

저전력 VLIW 명령어 추출을 위한 연산 재배치 기법

(Operation Rearrangement for Low-Power VLIW Instruction Fetches)

신 동 군 [†] 김 지 흥 ^{**}

(Dongkun Shin) (Jihong Kim)

요약 이동용 응용프로그램이 요구하는 계산량이 늘어남에 따라 많은 이동용 컴퓨터 시스템이 성능을 높이기 위해서 VLIW 프로세서를 사용하여 설계되고 있다. VLIW 구조에서는 하나의 명령어(instruction)가 여러 개의 연산(operation)을 가지고 있는 데, 이들이 명령어안에서 어떻게 배치되느냐에 따라 명령어 추출(fetch)시의 전력 소모가 큰 차이를 보인다. 본 논문에서는 저전력 VLIW 명령어 추출을 위해 컴파일러의 후 단계로 사용되는 최적의 연산 재배치 기법을 제시한다. 제안된 방법은 연속적인 명령어 추출시의 스위칭 활동(switching activity)이 최소화 되도록 연산의 순서를 수정한다. 벤치마크 프로그램에 대해 실험해 본 결과, 제안된 기법을 사용하여 명령어를 재배치하는 경우 명령어 추출시 스위칭 활동이 평균적으로 약 34% 줄어듦을 확인하였다.

Abstract As mobile applications are required to handle more computing-intensive tasks, many mobile devices are designed using VLIW processors for high performance. In VLIW machines where a single instruction contains multiple operations, the power consumption during instruction fetches varies significantly depending on how the operations are arranged within the instruction. In this paper, we describe a post-pass optimal operation rearrangement method for low-power VLIW instruction fetch. The proposed method modifies operation placement orders within VLIW instructions so that the switching activity between successive instruction fetches is minimized. Our experiment shows that the switching activity can be reduced by 34% on average for benchmark programs.

1. 서론

이동용 응용프로그램이 비디오 디코딩(decoding)과 같이 계산량이 많은 작업을 처리하도록 요구됨에 따라 많은 이동용 컴퓨터 시스템이 고성능을 위해서 VLIW 프로세서를 사용하여 설계되고 있다. 예를 들면, 이동용 인터넷 시장을 위해 개발된 Transmeta사의 Crusoe 프로세서[1]의 경우 64비트나 128비트 VLIW CPU에 기반하고 있다. 또한 무선 휴대용 전화기에 쓰이는 Fujitsu Microelectronics사의 FR300[2]도 VLIW 구조

를 사용한다. 또한, 디지털 신호처리를 위해 개발된 TMS320C6x 계열의 VLIW DSP들도 가까운 미래에 무선 장치들에 사용되리라 예상되고 있다[3,4].

VLIW 프로세서는 계산량이 많은 응용프로그램을 처리할 수 있는 계산능력을 제공하지만 대부분 많은 전력을 소비한다. 예를 들어, StrongArm 110과 같은 고성능 내장형 마이크로 프로세서가 공급전압 3V에서 100mW에서 1W의 전력을 소모하는 반면, TMS320C 620x 프로세서는 1.8V의 공급전압에서 1.2W에서 2.3W를 소비한다[5,6]. 그러므로, VLIW CPU를 사용하는 이동용 디바이스를 설계할 때 저전력은 중요한 제약조건이 된다.

잘 설계된 디지털 CMOS 회로에서, 전체 전력 소모 중 90%는 동적 전력 소모에 의해 발생한다[7]. 동적 전력 소모 $P_{dynamic}$ 는 $P_{dynamic} = C_L \cdot N_{sw} \cdot V_{DV}^2 \cdot f_{clk}$ 로 주어지며, 여기서 C_L 은 CMOS 회로의 정전 용량

· 본 논문은 2000년도 한국학술진흥재단의 연구비에 의하여 지원되었음 (KRRF-2000-041-F00287).

[†] 학생회원 : 서울대학교 컴퓨터공학부
sdk@davinci.snu.ac.kr

^{**} 종신회원 : 서울대학교 컴퓨터공학부 교수
jihong@davinci.snu.ac.kr

논문접수 : 2000년 8월 16일

심사완료 : 2001년 7월 27일

(capacitance), N_{SW} 는 클럭 사이클마다의 평균적인 스위칭 활동의 수, V_{DD} 는 공급 전압, 그리고 f_{clk} 는 클럭 주파수를 나타낸다. 일단 마이크로 프로세서가 제작되고 나면 보통 C_L , V_{DD} , 그리고 f_{clk} 는 고정되기 때문에 동적 전압 소모량은 N_{SW} 에 비례하게 된다. 따라서, 많은 설계 수준에서 스위칭 활동을 줄이려는 방법이 제시되었다[8]. 예를 들어, 버스-인버트(bus-invert) 코딩[9]에서는 인버트(invert)라 불리는 추가의 버스라인을 이용하는 데 현재 인코딩된 버스의 값과 다음 데이터 값 사이의 해밍거리(hamming distance)를 계산하여 그 값이 전체 버스 라인 수의 반 이상이 되면 버스 라인을 반전하고 인버트 버스 라인에는 1의 신호를 보내 반전되었음을 알려주도록 하는 방법을 사용하였다. 이 방법을 통해 버스에서의 스위칭 활동을 많이 줄일 수 있다. 레지스터 재명명(relabelling) 기법[10]은 연속해서 접근될 가능성이 많은 레지스터들 사이에 해밍거리(hamming distance)가 작도록 각 명령어내의 레지스터의 번호를 할당하여 명령어 추출과 디코딩 회로에서 스위칭 활동이 줄어들도록 한다.

본 논문에서는 VLIW 프로세서의 명령어 추출 단계에서 스위칭 활동을 줄이는 컴파일러 후단계(post-pass) 최적화 기법을 제시한다. 제시된 기법은 스위칭 활동을 줄이기 위해 VLIW CPU가 명령어¹⁾내에서 같은 연산을 여러 자리에 놓을 수 있다는 인코딩 특징을 이용한다. VLIW CPU에서 하나의 명령어는 일반적으로 여러 개의 연산들을 가지고 있으므로, 명령어 추출동안의 전력 소모는 여러 연산들이 명령어안에서 어떻게 정렬되어 있느냐에 따라 크게 달라진다. 본 논문에서는 연속된 명령어 추출사이의 스위칭 활동이 최소화되도록 연산들의 순서를 수정하여 스위칭 활동을 줄이게 된다.

본 논문의 구성은 다음과 같다. 제안된 연산 재배치 기법을 설명하기 전에 먼저 2장에서는 저전력을 위한 명령어 스케줄링에 관한 이전 연구를 살펴본다. 3장에서는 본 논문에서 사용하는 VLIW 기계 모델을 설명하고 몇 가지 용어들을 정의한다. 4장에서는 단일 기본 블록(basic block)에 적용되는 지역적 연산 재배치 기법을 설명하고 5장에서는 전체 프로그램에 적용 가능한 전역적 연산 재배치 기법이 설명된다. 6장에서는 연산 재배치 기법을 통한 스위칭 활동의 감소를 실험 결과로 보여 주고 7장에서 결론으로 논문을 매듭짓는다.

2. 관련 연구

저전력 프로그램 생성을 위한 명령어 스케줄링에 관해서는 많은 연구가 진행되어왔다[11,12,13,14,15,16].

Su는 [11]에서 제어 경로에서 스위칭 활동을 줄이기 위한 cold scheduling이라는 명령어 스케줄링 기법을 제시하였다. 전통적인 리스트 스케줄링(list scheduling)과 합쳐져, cold scheduling에서는 명령어의 전력 비용을 이용하여 준비 리스트(ready list)안의 명령어를 스케줄한다. 명령어의 전력 비용은 이전에 스케줄된 명령어와의 비트 전환값(bit transition)으로 결정된다. 여기서는 cold scheduling을 그레이 코드 주소(gray code addressing) 기법과 함께 사용하여 스위칭 활동을 20~30%정도 줄였다. 일반적으로 외부 메모리를 접근할 때나 버스를 사용할 때 많은 전력을 소비하기 때문에 시스템 버스에서의 스위칭 활동을 줄이려는 저전력 스케줄링도 연구되었다. Tomiyama는 [15]에서 캐시 적중 실패(miss)가 발생했을 때, 외부 메모리와 내부 캐시사이의 명령어 버스에서의 비트 전환을 줄이는 명령어 스케줄링 기법을 제시했다. 여기서는 프로그램의 제어 및 데이터 의존성(dependency)을 유지하면서 연속적인 두 개의 명령어의 이진 표현값의 비트 변환이 작도록 각각의 기본 블록내의 명령어를 스케줄하게 된다.

