17-3-R04

다중 메모리 구조를 위한 효율적인 자료 할당 기법

(Efficient Data Assignment for Multi-Bank Memory Architectures)

조 정 훈*, 백 윤 흥**, 최 준 식**
*하이닉스 반도체, **서울대학교 전기·컴퓨터학부
*jhcho@soar.kaist.ac.kr, **{ypaek, jschoi}@ee.snu.ac.kr

요 약

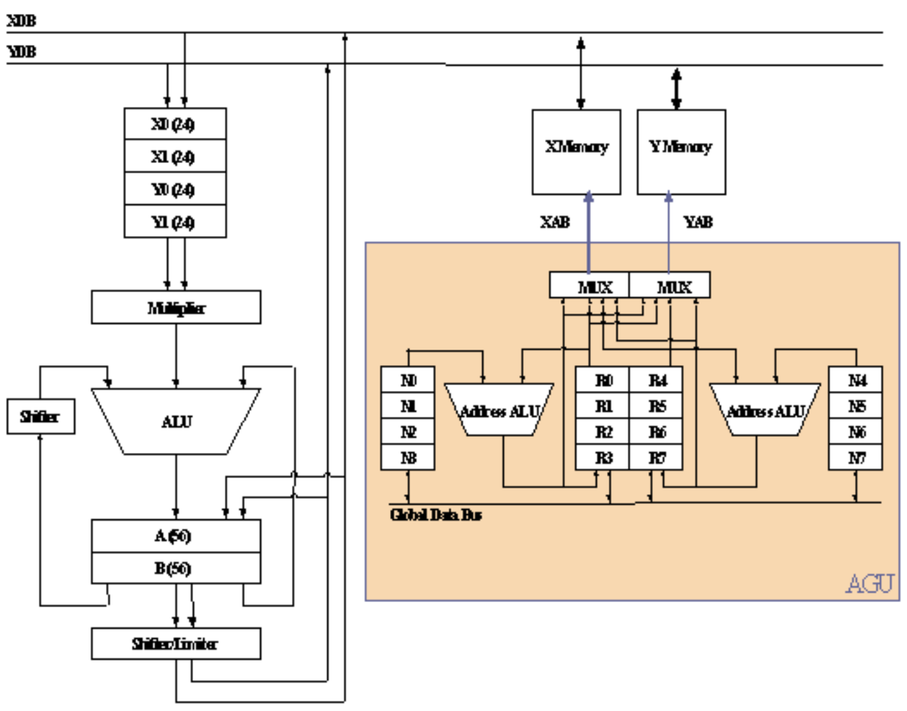
현존하는 대부분의 DSP들은 다중 메모리 뱅크를 지원하는 하버드 구조를 가진다. 따라서 하나의 명령어를 사용하여 메모리에서 여러 개의 데이터 워드를 동시에 읽어 들이는 것이 가능하다. 뿐만 아니라, 이러한 DSP들은 여러 개의 레지스터 파일이 각 기능(functional) 유닛에 분산되어 있는 비정규적인 구조로 되어 있다. 이러한 다중 메모리 뱅크 구조를 효율적으로 사용하기 위해 각 메모리 뱅크에 데이터를 효율적으로 할당하려는 노력이 여럿 선행되었지만, 대부분이 정규적인 구조를 가정한 연구였으며, 그 결과, 비정규적인 구조를 가지는 DSP에 대해서는 최적의 코드를 생성하지 못하는 경우가 종종 발생하였다. 본 논문에서는 다중 메모리 뱅크 구조에서 각 메모리 뱅크에 데이터를 효율적으로 할당하는 새로운 알고리듬을 설명한다. 기존의 연구는 메모리 뱅크 할당을 다른 코드 생성 알고리듬과 통합하여(tightly coupled) 수행하였으나, 우리의 메모리 뱅크 할당 알고리듬은 독립된 컴파일 단계에서 동작하도록 설계되었다(Decoupled). 실험 결과, 우리의 독립된(decoupled) 메모리 할당 알고리듬은 기존의 알고리듬과 비슷한 품질의 코드를 생성하면서도 컴파일 시간을 극적으로 향상시킨 것을 알 수 있었다.

1. 서론

내장형 시스템 개발자들이 응용 프로그램에서 요구하는 비용과 성능의 목표를 만족하기 위해 노력함에 따라 프로세서의 복잡성은 점차 증가하고 있다. 이러한 이유로 특별한 응용분야에 대해서만 최적화된 프로세서가 나타나고 있다. 일반적으로 특정 응용분야에 최적화된 하드웨어 구성을 찾기 위해서는 디자인 영역 조사(design space exploration)[8]의 과정이 필요로 되어 진다. 그리고 이러한 조사 과정을 통해 생성된 최종 구성들은 각각 고려한 응용 분야에 가장 적합한 데이터 패스와 명령어 집합을 가진다. 이러한 이들의 공통적인 특징 때문에 위와 같은 과정을 통해 생성된 프로세서를 일괄적으로 ASIP(application specific instruction-set processor)라고 부른다.

ASIP은 전형적으로 이종(heterogeneous) 레지스터와 다중 메모리 뱅크를 포함하는 비 규칙적인 데이터 패스를 가진다. 그러한 구조의 예로서 (그림 1)을 보자. (그림 1)은 디지털 신호 처리(DSP) 프로그램을 위해 특별히 구성되어진 Motorola DSP56000의 구조이다. 이 구조에서 주목할 만한 특징 중 하나는 한곳에 집중되어 있는 다수의 공용 레지스터가 존재하지 않는 다는 것이다. 대신에 적은수의 레지스터가 각 하드웨어 기능수행 구성단위의 다른 집합에

분산되고 해당 구성단위에서만 사용되어 진다. 또한 이 구조는 하나의 메모리 뱅크를 가지고 있는 전통적인 von Neumann 구조에서 벗어나 다중 메모리 뱅크를 가진다는 특징을 가진다. 아래그림은 두개의 데이터 메모리 뱅크와 하나의 프로그램 메모리 뱅크가 독립적인 버스를 통해 연결되어 있는 것을 보여준다. 이러한 종류의 메모리 구조는 많은 내장형 프로세서에서 지원되어지는데, 예를 들어 Analog Device ADSP2100, DSP Group PineDSPCore, Motorola DSP56000, 그리고 NEC uPD77016이 있다. 이 구조의 명확한 이점 중 하나는 한 명령어 사이클에 두개의 데이터 워드를 접근할 수 있다는 것이다.



