
Copy queue B_b to queue M_b.
end for
end for
END

그림 4. 명령어 중복 할당 알고리즘

명령어 그래프 $G=(D, I)$ 는 명령어 집합 $I=\{i_1, i_2, \ldots, i_n\}$ 과 자료 의존성(data dependency)을 표시하는 방향성 있는 간선(edge)을 나타내는 집합 $D=\{d_{ij} \mid 1\le i,j\le n\}$으로 구성된다. 연산처리기 집합은 $F=\{F_1, F_2, \ldots, F_q\}$로 나타낸다. 알고리듬은 먼저 명령어간의 자료 의존성에 기초하여 병렬로 수행될 수 있는 명령어들에게 동일한 우선 순위를 준다. root 노드에 속하는 모든 명령어 i_r 의 우선 순위 l_r은 가장 높은 우선 순위 0을 주고 나머지 명령어 i_i 의 우선순위 l_i는 다음과 같다.

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

이때 ψ_i^k 는 명령어 i_i 가 연산처리기 F_k 에서 완료되는 시간이다. μ 는 명령어로부터 연산 처리기로의 할당함수로 명령어 i_i 가 연산처리기 F_k 에 할당되면 μ(i) = k이다. G_i 는 명령어 i_i 에 직접적인 자료 의존 관계(data dependency relation)로 영향을 주는 명령어들의 집합이다. ex_i 는 명령어 i_i 의 실행에 요구되는 시간(사이클)으로 1 로 가정하였다. $\delta(F_{\mu(i)}, F_{\mu(j)})$ 는 명령어 i_i 가 실행된 연산 처리기 $F_{\mu(i)}$ 로부터 명령어 i_j 가 실행되는 연산처리기 $F_{\mu(j)}$ 로 결과를 전달받는데 걸리는 시간으로 이때 명령어 i_i 가 명령어 i_j 에 자료 의존 관계로 영향을 주므로 $d_{ij}=1$ 이 된다.

일단 명령어의 그룹이 정해지면 해당 그룹에 속한 명령어에 대해 ①자료 의존 관계에 있는 명령들이 다른 연산처리기에 할당된 경우, 이들로부터 레지스터 파일을 통하거나 바이패싱을 통하여 자료를 전달받는 경우의 수행완료 시간과 ②필요한 명령어를 복사하여 중복 수행할 경우의 명령어 실행 완료 시간을 비교하여 ②의 결과가 ①에 비하여 나은 경우에 명령어의 중복 할당을 허용하도록 하는 것이다.

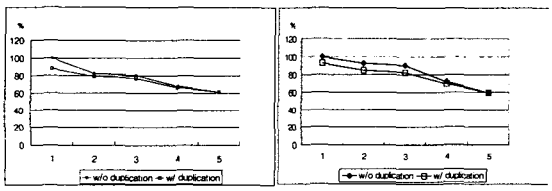
이러한 알고리듬을 수행하면 중복 할당된 명령어가 해당 명령어의 실행을 앞당기는 효과를 갖게 되고 같은 실행 완료 시간 (사이클) 에 할당된 명령어들을 긴 명령어로 만들면 중복 할당된 명령어가 긴 명령어에 포함되게 된다.

4. 실험 및 고찰

본 논문에서 제안한 명령어 중복 할당 기법의 성능을 평가하기 위하여 4개의 정수형 연산처리기에서 서로 다른 바이패싱 토폴로지를 가정하였다. 또한, 모든 명령어의 실행 사이클은 한 사이클이고 매 사이클마다 하나의 명령어 실행이 완료되며 캐시 미스는 일어나지 않는 것으로 간주하였다. 다른 연산처리기에서 수행된 결과를 필요로 할 때 통신비용은 두 개의 연산 처리기를 연결하는 바이패싱 회로가 없을 때 한 사이클의 시간이 걸리는 것으로 간주하였다.

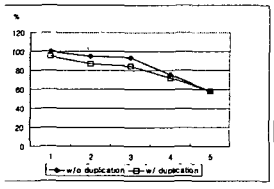
실험에서는 자료 의존 관계를 가지는 명령어 그래프(instruction graph)를 임의로 발생시키고, 각 명령어 그래프에서는 병렬성(parallelism), 밀도(density), 높이(height)를 제한된 범위에서 임의로 생성시켰다. 여기서 병렬성이란 명령어 그래프에서 같은 레벨에 있는 명령어의 수의 최대값을 말하며 밀도는 하위 레벨과 자료 의존 관계를 나타내는 정도로서 간선(edge)이 완전히 연결(fully connected)된 경우가 100%이며 완전한 이진 트리 구조의 밀도는 50%가 된다.