언급된 기법들을 비롯해 제시된 많은 저전력을 위한 명령어 스케줄링 기법은 대부분 한 사이클에 하나의 명령어를 실행하는 프로세서를 가정하고 있다. 그러므로, 이러한 기법들은 VLIW와 같은 다중 분기(multi-issue) 프로세서에 직접적으로 사용될 수 없다. VLIW CPU에서는 여러 개의 연산이 하나의 명령어에 들어있기 때문에 두 단계의 스케줄링이 결정되어야 한다. 첫 번째 단계에서는 어떤 연산들을 어떤 명령어에 넣을 것인지를 결정해야 한다. 그리고 두 번째 단계에서는 선택된 연산들을 특정 명령어안에서 어떤 순서로 배치할 것인지가 결정되어야 한다. 본 논문에서 제시되는 기법은 첫 번째 단계의 스케줄링이 이미 결정된 단계에서 VLIW CPU를 위해서 저전력을 위한 두 번째 단계의 스케줄링 문제를 해결한다.

제한적이긴 하나 최근에 이루어진 몇몇 연구에서는 VLIW 프로세서를 위한 저전력 코드 생성 기법에 관한 시도가 있었다. Toburen이 [16]에서 제시한 VLIW 프로세서를 위한 명령어 스케줄링 기법은 그 목표가 최대 전력(peak power)을 줄이고자 하는 데 있다. 이 스케줄링 알고리즘은 현재 명령어의 전력 소모량이 주어진 임계값을 넘지 않도록 연산을 명령어에 추가하는 방법을

1) 본 논문에서 VLIW CPU의 명령어와 연산을 분리하여 사용한다. 하나의 VLIW 명령어는 여러 개의 연산으로 구성되어진다.

사용하고 있다. 이 방법은 최대 전력량을 줄이는 데는 효과적이지만 명령어들 사이나 연산들 사이의 효과에 대해서는 전혀 고려하지 않고 있다. 본 논문의 스케줄링 알고리즘은 명령어안의 연산들을 배치하면서 명령어 사이와 연산들 사이의 스위칭 활동이 최소가 되도록 함으로써 전력소모에 더 좋은 결과를 낳게 된다.

Gebotys는 [17]에서 아키텍처 특징을 이용한 전력 감소기법을 제시했다. 전력 예측 공식과 실제 전력 측정 값을 이용하여 VLIW CPU의 여러 가지 기능 장치(functional unit)의 전력 비용을 계산하고 이 값을 이용하여 컴파일러는 작은 전력 비용을 가지는 기능 장치에 연산을 할당하여 전체 전력 소모량을 줄인다.

위에서 설명한 기법들은 모두 VLIW CPU에서의 첫 번째 스케줄링 문제, 즉 명령어의 순서와 명령어에 포함되는 연산들의 결정 문제를 주 대상으로 하며 두 번째 단계의 스케줄링 문제를 다루는 본 논문의 기법과는 독립적으로 함께 사용할 수 있다.

3. VLIW 기계 모델과 용어 정의

3.1 목표 VLIW 기계 모델

VLIW 아키텍처는 여러 개의 연산을 동시에 수행하기 위해 긴 명령어를 사용한다. 하나의 VLIW 명령어에 여러 개의 연산을 명시하는 데는 일반적으로 비압축 인코딩과 압축 인코딩의 두 가지 방법이 사용된다[18]. 비압축 인코딩을 사용하는 VLIW 기계에서는 명령어내의 연산의 위치가 연산이 사용하는 특정 기능 장치를 결정함을 의미한다. 만약 어떤 기능 장치가 주어진 사이클에 어떤 연산도 수행하지 않도록 스케줄된다면 해당 기능 장치를 위한 연산 자리에는 NOP이 사용된다. 이 인코딩 방법에서는 특정 연산을 수행할 수 있는 기능 장치의 개수에 따라 그 연산이 위치할 수 있는 위치가 제한된다.

반면에 압축 인코딩을 사용하는 VLIW 기계에서는 연산이 사용할 기능장치가 연산에 명시되어 있으므로 연산의 위치가 특정 기능 장치를 의미하지 않으며 메모리 효율성을 높이기 위해서 NOP 연산이 VLIW 명령어에서 사용되지 않는다. 이러한 형태의 VLIW 명령어에서는 연산이 명령어안의 어떤 위치에도 놓일 수 있다.

그림 1과 2는 예제 프로그램 S를 사용하여 두 가지 형태의 인코딩을 비교하고 있다. 프로그램 안에는 3개의 VLIW 명령어가 있으며 각 명령어가 포함하는 연산은 병렬 수행을 의미하는 “||”로 연결되어 있다. 그림 1.(b)와 1.(c)에서 보듯이 비압축 인코딩 방법에서는 연산이 재배치될 수 있는 가능성이 다소 제한된다. 예를 들어,

첫 번째 명령어에서는 IADD와 NOP, FADD와 NOP, 그리고 LOAD와 STORE만이 서로 위치를 바꿀 수 있다. 이에 반해 압축 인코딩기법에서는 그림 2.(b)와 2.(c)에서 보듯이 연산 재배치의 가능성이 훨씬 더 많은 데 이것은 연산의 위치와 사용할 기능장치가 무관하기 때문이다. 예를 들어, 프로그램 S의 첫 번째 VLIW 명령어에 대해 4!개의 서로 다른 연산 배치가 가능하다. 그림 2.(b)와 2.(c)에서 같은 VLIW 명령어안의 병렬로 실행되는 연산들은 병렬 비트로 표시되어있다(진하게 표시된 부분). 만약 어떤 연산의 병렬비트가 1이면, 그 연산은 다음 연산과 병렬로 함께 실행된다. 0일 경우에는 다음 연산은 현재의 연산이 실행된 뒤에 다음 사이클에 실행된다. 본 논문에서 제시하는 연산 재배치 기법은 압축/비압축 인코딩을 사용하는 VLIW CPU에서 모두 사용 가능하지만 간략한 설명을 위해 본 논문에서는 목표 VLIW CPU가 압축 인코딩 기법을 사용한다고 가정한다.