(그림 1) 듀얼 데이터 메모리 뱅크 X, Y를 가지는 모토롤라 DSP56000 데이터 패스

ASIP의 주요 특징 중 하나인 다중 메모리 뱅크 구조는 내장형 프로그램에서 발견할 수 있는 많은 연산에 대하여 효과적으로 사용 될 수 있다. 예를 들어 n개의 곱셈(z(i) = x(i) * y(i) i = 1,2,... N) 연산을 생각해 보자. 이 경우 만약 프로세서가 x(i)와 y(i) 두개의 피연산자를 동시에 가져올 수 있다면 실행속도에서 더 좋은 성능을 얻을 수 있을 것이다. 그러나 이러한 이상적인 연산 속도를 얻기 위해서는 프로세서가 두개의 데이터 메모리 뱅크를 가지고 있어야 한다. 또한 필요한 두개의 값이 서로 다른 데이터 메모리 뱅크에 할당되어 있어야 한다. 예를 들어 n개의 곱셈을 구현하는 다음의 DSP56000 어셈블리 코드에서 배열 x와 y가 각각 다른 메모리 뱅크 X, Y에 할당되어 진 경우에 좋은 결과를 얻을 수 있다. 그러나 불행하게도 몇 종류의 업체에서 제공하는 컴파일러에 대한 테스트로부터 우리는 그들이 듀얼 데이터 메모리라는 하드웨어 특성을 효율적으로 지원하지 않음을 알 수 있었다.

```
move    x:(r0)+,x0
y:(r4)+,y0
mpyr    x0,y0,a     x:(r0)+,x0
y:(r4)+,y0
do      #N-1,end
mpyr    x0,y0,a     a,x:(r1)+
y:(r4)+,y0
move                x:(r0)+,x0
end
move                a,x:(r1)+
```

이것은 만약 프로그래머가 듀얼 데이터 메모리 뱅크를 효율적으로 사용하고 싶다면 직접 어셈블리 코드를 작성해야만 한다는 것을 의미한다. 어셈블리 코딩은 복잡하고 많은 시간을 소모하는 작업이다. 더욱이 듀얼 메모리 뱅크를 지원하기 위해서는 각 변수들 간의 의존성을 포함한 많은 부분이 고려되어야 하므로 어셈블리 코딩에 대한 복잡성은 더욱 크게 증가되어 진다.

위에서 살펴 본 것처럼 데이터 할당은 비 규칙적인 구조를 가지는 ASIP을 위한 코드 생성에서 매우 중요한 기술이다. 본 논문에서는 이 문제의 핵심기술인 레지스터 할당과 메모리 뱅크 할당에 대한 우리의 접근 방식을 기술한다. 우리의 레지스터 할당 기법은 ASIP의 주요 특징 중 하나인 이종 레지스터를 효율적으로 다루기 위하여 다음과 같이 두 단계로 분리되어 진다.

1. 물리적인 레지스터들을 레지스터 클래스들로 분류 한다.(레지스터 클래스는 같은 명령어에 사용될 수 있는 레지스터들의 집합이

다.) 그리고 우리의 레지스터 분류 알고리즘을 이용하여 각각의 임시 변수를 레지스터 클래스 중 하나에 할당한다.
2. 전통적인 그래프 컬러링 알고리즘을 변형하여 임시 레지스터에 물리적 레지스터를 할당한다. 이때 할당되는 물리적 레지스터는 1번 과정에서 할당된 레지스터 클래스에 속해야 한다.

변수들을 다중 메모리 뱅크에 효율적으로 할당하기 위하여 메모리 뱅크 할당 역시 아래와 같은 두 단계로 나누어서 수행하였다.

1. MST(Maximum spanning tree) 알고리즘을 이용하여 각 변수에 메모리 뱅크를 할당한다.
2. 레지스터 할당에 사용되는 그래프 컬러링 알고리즘을 이용하여 1의 결과를 향상 시킨다.

우리의 알고리즘은 이종 레지스터와 다중 메모리 뱅크 할당을 한 단계에서 수행[13,14]하는 이전의 연구와 달리 위에서 보인바와 같이 별개의 작업으로 나누어 수행한다는 특징을 가진다. 본 논문의 뒷부분에서 언급되어지듯이 우리는 상당히 만족스러운 결과를 얻을 수 있었다. 무엇보다도 코드 생성 시간이 10^4의 비율로 줄어든 것을 확인 할 수 있었다. 이러한 결과는 코드 생성 단계가 레지스터 할당과 메모리 뱅크 할당 단계로 분리되어 전체 적인 코드 생성의 복잡성이 줄어 들 수 있다는 것으로부터 예측 할 수 있다. 특히 이렇게 향상된 코드 생성 시간이 기존 연구와 거의 동일한 코드 성능과 함께 나타났다는 점에서 더욱 만족스럽다 할 수 있다.

우리의 알고리즘을 기술하기 위해 2장에서는 먼저 연구에서 사용한 ASIP 구조를 설명한다. 3장에서는 알고리즘을 제시하고 4장에서는 초기의 연구[5] 이후로 가장 최근에 얻어진 실험 결과를 제시한다. 마지막으로 5장에서 결론을 맺는다.

2. 타겟 머신 모델

본 장에서는 이종 레지스터와 다중 메모리 뱅크라는 두 특징과 함께 ASIP의 비 규칙적인 구조를 기술한다.

2.1 이종 레지스터

이종 레지스터를 정형화해서 정의하기 위해 다음의 정의를 먼저 제시한다.

정의 1. 주어진 타겟 머신 M에 대하여 M에서 정의된 모든 명령어 집합을 $I = \{i_1, i_2, \ldots i_n\}$라고 하자. 또한 $i_j \in I$인 모든 명령어에 대하여 모든 오퍼랜드의 집합을 $Op(i_j) = \{O_{j1}, O_{j2}, \ldots, O_{jk}\}$라고 정의하자. C_{jl}을 피연산자 O_{jl} $(1 \le l \le k)$의 한 위치에서 나타날 수 있는 모든 레지스터들의 집합이라고 가정할 때, 우리는 C_{jl}을 명령어 i_j의 레지스터 클래스라고 정의한다.

정의 2. 정의 1로부터 명령어 i_j를 위한 유일한 레지스터 클래스 S_j는 다음과 같이 정의된다.

$$S_j = \bigcup_{l=1}^{k} C_{jl}$$

이로부터 우리는 S를 다음과 같이 정의 할 수 있다.

$$S = \bigcup_{j=1}^{n} S_j$$

여기서 S는 머신 M에 대한 레지스터 클래스의 전체 집합이다.

위의 정의와 함께 동종 레지스터와 이종 레지스터의 차이를 보이기 위해 DSP56000과 SPARC의 두개의 프로세스를 가지고 설명하겠다. 먼저 동종 레지스터를 가지는 SPARC 프로

세서를 살펴보자. 전형적인 SPARC의 명령어들은 세 개의 피연산자를 가지고 있다.

op_code reg_i, reg_j, reg_k

위의 명령어에서 나타나는 첫 번째 레지스터 reg_i는 동종 레지스터의 특징에 따라 레지스터 파일에 있는 모든 32개의 레지스터(r0,r1,...,r31)를 사용할 수 있다. 이 경우에 이 모든 레지스터들의 집합은 op_code에 대하여 하나의 레지스터 클래스를 형성한다. 다른 피연산자인 reg_j와 reg_k도 reg_i와 같은 32개의 레지스터를 사용할 수 있기 때문에 그 명령어에 대하여 reg_i와 같은 레지스터 클래스를 형성한다. 따라서 명령어 op_code에는 단지 하나의 레지스터 클래스가 정의된다. 이제 이종 레지스터를 가지는 DSP56000을 아래의 명령어와 함께 살펴보자.