그림 5의 실험 결과에서 병렬성은 2~5개, 높이는 20, 밀도는 60%~100%로 변화시켰다. 각 경우별로 100개씩의 명령어 그래프를 생성하여 실험하였다. 이 그래프에서 X축은 바이패싱 토폴로지 유형을 표시하며 Y축은 바이패싱 토폴로지 P1에서 기존의 방법(즉 명령어 중복 할당을 실행하지 않은 방법)으로 스케줄링하여 실행한 결과를 100%로 할 때 그에 대한 각각의 상대적인 비율을 표시한 것이다.

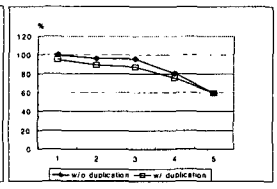


(a) 병렬성: 4, 밀도: 60%

(b) 병렬성:4, 밀도: 75%



(c) 병렬성: 4, 밀도: 85%



(d) 병렬성: 4, 밀도: 95%

그림 5. 실험 결과 비교

그림 (a)는 병렬성 4, 높이 20, 밀도 60%로 유지하였을 때의 실험 결과로서 P_1 구조에서 명령어 중복 할당의 효과가 가장 크고 P_5 구조에서 효과가 나타나지 않았다. (b)에서는 병렬성 4, 높이 20, 밀도 75%로서 명령어 중복 할당의 효과가 고르게 나타났다. 여기서 자료 의존 관계를 나타내는 밀도가 높아지면 명령어 중복 할당의 효과가 높아지는 것을 알 수 있다. (c)에서는 병렬성 4, 높이 20, 밀도 85%인 경우로 명령어 중복 할당이 효과를 보이고 있는데 바이패싱 연결도가 높아지면서 그 효과가 줄어듦을 알 수 있다. (d)에서는 병렬성 4, 높이 20, 밀도 95%로서 (c)의 경우와 같이 명령어 중복 할당의 효과가 나타났으며 바이패싱 연결도가 높아지면서 줄어들었다.

위의 실험 결과로부터 다음과 같은 사실을 알 수 있다.

1) 바이패싱 토폴로지의 연결도가 증가할수록 전체 실행 시간이 감소되어 성능이 향상됨을 알 수 있다.
2) 명령어 중복 할당을 실행한 결과가 실행하지 않은 결과보다 성능이 우수하거나 같다.
3) 바이패싱 토폴로지의 연결도가 높아짐에 따라 명령어 중복 할당을 실행한 결과와 안 한 것의 성능 차가 감소되었다.

5. 결론

바이패싱 토폴로지가 적용된 정수형 연산처리기를 갖는 VLIW 구조는 연산처리기에서 명령어의 파이프라인의 실행 단계가 종료되자마자 연산 결과를 다음 번 파이프라인 단계의 실행 유니트에서 사용할 수 있어 명령어 실행 시간이 단축된다. 반면에 연산처리기가 증가하면 바이패싱 회로의 복잡도가 증가되는 단점이 있다.

이 논문에서는 NOP이 차지하는 슬롯에 의미 있는 명령어를 중복 할당하여 자료 의존 관계를 해소하고 프로그램 실행 사이클을 단축시킬 수 있는 명령어 중복 할당 기법을 적용하였는데 그 결과 회로의 복잡도를 증가시키지 않으면서 실행 시간을 단축시켜 가상의 바이패싱 회로를 추가한 효과를 얻을 수 있었다.

참고 문헌

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

[2] 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

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

[4] Nicolau, A. (1985). *Percolation Scheduling: A Parallel Compilation Technique.* Technical Report TR 85-678, Cornell University, Ithaca, NY 14853, USA.

[5] D. W. Wall. *Speculative execution and instruction-level parallelism.* Technical Report, DEC-WRL, Mar. 1994

[6] M. Lam. "Software pipelining: An effective scheduling technique for VLIW machines," In *Proc. ACM SIGPLAN Conf. Programming Languages Design and Implementation,* pp. 318-328, Atlanta, GA, June 1988.

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

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

[9] 문현주, 이승수, 김석주, 김석일, "NOP 명령어 슬롯을 활용하는 VLIW 코드 생성 기법", *정보과학회 추계학술대회 논문집*, 27권 2호, pp. 615-617, 2000