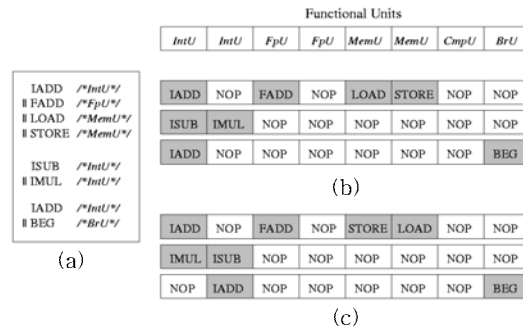


그림 1 비압축 VLIW 명령어 인코딩; (a) 예제 프로그램 S, (b) 비압축 인코딩으로 표시된 S, (c) S의 또 다른 비압축 인코딩

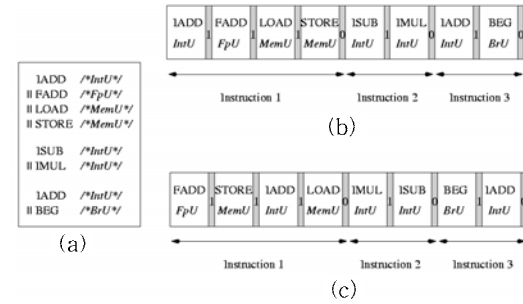


그림 2 압축 VLIW 명령어 인코딩; (a) 예제 프로그램 S, (b) 압축 인코딩으로 표시된 S, (c) S의 또 다른 압축 인코딩

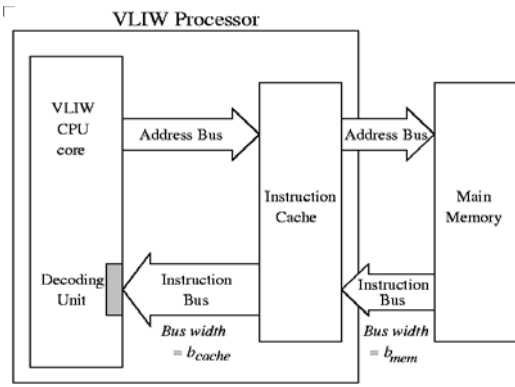


그림 3 목표 시스템 구조

또한 본 논문에서는 목표 시스템이 그림 3과 같은 시스템 구성을 가졌다고 가정한다. VLIW 프로세서는 칩 내장(on-chip) 명령어 캐쉬를 가지고 있으며 VLIW 명령어는 캐쉬로부터 b_{cache} 비트의 폭을 가진 버스를 통해 추출된다. 만약 명령어가 칩 내장 캐쉬에 없으면 명령어를 가진 메모리 블록이 외부 메모리로부터 b_{mem} 비트의 폭을 가진 버스를 통해 캐쉬로 추출된다. 압축 인코딩 방식을 사용하기 때문에 캐쉬로부터 한번의 추출에 여러 개의 명령어가 함께 추출된다. 본 논문에서는 한번의 명령어 추출에서 함께 추출되는 명령어들의 집합을 페치 패킷(fetch packet)이라고 정의한다. 간략한 설명을 위해 본 논문에서는 목표 시스템에 대해 다음의 추가적인 가정을 한다:

- b_{cache} 비트의 길이를 가지는 단일 페치 패킷에는 N 개의 연산이 포함되어 있다. (즉, 연산 하나의 크기는 b_{cache}/N 이다.)
- 단일 페치 패킷내에 있는 명령어들은 페치 패킷의 경계선을 넘지 않는다. (즉, 단일 명령어가 두 개 이상의 페치 패킷에 걸쳐 위치하는 경우는 없다.)
- b_{mem} 는 연산의 크기와 같다. (즉, $b_{mem} = b_{cache}/N$.)
- 외부 명령어 버스가 사용되지 않을 때는 각각의 버스 라인은 하이-임피던스(high impedance) 상태를 막기 위해 논리 값 1을 유지한다.

3.2 용어 정의

연산 재배치 기법의 설명을 위해서 다음의 용어들을 정의한다:

정의 1 순열 $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$ 을 연산 재배치 함수라고 정의한다.

정의 2 두 개의 VLIW 명령어 $I_1 = (OP_1^1, OP_2^1, \dots, OP_n^1)$

와 $I_2 = (OP_1^2, OP_2^2, \dots, OP_n^2)$ 에 대해 모든 $1 \leq i \leq n$ 에 대해서 $OP_{\sigma(i)}^1 = OP_i^2$ 를 만족하는 연산 재배치 함수 σ 가 존재하면 I_1 과 I_2 는 서로 연산 재배치에 의해 동치(equivalent)라고 한다.

정의 3 두 개의 페치 패킷 $FP_1 = (I_1^1, I_2^1, \dots, I_n^1)$ 과 $FP_2 = (I_1^2, I_2^2, \dots, I_n^2)$ 에 대해 모든 $1 \leq i \leq n$ 에 대해서 I_i^1 와 I_i^2 가 σ_i 에 의해서 동치가 되는 연산 재배치 함수들 $(\sigma_1, \sigma_2, \dots, \sigma_n)$ 가 존재하면 FP_1 과 FP_2 는 연산 재배치에 의해 서로 동치라고 한다. 그리고 $EQ(FP)$ 는 주어진 페치 패킷 FP 와 연산 재배치에 의해 동치인 페치 패킷들의 집합을 의미한다.

정의 4 두 개의 기본 블록 $bb_1 = (FP_1^1, FP_2^1, \dots, FP_n^1)$ 과 $bb_2 = (FP_1^2, FP_2^2, \dots, FP_n^2)$ 에 대해 모든 $1 \leq i \leq n$ 에 대해서 FP_i^1 와 FP_i^2 가 연산 재배치에 의해 동치이면 bb_1 과 bb_2 는 서로 연산 재배치에 의해 동치라고 한다. 그리고 $EQ(bb)$ 는 주어진 기본 블록 bb 와 연산 재배치에 의해 동치인 기본 블록들의 집합을 의미한다.

정의 5 두 개의 프로그램 $S_1 = (bb_1^1, bb_2^1, \dots, bb_n^1)$ 과 $S_2 = (bb_1^2, bb_2^2, \dots, bb_n^2)$ 에 대해 모든 $1 \leq i \leq n$ 에 대해서 bb_i^1 와 bb_i^2 가 연산 재배치에 의해 동치이면 S_1 과 S_2 는 서로 연산 재배치에 의해 동치라고 한다. 그리고 $EQ(S)$ 는 주어진 프로그램 S 와 연산 재배치에 의해 동치인 프로그램들의 집합을 의미한다.

본 논문의 나머지 장들에서는 혼동이 없는 한 “동치”라는 표현은 “연산 재배치에 의해 동치”라는 의미로 사용한다.

4. 지역적 연산 재배치 문제

연산 재배치 문제에 대한 이해를 돕고 재배치 문제의 해법에 사용되는 기본 모듈의 설명을 위해 본 장에서는 지역적 연산 재배치(Local Operation Rearrangement, LOR) 문제라고 하는 간단한 재배치 문제를 먼저 다룬다. LOR 문제에서는 각각의 기본 블록들이 독립적으로 고려되며 각 기본 블록들은 메모리로부터 추출되어 정확하게 한번 실행된다고 가정한다. 기본 블록은 주 메모리로부터 추출되기 때문에 명령어 추출동안에는 관련된 캐쉬 적중 실패(miss)가 발생한다. 전역적 연산 재배치(Global Operation Rearrangement, GOR) 문제라고 하는 완전한 재배치 문제는 다음 장에서 논의된다. GOR 문제에서는 프로그램내의 모든 기본 블록이 동시에 고려된다.

4.1 기본 개념

일반적으로 스위칭 활동은 비트 전환의 횟수에 비례하기 때문에 목표 시스템에서 명령어 추출동안에 스위

칭 활동을 줄이기 위해서는 연속된 명령어 추출 사이의 비트 전환의 수를 줄여야 한다. 압축 인코딩을 사용하는 VLIW 기계에서는 연산들이 명령어 경계내의 어느 위치든지 놓일 수 있기 때문에 주어진 VLIW 명령어를 더 작은 스위칭 활동을 가지며 동치인 다른 명령어로 재배치함으로써 연속된 명령어 추출 사이의 비트 전환의 수를 줄일 수 있다. 그림 4의 예를 살펴보자. 32비트(즉, $b_{cache}=32$)로 구성된 4개의 페치 패킷이 있으며 각각의 페치 패킷은 하나의 명령어로 되어 있으며 각 명령어는 4개의 연산들로 이루어져 있다. 그림 4.(b)는 버스에서의 비트 전환이 줄어들도록 연산들이 재배치된 후의 명령어들을 보여주고 있다. 재배치된 명령어는 원래의 명령어와 같은 의미를 가지면서도 전체 비트 전환을 39에서 29로 25%가량 줄인다.

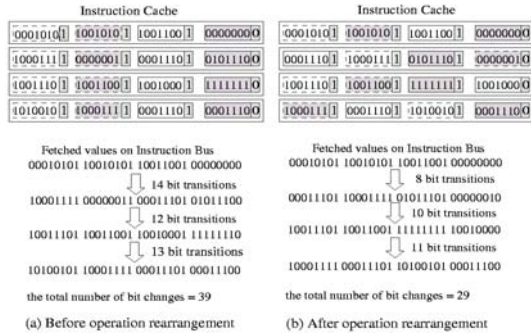


그림 4 연산 재배치 예제

4.2 LOR 문제 정의

만약에 주어진 기본 블록이 그림 3에서 보여준 목표 구조에서 단지 한번만 실행된다면, 명령어 추출동안의 비트 전환의 수인 SW^B 는 SW^B_{cache} 와 SW^B_{mem} 의 합으로

표현된다. SW^B_{cache} 는 내부 버스에서의 비트 전환의 수를 의미하며 SW^B_{mem} 는 외부 버스에서의 비트 전환의 수를 의미한다. 표 1에서 설명된 기호를 사용하여, SW^B_{cache} 와 SW^B_{mem} 는 다음과 같이 계산된다.