mpya reg_i, reg_j, reg_k

위의 명령어는 처음 두개의 피연산자(reg_i, reg_j)를 곱해서 세 번째 레지스터(reg_k)에 값을 저장하는 명령어이다. 이 명령어에 대해서 DSP56000은 reg_i와 reg_j로 올수 있는 레지스터를 X0, X1, Y0, Y1의 4개로 제한하고 reg_k의 경우는 누산기 A와 B로 제한한다. 따라서 mpya 명령어에 대하여 우리는 reg_i와 reg_j에 대하여 {X0,X1,Y0,Y1}의 레지스터 클래스를 정의하고 reg_k에 대하여 {A,B} 클래스를 정의한다. 그러므로 위의 예제에서 op_code와 mpya에 대한 S_j는 각각 {{r0,r1,...,r31}}과 {{X0,X1,Y0, Y1}, {A,B}}이 된다. 우리는 SPARC과 같이 n개의 범용 레지스터를 가지는 전형적인 프로세서를 동종 레지스터 구조라 한다. 이는 그 프로세서에 대한 S가 {{r0,r1,...,rn}}이므로 정의 1과 2에 의해서 n개의 레지스터가 어떤 명령어에 대하여도 균일(homogeneous)하게 사용 될 수 있기 때문이다. 그러나 DSP56000의 레지스터들은 명령어에 따라 다르게 고정되어 있기 때문에 명령어의 부분집합에 대하여만 균일하게 사용된다. 예를 들어 심지어 DSP56000의 mpya와 같은 하나의 명령어조차도 두개의 다른 동종의 레지스터 집합 (XYN과 AB)을 가지고 있는 것을 볼 수 있다. 표 1에서는 DSP 56000에서 정의된 모든 레지스터 클래스를 보이고 있다. 일반적으로 이렇게 복잡한 레지스터 클래스를 가지고 있는 구조를 이종 레지스터 구조라고 한다.

2.2 다중 데이터 메모리 뱅크

다중 메모리 뱅크 구조를 가지는 ASIP의 예제로서 우리는 (그림 1)에서 보인 DSP56000을 사용한다. 이 프로세서의 ALU 연산은 데이터 연산과 주소(address) 연산으로 나누어져 있다. 데이터 연산은 4개의 24 비트 입력 레지스터 (X0, X1, Y0, 그리고 Y1)와 두개의 56 비트 누산기 (A와 B)로 구성되어진 데이터 레지스터를 가지며 데이터 ALU에서 수행된다. 주소 연산은 데이터 연산에서 필요로 하는 메모리 주소를 계산하는 것으로 AGU(address generation unit)에서 수행된다.

AGU는 데이터 ALU와 독립적으로 동작하기 때문에 주소 계산과 데이터 연산은 동시에 수행 될 수 있다.

(표 1) 모토롤라 DSP56000의 레지스터 클래스

ID	레지스터 클래스	물리 레지스터
1	XYN	X0,X1,Y0,Y1
2	XY	X,Y(long word)
3	YR	R4-R7
4	AB	A, B
5	YN	N4-N7
6	XR	R0-R3
7	XN	N0-N3
8	X	X0,X1
9	Y	Y0,Y1

(그림 1)에서 보는 바와 같이 AGU는 두개의 동일한 부분으로 구성되어 있으며 각각은 address ALU와 16 비트 레지스터 파일과 멀티플렉서로 구성되어 있다. 레지스터 파일은 4개의 주소 레지스터로 이루어져 있는데 각각 (R0~R3)과

(R4~R7)를 사용한다. 멀티플렉서는 XAB와 YAB에 실리는 값을 선택하는 역할을 한다. 단 이 값으로는 address ALU의 출력과 주소 레지스터의 값만이 올 수 있다. 그리고 각 사이클에서 address ALU에 의해 생성된 주소는 X와 Y의 두개의 메모리 뱅크에 동시 접근할 수 있도록 병렬적으로 사용될 수 있다.

DSP56000은 X, Y, L, XY라고 하는 4개의 메모리 참조 모드를 가진다. X나 Y 모드에서 피연산자는 해당되는 메모리 뱅크 즉 X 또는 Y 하나로부터 가져온 하나의 워드이다. L이라는 말에서 유추 할 수 있는 것처럼 L 모드에서 피연산자는 롱 워드이다. 이 롱 워드는 X와 Y 메모리로부터 가져온 두개의 워드로 구성되어 진다. XY 모드역시 X와 Y메모리 둘 다로부터 하나의 워드를 가져온다. 단 L 모드가 같은 어드레스를 사용하는 반면 XY 모드는 두개의 독립적인 어드레스를 사용한다. XY 메모리 레퍼런스와 같이 같은 사이클에 발생하는 두개의 독립적인 데이터 이동을 병행 이동(parallel move)이라고 한다. 그림 1에서 우리는 DSP56000의 데이터 패스와 두 데이터 메모리 뱅크 X와 Y를 연결하는 두개의 데이터 버스 XDB와 YDB를 볼 수 있는데 이 버스들을 통해서 메모리와 데이터 레지스터 사이의 병행 이동이 만들어진다.

우리는 위에서 DSP56000의 구조적인 특성을 살펴보았다. 위와 같은 DSP56000의 구조적 특성은 다중 메모리 뱅크를 가지고 있는 다른 ASIP에서와 같이 한 사이클에 하나의 ALU 연산과 두개의 move 연산을 동시에 수행할 수 있게 한다. 단 이와 같은 구조를 최대화하기 위해서는 몇 가지 조건을 만족해야 하는데 이는 다음과 같다.

1. 두 워드는 다른 메모리 뱅크에 존재해야 한다.
2. 주소는 어드레스 레지스터를 사용하는 메모리 인디렉트 어드레싱 모드여야 한다.
3. 병행 이동에서 사용되는 각 어드레스 레지스터는 AGU의 두 레지스터 파일에서 서로 다른 집합에 있어야만 한다.

본 연구의 목표는 위와 같은 병행 이동 조건을 만족시킴으로서 가능한 한 많은 병행 이동이 발생하도록 듀얼 메모리 뱅크를 위한 코드를 생성하는 것이다.

3. 레지스터 할당과 메모리 뱅크 할당

본 장에서는 1장에서 간략히 기술된 레지스터 할당과 메모리 뱅크 할당을 위한 코드 생성 과정을 자세히 기술한다. 이 과정에서 우리의 코드 생성기가 최종 코드를 어떻게 생성하는지를 단계적으로 설명하기 위해 (그림 2)에서 보이는 DSP56000 어셈블리 코드를 예로서 사용한다. 이 코드는 명령어 선택 단계를 거친 후에 바로 얻을 수 있는 것으로서 순차적이고 최적화되지 않은 초기 코드이다. 이러한 초기 코드는 컴파일러의 다음 단계로 전달되어 최적화되는데 본 장에서는 DSP56000의 듀얼 메모리 구조를 위해 최적화되는 과정을 기술한다.

```
MOVE    a, r0
MOVE    b, r1
MOVE    c, r2
MAC     r0, r1, r2
MOVE    low(r2), low(v)
MOVE    high(r2), high(v)
MOVE    d, r3
MOVE    e, r4
MOVE    f, r5
MAC     r3, r4, r5
MOVE    low(r5), low(w)
MOVE    high(r5), high(w)
```

(그림 2) 명령어 선택 후에 생성된 최적화되지 않은 DSP56000 어셈블리 코드 예제

3.1 레지스터 클래스 할당

우리의 컴파일러에서 명령어 선택 단계는 레지스터 할당을 포함한 모든 다른 단계와 분리되어 있다. 사실 범용 목적의 프로세서 (GPP)를 위한 코드 생성을 목적으로 하는 gcc, lcc, 그리고 Zephyr와 같은 많은 전통적 컴파일러 역시 이 두 단계가 분리되어져 있다. 솔직히GPP는 하나 또는 몇 개의 레지스터 클래스를 가지는 동종 레지스터구조이기 때문에 GPP를 위한 코드 생성에 있어서 명령어 선택과 레지스터