SW^B_{cache} 는 명령어 캐쉬로부터 페치 패킷들이 연속적으로 추출될 때 발생하는 비트 전환의 합을 의미하므로 다음과 같이 계산된다:

$$SW^B_{cache} = \sum_{i=1}^{N_{fb}(B)-1} d_{fb}(FP_i^B, FP_{i+1}^B). \quad (1)$$

SW^B_{mem} 는 주 메모리로부터 인접한 연산들이 추출될 때 발생하는 비트 전환의 합이다. 3.1절에서 b_{mem} 는 b_{cache}/N_{op} 와 같다고 가정하였고 각각의 메모리 블록마다 한번의 캐쉬 적중 실패가 있으며 기본 블록들은 캐쉬 블록 크기에 정렬되어 있다고 가정하면 SW^B_{mem} 는 다음과 같이 계산된다:

$$SW^B_{mem} = \sum_{i=1}^{N_{fb}(B)} \sum_{n=1}^{N_{op}-1} d_{op}(OP_n^{FP_i^B}, OP_{n+1}^{FP_i^B}) + \sum_{i=1}^{N_{fb}(B)-1} d_{op}(OP_{N_{op}}^{FP_i^B}, OP_1^{FP_{i+1}^B}) + d_{op}(1, OP_1^{FP_1^B}) + d_{op}(OP_{N_{op}}^{FP_{N_{fb}(B)}^B}, 1). \quad (2)$$

내부 버스의 정전 용량에 대한 외부 버스의 정전 용량의 상대 값을 α 라고 하면, SW^B 는 식 (1)과 (2)를 사용하여 다음과 같이 계산된다:

$$SW^B = SW^B_{cache} + \alpha \cdot SW^B_{mem} = \sum_{i=1}^{N_{fb}(B)-1} SW^B_{fb}(FP_i^B, FP_{i+1}^B) + \sum_{i=1}^{N_{fb}(B)} SW^B_{op}(FP_i^B) \quad (3)$$

여기서

$$SW^B_{fb}(FP_i^B, FP_{i+1}^B) = d_{fb}(FP_i^B, FP_{i+1}^B) + \alpha \cdot d_{op}(OP_{N_{op}}^{FP_i^B}, OP_1^{FP_{i+1}^B}) \quad (4)$$

$$SW^B_{op}(FP_i^B) = \begin{cases} \alpha \cdot d_{op}(1, OP_1^{FP_i^B}) + S_{op} & \text{if } i=1 \\ \alpha \cdot d_{op}(OP_{N_{op}}^{FP_i^B}, 1) + S_{op} & \text{if } i=N_{fb}(B) \\ S_{op} & \text{otherwise} \end{cases} \quad (5)$$

(where $S_{op} = \alpha \cdot \sum_{n=1}^{N_{op}-1} d_{op}(OP_n^{FP_i^B}, OP_{n+1}^{FP_i^B})$).

표 1 4.2절에서 사용되는 기호들

기 호	의 미
$N_{fb}(B)$	기본 블록 B에 있는 페치 패킷들의 개수.
N_{op}	하나의 페치 패킷에 있는 연산의 개수. (이 값은 B에 상관없이 고정되어 있다.)
1	모든 비트의 값이 1이고 길이가 b_{mem} 인 비트 벡터.
FP_i^B	기본 블록 B의 i번째 페치 패킷.
$OP_n^{FP_i^B}$	페치 패킷 FP_i^B 의 n번째 연산. (FP_i^B 의 첫 번째 연산은 $OP_1^{FP_i^B}$ 이며 마지막 연산은 $OP_{N_{op}}^{FP_i^B}$ 이다.)
$d_{fb}(FP_i^B, FP_j^B)$	페치 패킷 FP_i^B 와 FP_j^B 사이의 해밍 거리(hamming distance).
$d_{op}(OP_n^{FP_i^B}, OP_m^{FP_j^B})$	연산 $OP_n^{FP_i^B}$ 와 $OP_m^{FP_j^B}$ 사이의 해밍 거리(hamming distance).

LOR 문제는 주어진 기본 블록 B 에 대해 모든 $B' \in EQ(B)$ 에 대해서 $SW^B \leq SW^{B'}$ 를 만족하며 B 와 동치인 기본 블록 B' 를 찾는 것으로 정의된다. 만약에 연산들이 재배치되면 식 (4)와 (5)에서 $d_{fp}(FP_i^B, FP_{i+1}^B)$, $d_{op}(OP_{N_{op}^{I_j^{FP_i^B}}}, OP_1^{I_j^{FP_{i+1}^B}})$, 그리고 $d_{op}(OP_n^{I_j^{FP_i^B}}, OP_{n+1}^{I_j^{FP_{i+1}^B}})$ 의 값이 바뀌게 되고 이들 값의 변화를 통하여 최소의 SW^B 을 가지는 B' 를 찾는 것이 LOR 문제의 목적이다.

4.3 LOR 문제를 위한 최적해법

LOR 문제의 최적해는 LOR 문제를 START와 END라는 두 노드(node)사이의 최단경로 문제로 전환하여 구하게 된다. 표 2에서 설명된 기호를 사용하여, 기본 블록 B 가 주어졌을 때 방향성 그래프(directed graph) $G_B = \{V, E, W_{node}, W_{edge}\}$ 을 만든다. 여기서,

$$\begin{aligned}
 V &= \{START, END\} \cup \bigcup_{i=1}^{N_{fb}(B)} EQ(FP_i^B) \\
 &= \{START, END\} \cup \bigcup_{i=1}^{N_{fb}(B)} \{FP_{i,1}^B, \dots, FP_{i,N_{op}(FP_i^B)}^B\}, \\
 E &= \{(v, w) | v = START, w \in EQ(FP_1^B)\} \cup \\
 &\quad \{(v, w) | w = END, v \in EQ(FP_{N_{fb}(B)}^B)\} \cup \\
 &\quad \{(v, w) | v \in EQ(FP_i^B), w \in EQ(FP_{i+1}^B) \\
 &\quad \text{for } 1 \leq i < N_{fb}(B)\}, \\
 W_{node}(v) &= \begin{cases} SW_{min}^{FP_i^B}(v) & \text{if } v \in V - \{START, END\} \\ 0 & \text{otherwise} \end{cases} \\
 W_{edge}(v, w) &= \begin{cases} SW_{min}^{FP_i^B}(v, w) & \text{if } v, w \in V - \{START, END\} \\ 0 & \text{otherwise} \end{cases}
 \end{aligned}$$

그림 5는 LOR 문제를 최단거리 문제로 변환함으로써 만들어진 그래프의 예를 보여주고 있다. 패치 패킷 FP_i^B 에 대해서 그래프 G_B 에 $N_{op}(FP_i^B)$ 개의 노드가 생성되며, 연속적인 패치 패킷 FP_i^B 와 FP_{i+1}^B 에 대하여 모든 $(FP_{i,k}^B, FP_{i+1,l}^B)$ 의 쌍에 간선(edge)이 연결된다. 패치 패킷 FP_i^B 로부터 만들어진 $N_{op}(FP_i^B)$ 개의 노드들은 모두 레벨 i 에 있다고 부른다. 그래프 G_B 에서 경로 $P = (START, v_1, \dots, v_k, END)$ 의 거리는 $\sum_{i=1}^k W_{node}(v_i) + \sum_{i=1}^{k-1} W_{edge}(v_i, v_{i+1})$ 로 주어진다. 경로 P 의 거리는 모

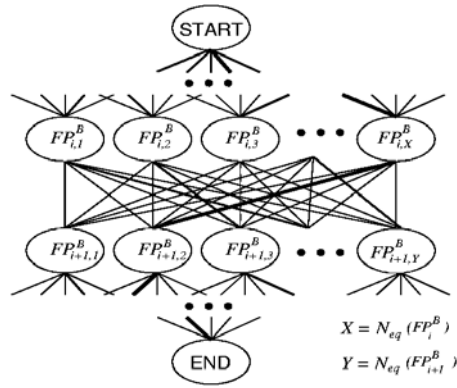


그림 5 LOR 문제의 최단거리 문제로의 변형

든 $1 \leq i < k$ 에 대해서 패치 패킷 FP_i^B 가 v_i 로 재배치되었을 때의 SW^B 와 같은 값이 된다.

이와 같이 만들어진 최단경로 문제의 최적 해는 그림 6에 보이는 바와 같은 수정된 최단경로 탐색 알고리즘으로 구해진다. 이 알고리즘은 다음의 정리를 바탕으로 두고 있다. 본 정리는 간단히 증명될 수 있으므로 본 논문에서는 증명을 생략한다.

정리 1 경로 $P(FP_{i,j}^B) = (START, v_1, \dots, v_{i-1}, FP_{i,j}^B)$ 를 START로부터 $FP_{i,j}^B \in EQ(FP_i^B)$ 까지의 최단 경로로 정의하고 그 경로의 거리를 $d_{P(FP_{i,j}^B)}$ 라고 하자. 그러면 경로 $P(FP_{i+1,k}^B) = (START, v_1, \dots, v_i, FP_{i+1,k}^B)$ 의 최단 거리인 $d_{P(FP_{i+1,k}^B)}$ 는 $\min_{1 \leq j \leq N_{op}(FP_i^B)} [d_{P(FP_{i,j}^B)} + W_{edge}(FP_{i,j}^B, FP_{i+1,k}^B) + W_{node}(FP_{i+1,k}^B)]$ 로 주어진다.

그림 6에서 SW_{min} 은 START로부터 $FP_{i+1,k}^B$ 까지의 경로의 최단거리를 저장하기 위한 변수이며(15행) SW_{cur} 는 START로부터 $FP_{i+1,k}^B$ 까지의 경로 중 $FP_{i,j}^B$ 를 통과하는 경로의 최단거리를 저장하기 위한 변수이다. 최종적으로 구해진 최단 경로는 MinPath를

표 2 4.3절에서 사용되는 기호들

기 호	의 미
$N_{ins}(FP_i^B)$	FP_i^B 에 있는 명령어의 개수.
$I_j^{FP_i^B}$	FP_i^B 의 j 번째 명령어 ($1 \leq j \leq N_{ins}(FP_i^B)$).
$N_{op}(I_j^{FP_i^B})$	$I_j^{FP_i^B}$ 에 있는 연산의 개수.
$N_{eq}(I_j^{FP_i^B})$	$I_j^{FP_i^B}$ 와 동치인 명령어들의 개수 ($N_{eq}(I_j^{FP_i^B}) = (N_{op}(I_j^{FP_i^B}))!$).
$N_{op}(FP_i^B)$	FP_i^B 의 동치인 패치 패킷의 개수 ($N_{op}(FP_i^B) = \prod_{j=1}^{N_{ins}(FP_i^B)} N_{eq}(I_j^{FP_i^B})$).
$FP_{i,n}^B$	$EQ(FP_i^B)$ 안의 n 번째 패치 패킷 ($1 \leq n \leq N_{op}(FP_i^B)$).

```

1: for i ← 0 to Nfp(B) {
2:   /* for each vertex in the level i + 1 */
3:   for k ← 1 to Neq(FPi+1B) {
4:     SWmin := ∞;
5:     /* for each vertex in the level i */
6:     for j ← 1 to Neq(FPiB) {
7:       SWcur := dp(FPi,jB) + Wedge(FPi,jB, FPi+1,kB)
8:         + Wnode(FPi+1,kB);
9:       /* find the minimum value */
10:      if (SWmin > SWcur) {
11:        SWmin := SWcur;
12:        MinNode := j;
13:      }
14:    }
15:    dp(FPi+1,kB) := SWmin;
16:    /* store MinNode for the final path construction */
17:    MinPath[FPi+1,kB] := FPi,MinNodeB;
18:  }
19: }
    
```

그림 6 최단경로 탐색 알고리즘

거꾸로 따라가면서 구할 수 있다. 수정된 최단 경로 탐색 알고리즘의 복잡도는 $O(N_{fp}(B) \cdot (N_{eq}^{fp})^2)$ 이다. 여기서 $N_{eq}^{fp} = \frac{1}{N_{fp}(B)} \sum_{i=1}^{N_{fp}(B)} N_{eq}(FP_i^B)$ 로 주어지며 N_{eq}^{fp} 의 최대값은 N_{eq} 이다.

5. 전역적 연산 재배치 문제

GOR 문제에서는 전역적인 최적 해를 찾기 위해서 프로그램 안의 모든 기본 블록이 동시에 고려된다. LOR 문제는 기본 블록 사이의 스위칭 활동은 고려하지 않기에 각각의 기본 블록에 대해서만 LOR 문제를 푸는 것은 전체 프로그램의 비트 전환을 항상 최소화시키지는 않는다. GOR 문제의 최적 해를 구하기 위해서는 프로그램 수행의 동적인 성질들에 대한 정보가 필요하다. 예를 들어, 각각의 기본 블록이 몇 번 수행되는 지, 캐시 적중 실패는 얼마나 자주 발생하는 지, 그리고 기본 블록들이 서로 어떻게 연관되어 있는 지 등을 미리 알아야 한다.

5.1 GOR 문제 정의

프로그램 S가 기본 블록 $bb_1, bb_2, \dots, bb_{N_m(S)}$ 들로 이루어

어져 있다고 하면, 프로그램 수행중에 명령어 추출로부터 발생하는 전체 비트 전환값인 SW^S 는 표 3에서 설명된 기호를 사용하여 다음과 같이 나타낼 수 있다:

$$SW^S = \sum_{i \in N_{eq}(S)} \sum_{j \in N_{eq}(S)} SW_{fp}^{inter}(bb_i, bb_j) + \sum_{i \in N_{eq}(S)} SW_{fp}^{intra}(bb_i) \tag{6}$$

여기서

$$SW_{fp}^{inter}(bb_i, bb_j) = u(bb_i, bb_j) \cdot SW_{fp}^{inter}(FP_{N_{fp}(bb_i)}^{bb_i}, FP_1^{bb_j}) \tag{7}$$

$$SW_{fp}^{intra}(bb_i) = u(bb_i) \cdot \left(\sum_{k=1}^{N_{fp}(bb_i)-1} SW_{fp}^{inter}(FP_k^{bb_i}, FP_{k+1}^{bb_i}) \right) + \sum_{k=1}^{N_{fp}(bb_i)} SW_{fp}^{intra}(FP_k^{bb_i}) \tag{8}$$

$$SW_{fp}^{inter}(FP_n^{bb_i}, FP_m^{bb_j}) = d_p(FP_n^{bb_i}, FP_m^{bb_j}) \tag{9}$$

$$SW_{fp}^{intra}(FP_n^{bb_i}) = \left\{ \sum_{k=1}^{N_{fp}(bb_i)-1} \sum_{l=k+1}^{N_{fp}(bb_i)} d_{op}(OP_k^{fp^{bb_i}}, OP_{l-1}^{fp^{bb_i}}) + \sum_{k=1}^{N_{fp}(bb_i)-1} d_{op}(OP_{N_{fp}}^{fp^{bb_i}}, OP_1^{fp^{bb_i}}) + d_{op}(1, OP_1^{fp^{mb}}) + d_{op}(OP_{N_{fp}}^{fp^{mb}}, 1) \right\} \cdot \alpha \cdot R_{miss}^{mb} \text{ where } mb = MB(FP_n^{bb_i}). \tag{10}$$