할당을 분리시키는 것은 상당히 직관적이다. 단지 명령어 선택 단계에서는 레지스터를 필요로 하는 명령어에 임시 레지스터를 할당 하면 된다. 그리고 뒤의 레지스터 할당 단계에서 같은 레지스터 클래스에서 사용 가능한 레지스터를 배정해 주면된다. 그러나 ASIP에서는 각 개별 명령어의 레지스터 클래스가 다르고 또한 레지스터는 많은 다른 레지스터 클래스들에 속해있기 때문에 GPP와 같이 단순하게 해결되지 않는다(표 1참조). 즉, 레지스터와 명령어 사이에 밀접한 관련이 있기 때문에 우리가 명령어를 선택하려고 할 때 어떤 레지스터 클래스의 레지스터가 사용될 건지도 같이 결정해야만 하는 것이다. 이와 같은 이유로 ASIP을 위한 코드를 생성하기 원하는 컴파일러는 일반적으로 phase-coupling[9]이라고 하는 밀접한 단계를 결합하는 기법을 사용한다. 그러나 phase-coupling은 코드 생성 시 너무 많은 제약을 야기한다. 때문에 4.1 절에서 우리의 결과와 비교되어질 기존의 연구 결과와 같이 엄청난 컴파일 시간을 필요로 할지 모른다.

우리는 phase-coupling의 제약에 기인하여 두 단계를 분리하여 다루는 방법을 채택하였다. 대신 이종 레지스터 구조를 여전히 다루기 위하여 이 두 단계 사이에 레지스터 클래스 할당(register class allocation)이라 불리는 추가적인 단계를 삽입함으로서 두 개의 분리된 단계를 연결해주었다. 이러한 체계에서 레지스터는 레지스터 클래스와 그 클래스내부에서의 레지스터 번호라고 하는 두 가지 개념으로 표현된다. 그리고 이 두 값은 레지스터 클래스 할당과 배정의 두 단계를 통하여 결정되어진다. 레지스터 클래스 할당 단계에서는 물리적인 레지스터 대신 레지스터 클래스를 임시레지스터에 할당한다. 물리적인 레지스터로의 할당은 레지스터 배정(register assignment) 단계에서 수행되어 진다. 이 단계에서 배정되는 레지스터는 레지스터 클래스 할당 단계에서 할당되어진 클래스들 사이에서 이용 가능한 레지스터이다. 우리가 예제로 채택한 그림 2의 코드로부터 할당되어진 레지스터 클래스들은 아래와 같다. 여기서 각 항목은 코드에서의 각 임시 레지스터 r_i에 대한 레지스터 클래스를 나타낸다. 본 논문의 초점은 레지스터 클래스 할당이 아니기 때문에 전체 알고리즘에 대해서 언급하지는 않겠다. 더 자세한 사항은 [6]을 참고하면 된다.

```
r0 : XYN          r1 : XYN
r2 : AB
r3 : XYN          r4 : XYN
r5 : AB
```

레지스터 클래스 할당과 레지스터 배정 사이의 코드 압축(code compaction) 단계는 코드 크기를 줄이는 것뿐만 아니라 최대 하나의 ALU 연산과 두개의 move를 동시에 수행할 수 있는 병렬 연산을 찾는다. (그림 3)은 (그림 2)의 코드가 압축된 후의 결과를 보이고 있다. 여기에서 우리는 하나의 MAC(multiply-and-accumulate) 명령어와 두개의 move 명령어가 한 명령어 워드로 결합되어 있는 것과, 두개의 move 명령어가 하나의 명령어 워드로 결합되어 있는 것을 볼 수 있다. 이러한 코드 압축을 행하기 위해 우리는 전통적인 리스트 스케쥴링 알고리즘을 사용하였다.

```
MOVE              a, r0   b, r1
MOVE              c, r2   d, r3
MAC  r0, r1, r2   e, r4   f, r5
MAC  r3, r4, r5   low(r2), low(v)
MOVE              high(r2), high(v)   low(r5), low(w)
MOVE              high(r5), high(w)
```

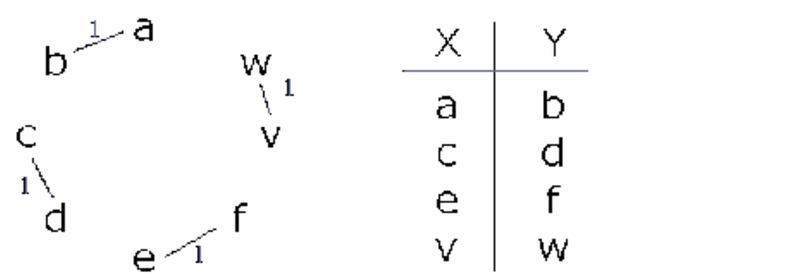
(그림 3) 그림 2로부터 압축 후의 코드

3.2 메모리 뱅크 배정

레지스터 클래스 할당과 코드 압축까지의 단계를 거쳐 생성된 코드에서 각 변수들은 하나의 메모리 뱅크로만 배정되어 있다. 본 절에서 우리는 두개의 잘 알려진 알고리즘을 사용하여 변수들을 다중 메모리 뱅크로 배정하는 기법을 제시한다.

3.2.1 MST 알고리즘 사용

기본적인 메모리 뱅크 배정 단계에서 우리는 MST 알고리즘을 사용한다. 이 단계에서 첫 번째 할 일은 SRG(simultaneous reference graph)라 불리는 MST를 수행할 그래프를 생성하는 것이다. 이름에서 추론할 수 있듯이 이 그래프는 어떤 변수들이 동시에 메모리 참조를 시도하는지를 타나내는 그래프이다. 따라서 이 그래프의 노드들은 코드에서 참조되는 변수를 나타내며 에지들은 같은 명령어에서 참조되어 지는 변수들을 연결한다. 즉, SRG에서 에지 $e=(v_j, v_k)$는 노드 v_j와 v_k에 해당하는 변수들이 압축된 코드상의 같은 명령어 워드에서 메모리를 참조한다는 것을 의미한다. (그림 4(a))는 (그림 3)의 코드를 위한 SRG를 보인다. 아래 그림에서 에지에서의 가중치는 변수들이 같은 워드에서 참조된 회수를 나타낸다.