로 구해진다. 식 (7), (8), 그리고 (10)에서 $u(bb_i, bb_j)$, $u(bb_i)$ 그리고 R_{miss}^{mb} 는 프로그램의 실행 트레이스(trace)를 분석함으로써 구할 수 있다. 페치 패킷 F에 대해서 캐시 적중 실패가 발생하면, F를 포함하고 있는 메모리 블록 안의 모든 페치 패킷들이 주 메모리로부터 추출된다. 그러므로 식 (10)에서 $MB(FP_n^{bb_i})$ 안에 있는 모든 페치 패킷들이 외부 버스에서의 비트 전환을 계산하는 데 사용된다. 각각의 기본 블록들은 캐시 메모리 블록 크기에 대해 정렬되어 있다고 가정한다. 식 (9)에서 $FP_n^{bb_i}$ 의 마지막 연산과 $FP_m^{bb_j}$ 의 처음 연산간의 해밍 거리는 빠져있는 데, 이것은 식 (10)에 포함되어 있다. 주어진 프로그램 S에 대해서, GOR 문제는 모든 $S'' \in EQ(S)$ 에 대해서 $SW^S \leq SW^{S''}$ 를 만족하는 S와 동

표 3 5.1절에서 사용되는 기호들

기 호	의 미
$N_{fp}(S)$	프로그램 S안의 기본 블록의 개수.
$u(bb_i, bb_j)$	기본 블록 bb_j 가 기본 블록 bb_i 직후에 실행되는 횟수의 기대값.
$u(bb_i)$	기본 블록 bb_i 가 실행되는 횟수의 기대값.
$MB(FP_n^{bb_i})$	페치 패킷 $FP_n^{bb_i}$ 를 포함하고 있는 메모리 블록.
$FP_j^{MB(FP_n^{bb_i})}$	페치 패킷 $FP_n^{bb_i}$ 를 포함하는 메모리 블록내의 j번째 페치 패킷.
$R_{miss}^{MB(FP_n^{bb_i})}$	메모리 블록 $MB(FP_n^{bb_i})$ 의 캐시 적중 실패율.

표 4 5.2절에서 사용되는 기호들

기 호	의 미
$N_{\alpha}(bb_i^S)$	bb_i^S 와 동치인 기본 블록들의 개수 ($N_{\alpha}(bb_i^S) = \prod_{j=1}^{N_{\beta}(bb_i^S)} N_{\alpha}(FP_j^{bb_i^S})$).
$bb_{i,n}^S$	$EQ(bb_i^S)$ 안에서 n 번째 기본 블록 ($1 \leq n \leq N_{\alpha}(bb_i^S)$).

치인 프로그램 S 를 찾는 것이다.

5.2 GOR 문제를 위한 최적해법

GOR 문제도 역시 LOR 문제처럼 최단 경로 문제로 변환하여 풀 수 있다. LOR 문제와의 가장 큰 차이는 프로그램은 일반적으로 분기문과 반복문으로 구성되어 있으므로 만들어진 그래프도 하나의 노드로부터 여러 개의 경로가 있다는 것이다. LOR 문제에서 사용한 최단 경로 탐색 알고리즘을 GOR에서도 사용하기 위해 생성된 그래프를 분기문이나 반복문이 없도록 변환한다.

표 4에서 설명된 기호들을 사용하여 프로그램 S 가 주어졌을 때, 그래프 $G_S = \{V, E, W_{make}, W_{take}\}$ 를 만든다. 여기서

$$\begin{aligned}
 V &= \{\text{START}, \text{END}\} \cup \bigcup_{i=1}^{N_{\alpha}(S)} EQ(bb_i^S) \\
 &= \{\text{START}, \text{END}\} \cup \bigcup_{i=1}^{N_{\alpha}(S)} \{bb_{i,1}^S, \dots, bb_{i,N_{\alpha}(bb_i^S)}^S\}, \\
 E &= \{(v, w) | v = \text{START}, w \in EQ(bb_{i,m}^S)\} \cup \\
 &\quad \{(v, w) | w = \text{END}, v \in \bigcup_{i \in \text{Exit}^S} EQ(bb_i^S)\} \cup \\
 &\quad \{(v, w) | v \in EQ(bb_i^S), w \in EQ(bb_j^S) \text{ for } 1 \leq i, j \leq N_{\alpha}(S) \\
 &\quad \text{where } bb_j^S \text{ is an immediate} \\
 &\quad \text{successor of } bb_i^S \text{ in a control flow graph}\}, \\
 W_{make}(v) &= \begin{cases} SW_{IB}^{intra}(v) & \text{if } v \in V - \{\text{START}, \text{END}\}, \\ 0 & \text{otherwise} \end{cases}, \\
 W_{take}(v, w) &= \begin{cases} SW_{IB}^{inter}(v, w) & \text{if } v, w \in V - \{\text{START}, \text{END}\} \\ 0 & \text{otherwise.} \end{cases}
 \end{aligned}$$

이다. 여기서 $bb_{i,m}^S$ 는 프로그램 S 의 진입점이며 $Exit^S$ 는 프로그램 S 의 출구지점에 해당하는 기본 블록들의 인덱스를 가지고 있다.

GOR 문제의 계산 복잡도를 줄이기 위해서 $bb_{i,j}^S$ 와 $bb_{i,k}^S$ ($j \neq k$) 사이에 다음 조건이 만족되면 $bb_{i,k}^S$ 를 그래프 G_S 에서 생략한다:

$$\begin{aligned}
 FP_1^{bb_{i,j}^S} &= FP_1^{bb_{i,k}^S}, FP_{N_{\beta}(bb_{i,j}^S)}^{bb_{i,j}^S} = FP_{N_{\beta}(bb_{i,k}^S)}^{bb_{i,k}^S}, \\
 \text{and } SW_{IB}^{intra}(bb_{i,j}^S) &\leq SW_{IB}^{intra}(bb_{i,k}^S). \tag{11}
 \end{aligned}$$

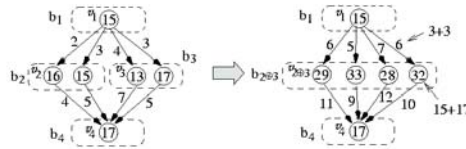
만약 $bb_{i,k}^S$ 가 식 (11)을 만족하면, $bb_{i,k}^S$ 는 GOR의 최적해의 일부가 될 수 없는 데 그 이유는 $bb_{i,j}^S$ 와 $bb_{i,k}^S$ 가 같은 SW_{IB}^{intra} 값을 가지기 때문이다. 각각의 기본 블록 bb_i^S 에 대해서, 수정된 LOR 알고리즘(첫 번째와 마지막 페치

패킷을 고정시킨 채)을 $N_{\alpha}(FP_1^{bb_i^S}) \times N_{\alpha}(FP_{N_{\beta}(bb_i^S)}^{bb_i^S})$ 번 적용시킴으로서 불필요한 $bb_{i,k}^S$ 가 없는 단순화 된 G_S 를 구할 수 있다. G_S 에서 불필요한 $bb_{i,k}^S$ 를 없앴으로서, $N_{\alpha}(bb_i^S)$ 는 실질적으로 $N_{\alpha}(FP_1^{bb_i^S}) \times N_{\alpha}(FP_{N_{\beta}(bb_i^S)}^{bb_i^S})$ 로 줄어든다.(논문의 나머지에서 $N_{\alpha}(FP_1^{bb_i^S}) \times N_{\alpha}(FP_{N_{\beta}(bb_i^S)}^{bb_i^S})$ 를 간단히 N_i 로 표기한다.)

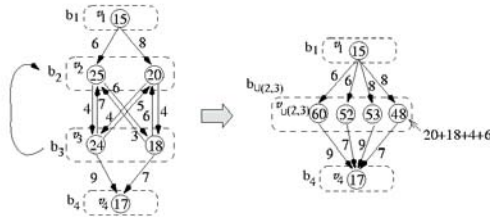
일단 단순화된 그래프 G_S 가 만들어지고 나면 여기에 LOR에서 사용한 수정된 최단 경로 알고리즘을 적용하기 위해서 분기문과 반복문이 없도록 변환된다. 그림 7은 그래프 G_S 에서 분기문 합병과 반복문 롤링 기법을 통해 분기문과 반복문을 변환하는 과정을 보여주고 있다.