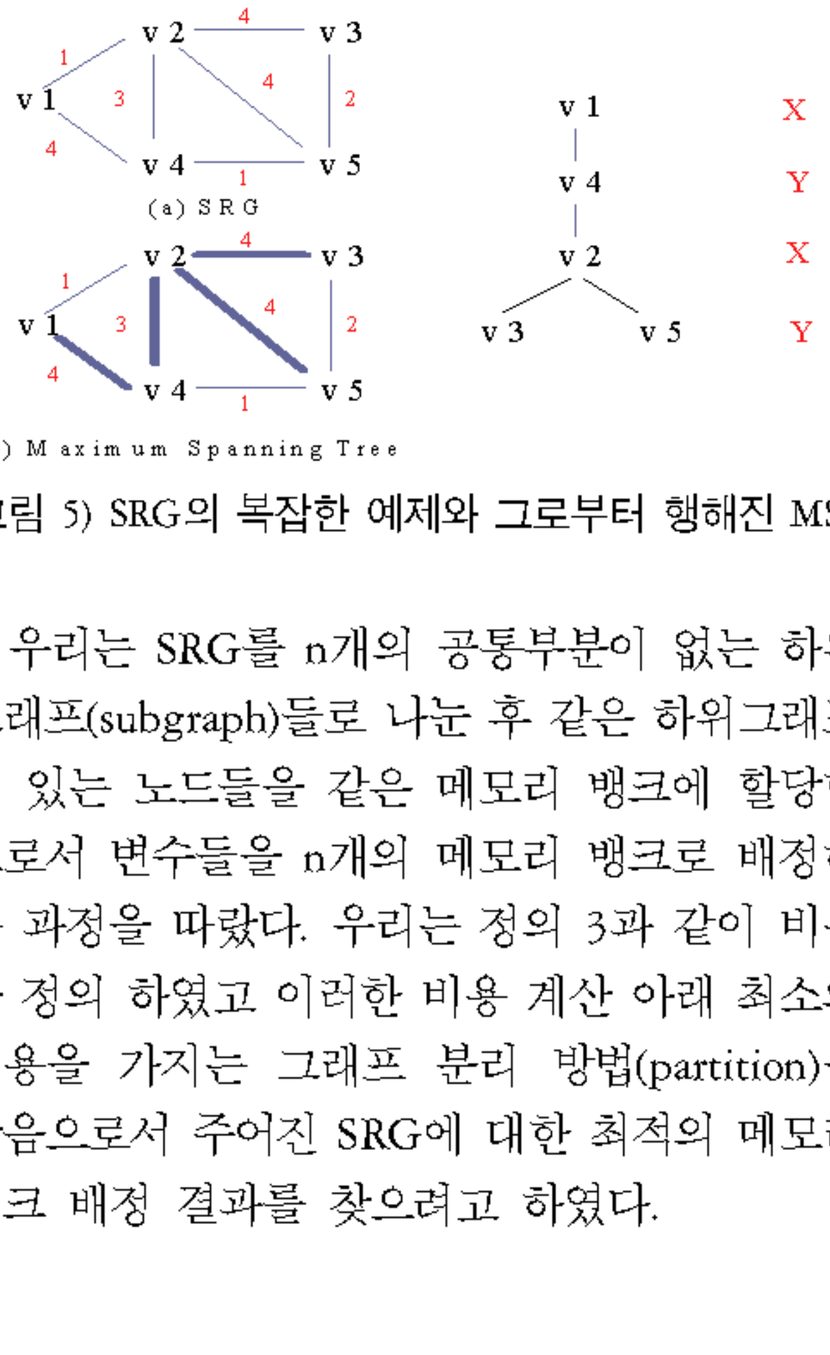


(그림 4) 그림 3의 코드를 위해 생성된 SRG와 이로부터 결정된 메모리 뱅크 배정 후의 결과

병행 이동 조건에 따라서 한 명령어 워드에서 참조되는 두 변수들을 한 명령어 사이클 내에 읽어오기 위해서는 그 변수들이 다른 메모리 뱅크에 배정되어야 한다. 그렇지 않은 경우 그 두 변수들은 한 명령어 워드에 올 수 없기 때문에 분리되어야만 한다. 따라서 추가적인 사이클이 필요하게 된다. 그러므로 메모리 효율을 최대화하기위하여 우리는 같은 명령어 워드에서 참조되는 변수들을 가능한 한 다른 메모리 뱅크에 배정을 하는 방식을 택하였다. 만약 두 쌍의 변수들 사이에 충돌이 발생한다면 같은 워드에서 더 빈번하게 나타나는 한 쌍의 변수들이 다른 변수들 보다 더 높은 우선순위를 갖도록 하였다. 이에 해당하는 빈도는 SRG에 있는 에지의 가중치를 통해 알 수 있다.

(그림 4(b))는 a, c, e, v의 4개의 변수가 X 메모리에 배정되고 나머지 변수 b, d, f, w가 Y 메모리에 배정된 것을 보여준다. 에지를 통해 연결된 모든 쌍의 변수들이 다른 메모리 뱅크에 배정이 되었으므로 이 예는 최적화된 것이다. 따라서 (그림 4(c))의 결과 코드에서 변수를 읽어오기 위한 여분의 사이클을 피할 수 있다. 비록 변수 v와 w가 두 워드의 길이를 가지기 때문에 데이터를 이동하기 위해서 두 사이클이 필요 하지만, 두 변수들의 반이 같은 사이클에 동시에 옮겨지기 때문에 최적화되었다고 말할 수 있다.

그러나 현실적으로 만나게 되는 메모리 뱅크 배정 문제는 (그림 4)에서와 같이 항상 단순하지 않다. 이 문제의 좀 더 현실적이고 복잡한 경우를 보이기 위해 5개의 변수들을 가지고 있는 (그림 5)의 SRG를 예로 들겠다.



(그림 5) SRG의 복잡한 예제와 그로부터 행해진 MST

우리는 SRG를 n개의 공통부분이 없는 하위 그래프(subgraph)들로 나눈 후 같은 하위그래프에 있는 노드들을 같은 메모리 뱅크에 할당함으로서 변수들을 n개의 메모리 뱅크로 배정하는 과정을 따랐다. 우리는 정의 3과 같이 비용을 정의 하였고 이러한 비용 계산 아래 최소의 비용을 가지는 그래프 분리 방법(partition)을 찾음으로서 주어진 SRG에 대한 최적의 메모리 뱅크 배정 결과를 찾으려고 하였다.

정의 3 : V와 E가 각각 노드와 에지들의 집합이라 할 때, 가중치를 가지고 연결되어 있는 그래프 G를 $G=(V,E)$라 표기하자. 또한 w_e를 에지 $e \in E$의 가중치라고 하자. 이 때, 그래프 G의 분리 방법 $P = <G_1, G_2, ...G_n>$을 찾는 문제는 G를 아래의 조건을 만족하는 n개의 공통부분을 가지지 않는 하위그래프 $G_i = (V_i, E_i)$, $1 \le i \le n$ 로 나누는 문제이다.

$v_j, v_k \in V_i$ 이고 $(v_j, v_k) \in E$ 이면 $(v_j, v_k) \in E_i$

이때 분리 방법 P의 비용은 다음과 같이 정의할 수 있다.

$$\sum_{i=1}^{n} \sum_{e \in E_i} w_e$$

최소한의 비용을 갖는 최적의 분리 방법을 찾는 문제는 NP-complete 문제이다. 따라서 우리는 학습적인(heuristic) 방법 중 하나로 greedy approximation을 사용하였다. 이 방법을 통하여 얻은 우리의 알고리즘은 (그림 6)과 같으며 $O(E+VlgV)$의 시간 복잡성을 갖는다. 그러나 실제적으로 우리의 문제에서 E와 V는 비슷한 수를 가지기 때문에 알고리즘은 $O(VlgV)$로 더 빠르게 동작한다. 일반적으로 두개 이상의 데이터 메모리 뱅크를 가지는 ASIP은 존재하지 않기 때문에 알고리즘에서 우리는 n의 값을 2로 가정 하였다. 그러나 이 알고리즘은 n이 2보다 큰 경우로도 쉽게 확장 가능하다.

(그림 6)의 메모리 뱅크 배정 알고리즘에서 우리는 먼저 SRG의 MST를 찾는다. 연결된 그래프(connected graph) G가 주어졌을 때, G의 스패닝 트리는 G의 모든 노드들을 포함하는 사이클이 없는 연결된 하위그래프이다. MST는 스패닝 트리의 모든 에지의 가중치의 합이 G의 다른 스패닝 트리의 에지의 가중치의 합 보다 작지 않은 스패닝 트리이다. 그런데 스패닝 트리는 그것이 두 부분으로 나누어 질 수 있다는 흥미로운 특징을 가진다. 그래서 그래프 G에 대한 스패닝 트리 T가 주어졌을 때 T상의 임의의 노드에서 짝수 거리에 있는 노드들과 홀수 거리에 있는 노드들을 분리함으로서 2개의 하위그래프로 그래프 G를 분할 할 수 있다.

```
Input:     a simultaneous reference graph G_SR = (V_SR, E_SR)
           two memory banks M_X and M_Y
Output:    a set V_SR whose nodes are all colored either with M_X or M_Y
           a set E_UN whose edges are unselected in MST
Algorithm: BB A
  S_T ← Q ← ∅;                          // S_T is a set of MSTs and Q is a priority queue
  for all nodes v in V_SR do unmark v;
  u ← select_unmarked_node_in(V_SR); // Return ⊥ if every node in V_SR is marked
  i ← 1; create a new MST T_i;
  while u ≠ ⊥ do // Find all MSTs for connected subgraphs of G_SR
    mark u;
    E_u ← the set of all edges incident on u;
    sort the elements of E_u in increasing order by weights, and add them to Q;
    while Q ≠ ∅ do
      remove an edge e = (v, z) with highest priority from Q;
      if z is unmarked then T_i ← T_i ∪ {e}; u ← z; break;
      if w is unmarked then T_i ← T_i ∪ {e}; u ← w; break;
    od
    if u is marked then        // All nodes in a connected subgraph of G_SR have been visited
      u ← select_unmarked_node_in(V_SR);// Select a node in another subgraph, if any, of G_SR
      add T_i to S_T; i++; create a new MST T_i;
    fi
  od
  E_UN = Q;          // For generalized memory bank assignment
  for all nodes v in V_SR do uncolor v;
  for every MST T_i ∈ S_T do  // Assign variables in T_i's to memory banks M_X and M_Y
    next_visitors.Q ← ∅;
    m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
    select an arbitrary node v in T_i;
    if m > 0 then            // More nodes have been M_X-colored
      color v with M_Y-color;
    else                     // More nodes have been M_Y-colored
      color v with M_X-color;
    repeat
      for every node u adjacent to v do
        if u is not colored then
          color u with a color different from the color of v;
          append u to next_visitors.Q;
        fi
      v ← extract one node from next_visitors.Q;
    until all nodes in T_i are colored;
  od
  m ← # of nodes in V_SR of M_X-color − # of nodes in V_SR of M_Y-color;
  while m_i > 0 do           // While there are more M_X-colored nodes than M_Y-colored ones
    if ∃ uncolored node v ∈ V_SR then
      color v with M_Y-color; m--;
    else break;
  while m_i < 0 do           // While there are more M_Y-colored nodes than M_Y-colored ones
    if ∃ uncolored node v ∈ V_SR then
      color v with M_X-color; m_i++;
    else break;
  if m_i = 0 then
    for any uncolored node v in V_SR do
      color v alternately with M_X and M_Y colors;
  return V_SR and E_UN;
```

(그림 6) 듀얼 메모리 뱅크를 위한 메모리 뱅크 배정 알고리즘

이러한 특성에 기반을 두어 우리의 알고리즘은 SRG로부터 스패닝 트리를 먼저 생성하고 생성된 스패닝 트리를 이용하여 그래프를 분리하는 방법을 찾는다. 이때 정의 3과 같이 정의한 비용을 줄이기 위하여 MST를 찾는 학습법을 사용한다. 이는 그래프 분리에 이용되어 지는 스패닝 트리의 에지들이 비용 계산에 포함되지 않는다는 사실에 근거한다. 즉, MST를 이용하여 그래프를 분리 한 경우 MST에 포함되는 큰 가중치를 가진 에지를 제거할 수 있게 되어 그래프 분리의 전체적인 비용이 줄어들 수 있다는 사실에 근거한다. 그러나 불행하게도 MST로부터 만들어진 분리 방법이 항상 최적의 해를 생성하는 것은 아니다. 그렇지만 우리의 초기 작업[5]에 따르면 MST 개념이 적은 비용을 가지고 최적에 가까운 분리 방법을 생성하는 결정적인 아이디어라는 것을 알 수 있다. 예로, (그림 5)의 SRG에 대해 우리의 알고리즘은 최적의 분리방법을 찾아낸다.

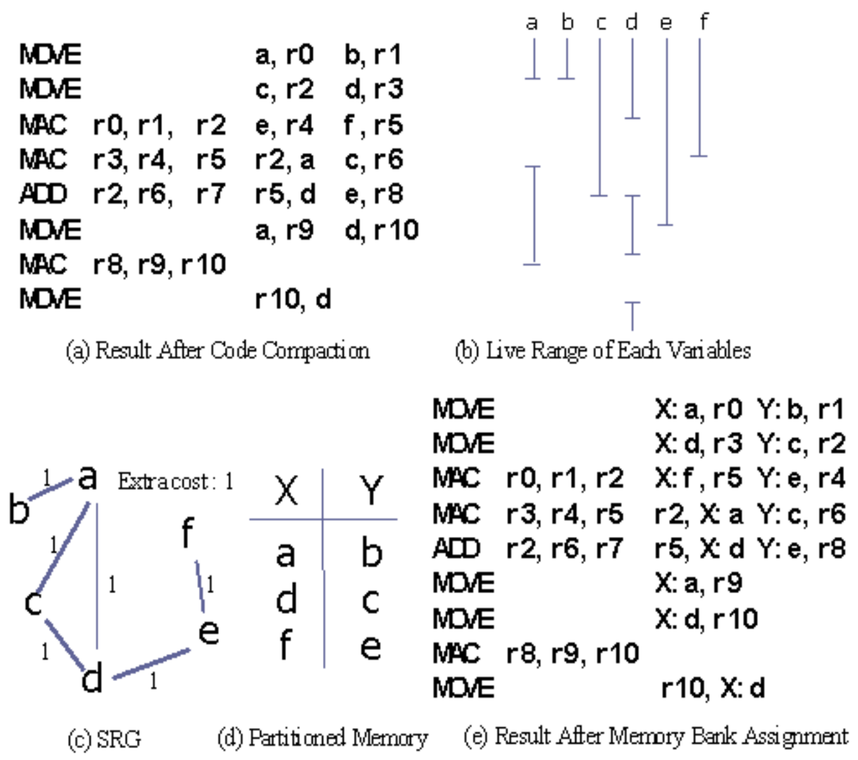
MST를 찾기 위해 우리는 Prim의 MST 알고리즘[11]을 사용했다. 우리의 알고리즘은 기본 단위를 넘어서 적용되는 글로벌 알고리즘으로서 각 노드에 대하여 SRG의 모든 노드들이 표시될 때까지 에지 선택 과정을 반복함으로서 수행된다. 이를 위해 알고리즘은 우선순위를 가지는 큐 Q안에 에지들을 저장하며 반복시마다 에지들의 가중치의 값에 대하여 그들을 정렬한 후 가장 큰 가중치를 가지는 에지를 제거한다. 만약 가장 큰 값을 가지는 에지가 여러 개 있다면 먼저 입력된 에지를 제거한다. 여기에서 우리가 고려해야 할 점은 SRG G_{st}가 모두 연결되어 있는 하나의 그래프가 아니라는 것이다. 따라서 G_{st}의 연결되어 있는 하위그래프 각각에 대하여 하나의 MST가 생성되고 결과적으로 알고리즘은 MST의 집합을 생성하게 된다. 알고리즘에서 고려해야 할 또 다른 사항은 알고리즘에서 표시된 노드 u의 에지들이 항상 Q에 추가되기 때문에 노드 w와 z중 적어도 하나는 반드시 표시가 되어져 있어야만 한다는 것이다. (그림 5(b))는 (그림 5(a))에서 주어진 SRG에 우리의 알고리즘을 적용하여 얻은 스패닝 트리이다. 이 트리에서 짝수 번째 노드에 X 메모리가 홀수 번째 노드에 Y 메모리가 배정 된 걸 알 수 있다.