분기문 합병은 분기문의 두 개의 후속 노드인 v_i 와 v_j 를 새로운 노드 $v_{\oplus ij}$ 로 대체한다.

새로운 노드 $v_{\oplus ij}$ 에 대해서 $N_{\alpha}(bb_{\oplus ij})$ 는 $N_{\alpha}(bb_i) \times N_{\alpha}(bb_j)$ 의 값을 가진다. 예를 들어, 그림 7.(a)에서 기본 블록 b_2 와 b_3 는 각각 2개의 동치인 기본 블록들을 가지고 있다. 분기문 합병이 적용된 후, $v_{2\oplus 3}$ 는 4개의 동치인 기본 블록을 가지게 된다. 노드 v_2 ($W_{make}(v_2) = 16$)와 v_3 ($W_{make}(v_3) = 13$)는 W_{make} 값이 $29 (= 16 + 13)$ 인 $v_{2\oplus 3}$ 로 합병된다. $v_{2\oplus 3}$ 는 v_1 과 간선(edge)을 가지며 간선의 W_{take} 값은 $6 (= 2 + 4)$ 이다.



(a) branch merging



(b) loop rolling

그림 7 분기문 합병과 반복문 롤링의 효과

그림 7.(a)에서 분기문 합병이 끝난 후에, 3개의 기본 블록 b_1 , $b_{2\&3}$, 그리고 b_4 는 LOR 알고리즘에 의해 하나의 기본 블록으로 다시 합쳐진다. 이 노드의 W_{mix} 값은 78이 된다. 이 과정을 순차적 합병이라고 한다. 만약 기본 블록 b_1, \dots, b_j 가 순차적 합병에 의해 하나의 기본 블록으로 되면, 합병된 기본 블록은 $N_{ar}(FP_1^{bb}) \times N_{ar}(FP_{N_{ar}(bb)})$ 개의 동치인 노드를 가지게 된다.

반복문 롤링(rolling)도 순차적 합병과 비슷한 방식으로 작동한다. 이것은 반복문의 몸체부분인 v_1, \dots, v_j 를 새로운 노드 $v_{U(i, \dots, j)}$ 로 합병한다. 차이점은 반복문 롤링은 합병된 노드의 값을 계산할 때 백-간선(back edge)의 값을 함께 더하게 된다. 예를 들어, 그림 7.(b)에서 각각 2개의 동치인 블록을 가지는 b_2 와 b_3 를 살펴보자. 노드 v_2 와 v_3 는 합쳐져 새로운 노드 $v_{U(2,3)}$ 가 되며 이 노드의 W_{mix} 값은 $60 (= 25 + 24 + 4 + 7)$ 이 된다. 노드 $v_{U(2,3)}$ 는 노드 v_1 과 간선을 가지고 있으며 간선의 값은 $W_{back}(v_1, v_2)$ 의 값인 6이 된다.

일단 G_S 가 분기문과 반복문이 없는 그래프로 바뀌고 나면, LOR 문제에서 사용되었던 최단 경로 알고리즘을 사용하여 최적의 해를 구할 수 있다.

5.3 GOR를 위한 경험적 기법

앞 절에서 제시된 방법으로 그래프 G_S 를 만들어 최적해를 구하는 방법은 실제로 많은 메모리와 시간을 요구한다. 예를 들어, 각각의 기본 블록 b_i 에 대해서 N_i 개의 노드 구조가 만들어져야 한다. 더구나, 두 개의 기본 블록 b_i 와 b_j 가 분기문 합병을 통해 합쳐질 때, 합병된 노드를 위해 가져야 하는 노드 구조의 개수는 $N_i \times N_j$ 개로 늘어난다. 이 절에서는 GOR-H 알고리즘이라고 부르는 GOR 문제에 대한 경험적 기법(heuristic)을 제시한다.

GOR-H는 메모리 요구량과 계산 시간을 줄이기 위해서 두 가지 휴리스틱 규칙을 사용한다. 첫째로, 모든 기본 블록이 평등하게 처리되지 않는다. 각각의 기본 블록 bb_i 에 대해서 다음과 같이 정의되는 $FR(bb_i)$ 를 계산한다.

$$FR(bb_i) = \frac{u(bb_i) \cdot N_{fb}(bb_i)}{\sum_{j=1}^{N_{ar}(S)} u(bb_j) \cdot N_{fb}(bb_j)}$$

$FR(bb_i)$ 는 프로그램의 전체 기본 블록들에 대해서 기본 블록 bb_i 안에 있는 페치 패킷의 유효한 추출 비율을 나타낸다. 큰 $FR(bb_i)$ 를 가진 기본 블록이 명령어 추출 단계에서 전체 스위칭 활동에도 큰 영향을 가지기 때문에, 큰 $FR(bb_i)$ 값을 가진 기본 블록들을 작은 $FR(bb_i)$ 값을 가진 기본 블록들보다 좀 더 철저히 재배치한다. 이것은 작은 $FR(bb_i)$ 값을 가진 기본 블록보다 이전에

큰 $FR(bb_i)$ 값을 가진 기본 블록이 먼저 재배치되며 만약 재배치하려는 기본 블록의 바로 다음이나 이전에 실행되는 기본 블록이 이미 재배치되어 있으면 재배치과정이 이미 재배치된 기본 블록의 처음 페치 패킷이나 마지막 페치 패킷에 의해서 영향을 받는다는 것을 의미한다.

두 번째로, 각각의 기본 블록 bb_i 에 대해서 최적해를 찾기 위해서 $EQ(bb_i)$ 안의 모든 동치인 기본 블록들이 검사되지 않는다. 단지 N_{cmd} 개의 동치인 기본 블록들만이 생성되고 그래프 G_S 안에 포함된다. 이들 N_{cmd} 개의 동치인 기본 블록들은 $EQ(bb_i)$ 안의 모든 기본 블록들 중에서 N_{cmd} 번째까지 작은 스위칭 활동값을 가진 것들이다.

일단 그래프 G_S 가 위의 두 가지 규칙에 의해서 만들어지고 나면, 나머지 처리단계(분기문 합병, 반복문 롤링, 순차적 합병)는 이전 절에서의 과정과 동일하다. 변환된 그래프 G_S 로부터 LOR 알고리즘을 사용하여 GOR 문제를 풀게 된다.

6. 실험

제시된 연산 재배치 알고리즘이 응용 프로그램에서 얼마나 효과적인지를 살펴보기 위해서, 우리는 Texas Instruments사의 VLIW 디지털 신호 처리기인 TMS320C6201[19]을 사용하여 실험을 하였다. TMS320C6201 프로세서는 고정 소수점 DSP로 하나의 256비트 명령어안에 8개의 32비트 연산을 사용할 수 있다. TMS320C6201은 압축 인코딩 방식을 사용하며 $b_{cache} = 256$ 이다. 외부 메모리 연결 장치는 32비트의 명령어 버스를 통해 외부 메모리와 연결된다(즉, $b_{mem} = 32$).

본 논문에서 제안된 연산 재배치 기법은 별도의 후처리 도구로 구현되어 TI의 TMS320C6x 최적화 C 컴파일러에 의해 생성된 실행 파일을 입력으로 받고 같은 기능의 재배열된 저전력 버전을 출력으로 생성한다. 벤치마크 프로그램으로는 다양한 DSP 프로그램들이 사용되었다.

벤치마크 프로그램들에 대한 명령어 추출시의 비트 전환 회수는 스위칭 활동 계수기(counter)에 의해서 측정되었다. 적당한 입력 데이터를 가진 실행 파일이 주어졌을 때, 스위칭 활동 계수기 프로그램은 명령어의 주소 트레이스를 이용하여 프로그램의 수행동안 내부와 외부 버스에서의 비트 전환을 계산한다. 벤치마크 프로그램에 대한 명령어 주소 트레이스는 소스 프로그램에 대한 수작업의 분석에 의해 만들어졌다.