3.2.2 그래프 컬러링 알고리즘의 사용

그래프 컬러링 접근 방식[4]은 많은 컴파일러에서 레지스터 할당을 위해 전통적으로 사용되어져 왔다. 이러한 그래프 컬러링의 중심적인 아이디어는 변수 대신에 사용 범위(live range)를 레지스터 할당의 후보로 보고 각 변수를 별개의 사용 범위로 나누는 것이다. 우리는 이와 같은 아이디어를 이용함으로서 메모리 뱅크로 할당 될 변수들의 이름에 관련된 제한을 약화시킬 수 있다는 것을 발견했다. 그리고 이를 통해 3.2.1에서 기술한 기본적인 메모리 뱅크 배정문제를 향상 시킬 수 있다는 것 역시 발견 하였다.

그래프 컬러링 알고리즘에서 사용한 방식을 이용하였을 때 메모리 뱅크에 할당되는 것은 변수가 아니라 변수의 사용 범위가 된다. 따라서 우리는 어떤 사용 범위들이 충돌을 일으킴으로서 같은 메모리 뱅크에 배정 될 수 없는지에 대한 정보를 알아야 한다. 이 정보를 모으기 위하여 우리는 먼저 메모리 뱅크 간섭 그래프(memory bank interference graph)라 불리는 그래프를 만들었다. 이 때 만약 같은 변수가 충돌하지 않는 분리된 사용 범위들을 가진다면 이들 각각에 다른 이름이 할당되어 진 후 그들은 각기 다른 메모리에 할당 될 수 있다. 그리고 그래프 컬러링 방식에 의해 생기는 바로 이 부가적인 특징이 메모리 뱅크에 좀 더 효율적으로 변수를 할당하게 하는 것을 가능하게 한다.

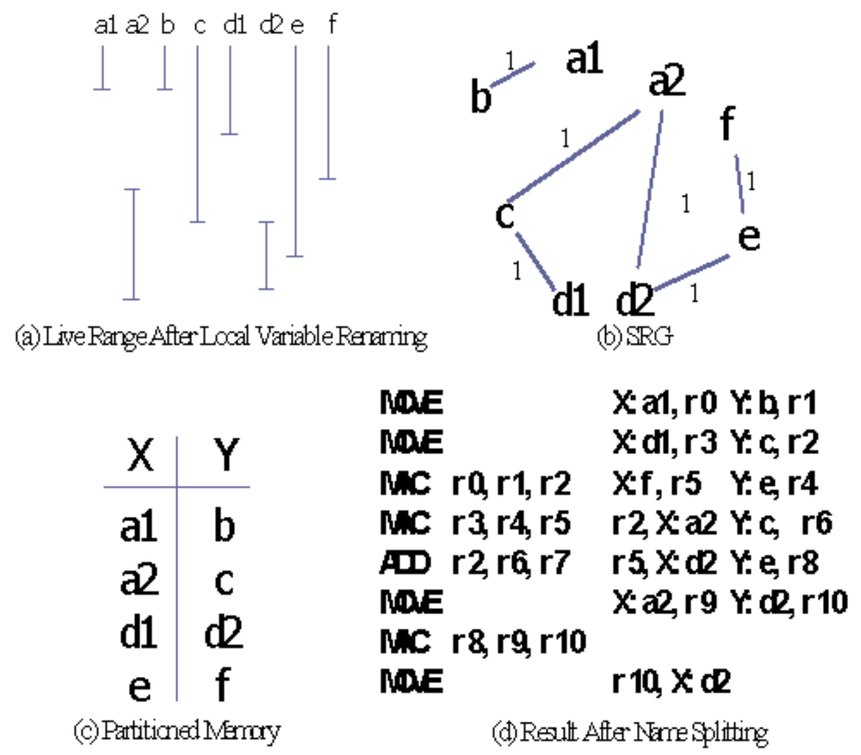
위에서 언급한 특징의 구현뿐만 아니라 전반적인 그래프 컬러링을 이용한 메모리 배정을 돕기 위하여 우리는 이름 분할과 통합(name splitting and merging)이라 불리는 새로운 두 가지 기법을 도입하였다. 이 기법들의 다양한 특징을 보이기에는 그림 4의 예제가 너무 단순하므로 (그림 7)에 보이는 다른 예제를 사용하여 설명하도록 하겠다.



(그림 7) 이름 분할과 통합을 보이기 위한 코드 예제와 데이터 구조

(그림 7(a))는 코드 압축 단계 후에 생성된 코드를 나타내고 7(b)는 7(a)의 코드에 있는 각 변수들의 사용 범위를 보여준다. 여기서 변수 a와 d가 여러 사용 범위를 가지고 있음에 주목하자. (그림 7(c))는 7(a) 코드에 대한 SRG를 나타내며 7(d)는 3.2.1에서 기술한 기본적인 MST 알고리즘으로부터 생성된 메모리 뱅크 배정 결과를 보인다. 위의 예제로부터 a와 d는 같은 메모리 뱅크에 배정되어 있기 때문에 병행 이동으로 사용될 수 없음을 알 수 있다. 이는 메모리 뱅크 배정 후의 결과 코드인 7(e)로부터 확인 할 수 있다.

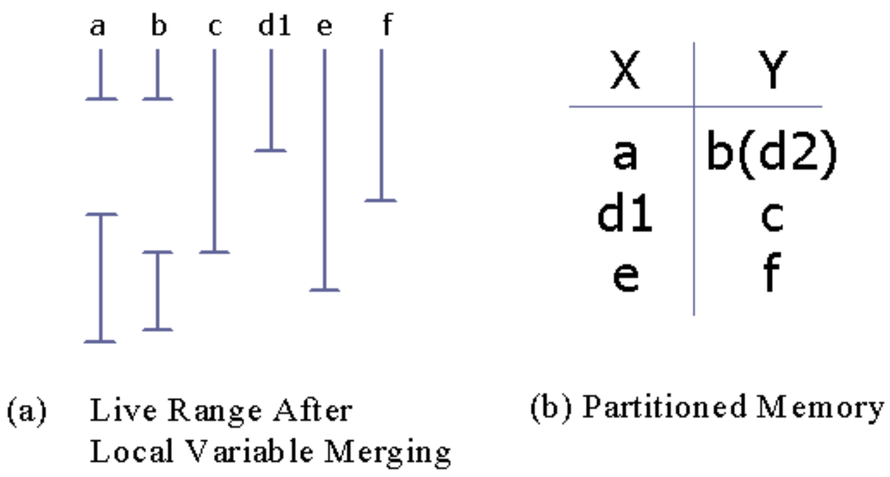
(그림 8)은 어떻게 이름 분할 방법이 메모리 뱅크 배정에 도움을 주는지를 보여준다. 이해를 돕기 위하여 그림 7에서 사용한 것과 같은 예제 코드에 대해 이름 분할 방식을 적용하여 설명하겠다. 이름 분할은 많은 메모리 참조들을 병행 이동으로 묶음으로서 코드 사이즈를 줄이고자 하는 기법이다. 이 기법은 잘 알려진 그래프 컬러링 방법에 기반을 둔다. 따라서 전체 알고리즘을 제시하는 것 대신 (그림 8)에 주어진 예제를 사용하여 알고리즘을 기술하겠다. (그림 8(a))로부터 우리는 변수의 각 사용 범위가 메모리 뱅크에 배정되기 위한 후보라는 것을 알 수 있다. 이를 위해 겹치지 않는 사용 범위를 가지는 두 변수 a와 d는 각 사용 범위에 다른 이름을 줌으로서 여러 개의 변수로 분리되어 졌다.