표 5 실험 결과

Benchmark Program	Bit transitions/IF			Reduction	
	default	LOR	GOR-II	LOR	GOR-II
vector multiply	68.6	46.0	43.7	33.0%	36.3%
FIR8	86.8	59.3	56.7	31.6%	34.6%
FIRcx	79.5	60.6	60.5	23.9%	24.0%
IIR	71.7	52.1	51.7	27.4%	28.0%
lattice analysis	88.4	63.4	58.2	28.3%	34.2%
W_vec	89.5	62.9	57.1	30.0%	36.3%
dotp_sqr	79.2	44.5	44.3	43.9%	44.1%
minerror	50.6	33.2	31.3	34.3%	38.1%
biquad	78.1	54.6	52.3	30.0%	33.0%
Average	76.9	53.0	50.6	31.4%	34.3%

표 5는 몇몇 DSP 벤치마크 프로그램에 대한 실험 결과를 보여 주고 있다. 각 프로그램에 대해, 명령어 추출 당 평균적인 비트 전환 회수(BT/IF)가 계산되어 있다. 여기서 α 값으로는 100을 사용하였으며[20], TI 컴파일러가 만든 프로그램(표 5의 default 열)과 제시된 LOR 기법에 의해 재배치된 프로그램(표 5의 LOR 열), 그리고 GOR 휴리스틱 기법에 의해 재배치된 프로그램(표 5의 GOR-H 열)들의 BT/IF를 비교하였다. GOR 휴리스틱 기법에서는 N_{count} 값으로 100을 사용하였다.

표 5에서 보듯이, 연산 재배치 기법은 TI 컴파일러가 생성한 프로그램과 비교하여 명령어 추출시의 비트 전환의 수를 평균 34.3% 줄였다. GOR 휴리스틱 기법은 LOR 기법보다 2.9%정도 스위칭 활동을 더 줄일 수 있었다. 그러나 대부분의 벤치마크 프로그램에서 LOR 기법은 GOR 휴리스틱 기법과 거의 동일한 효과를 나타내고 있다.

7. 결론

본 논문에서 우리는 VLIW 기계에서 명령어 추출을 위한 연산 재배치 기법을 제시하였다. 컴파일된 프로그램에 후처리 도구로 사용할 수 있도록 제시된 기법은 VLIW 명령어내의 연산의 순서를 재배치하여 명령어 추출동안 비트 전환의 회수가 최소가 되도록 한다. 실험 결과는 제시된 재배치 기법이 VLIW 기계에서 명령어 추출 동안의 스위칭 활동을 많이 줄일 수 있음을 보여주었다.

본 논문에서 제안된 연산 재배치 기법은 몇 가지로 연구방향을 확장할 수 있다. 첫째, 본 논문에서는 VLIW 프로세서에 초점을 두었지만 비슷한 연산 재배치 기법이 슈퍼스칼라(superscalar) 프로세서에서 저전

력 명령어 추출을 위해서도 효과적인 것으로 기대된다. 예를 들어, 4-way 슈퍼스칼라 프로세서를 사용하여 행한 간단한 실험의 결과는 15%정도의 스위칭 활동 감소가 있음을 알 수 있었다. 현재 슈퍼스칼라 구조의 CPU를 위한 연산 재배치 알고리즘이 개발되고 있다.

둘째, 본 논문에서는 이미 컴파일된 프로그램에 대해 연산의 순서를 수정하는 문제를 다루었으나 컴파일 과정 중 내려진 코드 최적화에 관한 사항들은 연산 재배치에 많은 영향을 미칠 수 있다. 예를 들면, 명령어들이 어떻게 스케줄되느냐에 따라 명령어 추출단계에서의 비트 변화의 횟수는 크게 달라진다. 연산 재배치 문제를 후처리 과정이 아닌 컴파일의 한 과정으로 다루며 일반적으로 행해지는 컴파일러 최적화 과정과의 상호 영향을 연구하는 것은 저전력 코드를 생성하는 데 중요한 기여를 할 것으로 기대되고 더 많은 연구가 필요한 부분이라 생각된다.

참고 문헌

- [1] A. Klaiber. *The technology behind the Crusoe processor*. Transmeta Corporation White Paper, 2000.
- [2] Fujitsu Microelectronics, Inc. *Fujitsu's new high-performance VLIW processor cores*. <http://www.fujitsumicro.com/>.
- [3] P. Faraboschi, G. Desoli, and J. A. Fisher. The latest word in digital and media processing. *IEEE Signal Processing Magazine*, 15(2):59-85, 1998.
- [4] R. Henning and C. Chakrabarti. High-level design synthesis of a low power, VLIW processor for the IS-54 VSELP speech encoder. In *Proc. of Int. Conf. on Computer Design (ICCD'97)*, pp. 571-576, 1997.
- [5] Texas Instruments. *TMS320C6000 Power Consump-*

- tion Summary, 1999.
- [6] J.-M. Puiatti, J. Llosa, C. Piguet, and E. Sanchez. Low-power VLIW processors: A high-level evaluation. In *Proc. of Int. Workshop-Power and Timing Modeling, Optimization and Simulation (PATMOS '98)*, pp. 399-408, 1998.
- [7] A. Chandrakasan, T. Shung, and R. W. Broderson. Low power CMOS digital design. *IEEE Journal of Solid State Circuits*, 27(4):473-484, 1992.
- [8] S. Devadas and S. Malik. A survey of optimization techniques targeting low power VLSI circuits. In *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97)*, pp. 239-242, 1997.
- [9] M. R. Stan and W. P. Burleson. Bus-invert coding for low power I/O. *IEEE Trans. on VLSI Systems*, 3(1):49-58, 1995.
- [10] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97)*, pp. 72-75, 1997.
- [11] C. L. Su, C. Y. Tsui, and A. Despaigne. Low power architectural design and compilation techniques for high-performance processor. In *Proc. of COMPCON94*, pp. 489-498, 1994.
- [12] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proc. of Int. Symp. on Low-Power Electronics*, 1994.
- [13] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. *IEEE Trans. VLSI Systems*, 2(4):437-445, 1994.
- [14] M. T. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *IEEE Trans. VLSI Systems*, 5(1):123-135, 1997.
- [15] H. Tomiyama, T. Ishihara, A. Inoue, and H. Yasuura. Instruction scheduling for power reduction in processor-based system design. In *Proc. of the 1998 Design Automation and Test in Europe (DATE '98)*, pp. 855-860, 1998.
- [16] M. C. Toburen, T. M. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. In *Proc. of Power Driven Microarchitecture Workshop*, 1998.
- [17] C. Gebotys, R. Gebotys, and S. Wiratunga. Power minimization derived from architectural-usage of VLIW processors. In *Proc. of Conf. on Design Automation (DAC'2000)*, pp. 308-311, 2000.
- [18] T. Conte, S. Banerjia, S. Larin, K. N. Menezes, and S. W. Sathaye. Instruction fetch mechanisms for VLIW architectures with compressed encodings.

In *Proc. of the 29th IEEE/ACM Int. Symp. on Microarchitecture*, pp. 201-211, 1996.

- [19] Texas Instruments. *TMS320C62xx CPU and Instruction Set*, 1997.
- [20] E. Musoll, T. Lang, and L. Cortadella. Exploiting the locality of memory references to reduce the address bus energy. In *Proc. of Int. Symp. on Low Power Electronics and Design (ISLPED'97)*, pp. 202-207, 1997.



신 동 군

1994년 서울대학교 계산통계학과 학사.
2000년 서울대학교 전산학과 석사. 현재 서울대학교 전기 컴퓨터 공학부 박사과정. 관심분야는 컴퓨터구조, 내장형 시스템, 저전력 시스템, 실시간시스템



김 지 흥

1986년 서울대학교 계산통계학과 학사.
1988년 University of Washington 컴퓨터과학과 석사. 1995년 University of Washington 컴퓨터과학 및 공학과 박사
1995년 ~ 1997년 미국 Texas Instruments사 선임연구원. 1997년 ~ 현재 서울대학교 전기 컴퓨터공학부 조교수. 관심분야는 컴퓨터구조, 내장형 시스템, 저전력 시스템, 멀티미디어 시스템