(그림 8) 로컬 변수를 위한 이름 분할

(그림 8(b))는 이름 분할 후의 변경된 SRG를 보여주며 8(c)는 이에 대한 메모리 뱅크 배정을 보여준다. 이 경우 기존 변수 d에서 분할 된 두 개의 변수 d1, d2가 다른 메모리 뱅크에 배정되는 것을 볼 수 있다. 이로 인해 (그림 7)에서 나타났던 변수 a와 d 사이의 충돌이 없어져 (그림 8 (d))에서 확인 할 수 있는 것처럼 두개의 move 명령어 대신에 병행 이동이 사용되어 졌다. 즉, 이름 분할 방법을 사용함으로서 변수들의 메모리 접근에서의 충돌을 줄 일 수 있었고 그러함으로서 병행 이동의 가능성이 높아 졌다는 것을 확인 할 수 있다.

비록 이름 분할 방법이 더 많은 병행 이동을 사용할 수 있게 함으로서 코드 사이즈를 줄여주지만 (그림 8(c))에서 알 수 있는 바와 같이 더 많은 메모리를 사용하게 된다. 이러한 문제를 해결하기 위해 우리는 이름 분할 후에 이름 통합 과정을 실행하였다. (그림 9)는 (그림 8)의 예제에 이름 통합 과정을 적용한 결과이다. 이름 분리 과정에서 변수 a가 사용 범위에 따라 a1과 a2로 분리되었지만. 두 변수는 같은 메모리 뱅크 X에 배정이 되었다. 더욱이 이 두 변수가 충돌 하지 않기 때문에 이 두변수를 메모리상의 같은 위치로 배정 할 수 있는데 이러한 작업을 이름 통합 과정에서 수행한다.



(a) Live Range After Local Variable Merging

(b) Partitioned Memory

(그림 9) 로컬 변수를 이름 통합

컴파일러는 변수 a와 같이 동일한 변수에 대해서만 이름 통합 방법을 적용하는 것은 아니다. 만약 다른 변수에 대한 사용 범위들이 충돌하지 않는다면 그들 역시 통합 될 수 있다. 예로 (그림 9(a))와 같은 경우 두 변수 b와 d2는 하나의 변수로 통합될 수 있다.

이름 분할과 통합의 키 아이디어는 메모리 뱅크에 배정 될 대상을 변수 대신에 사용 범위로 보는 것이다. 위의 예제로부터 알 수 있듯이 이를 통해 컴파일러는 잠재적으로 사용될 메모리 공간과 실행 명령어의 개수를 줄일 수 있다.

위에서 우리는 그래프 컬러링 방법이 레지스터 배정 때보다 메모리 뱅크 배정시 더 좋은 성능 향상을 가져온다는 것을 보였다. 이는 일반적으로 메모리 뱅크의 개수가 레지스터의 개수보다 훨씬 적기 때문이다. 비록 이름 분리와 통합 과정에서 사용되는 그래프 컬러링 알고리즘이 NP-complete 이지만 메모리 뱅크 배정의 경우 메모리 뱅크의 개수가 작으므로 실제적으로 거의 polynomial 시간 내에 풀려진다. 물론 듀얼 메모리 뱅크에 대하여 최악의 경우 $n2^n$ 시간이 필요로 되어진다. 그러나 이것은 일반적으로 m이 32 이상인 GPP의 레지스터 할당에서 소요되는 $O(nm^n)$보다 훨씬 빠르다. 비록 레지스터 할당이 높은 복잡성을 가지지만 다양한 학습법을 이용하여 polynomial 시간 내에 실행될 수 있다는 것이 실험적으로 증명 되어있다. 우리가 이번 장에서 보인 이름 분할과 통합 역시 마찬가지이다.

3.3 레지스터 배정

변수에 대한 메모리 뱅크 배정이 끝난 후에 물리 레지스터 배정 작업이 수행된다. 우리는 레지스터 배정을 위해서 다시 한번 그래프 컬러링 알고리즘을 사용하였다. 단 이 알고리즘은 앞에서 언급된 ASIP의 비 규칙적인 구조를 다루기 위해 추가된 제한 아래에서 사용되어 졌다. 고려된 제한들을 설명하기 전에 앞 절의 레지스터 클래스 할당 단계에서 임시 레지스터에 레지스터 클래스가 할당되었던 사실을 상기하자. 레지스터 배정 단계에서는 이전 단계에서 할당된 레지스터 클래스 내에 있는 물리 레지스터 중에 하나를 배정하는 작업을 수행한다. 예로 (그림 4)의 임시 레지스터 r0는 앞 단계에서 레지스터 클래스 1이 할당되기 때문에(표 1 참조) X0, X1, Y0, 그리고 Y1의 레지스터만이 배정 될 수 있다.

레지스터 클래스에 더하여 명령어 타입 역시 레지스터 배정 작업에서 고려되어 져야 한다. 예를 들어 병행 이동을 포함하는 명령어는 각 메모리 뱅크 데이터가 미리 정의된 레지스터 집합으로만 이동할 수 있다는 구조적 제한점을 만족해야 한다. 물론 이러한 제한점은 부분적으로는 ASIP의 이종 레지스터 구조 때문이기도 하다. 즉 (그림 4)의 예제에서 비록 임시레지스터 r0에 클래스 1이 할당되었지만 MOVE X:a, r0 명령어의 경우 a가 X 메모리에 할당되어 있어 r0에 할당 가능한 레지스터는 X0, X1의 두 개로 국한된다. 만약 이런 물리 레지스터가 이미 다른 명령어에서 배정이 되어 있다면 레지스터 유출이 발생하게 된다.

우리의 그래프 컬러링 알고리즘은 위와 같은 레지스터와 메모리 뱅크에 관한 제한 조건을 만족하도록 물리 레지스터를 배정한다. (그림 10)은 (그림 4(c))의 코드에 레지스터 배정을 적용한 후의 결과이다. (그림 10)에서 a와 b 같이 변수 이름의 형태로 표시된 메모리 참조가 머신에서 제공하는 어드레싱 모드를 사용한 실제 메모리 참조로 변환된 것을 알 수 있다. 이러한 변환은 레지스터 배정 이후의 단계인 메모리

오프셋 배정단계에서 실행된다. 이와 같은 코드를 얻기 위하여 우리는 Leupers와 Marwedel[7]에 의해 제안된 MWP(maximum weighted path) 알고리즘과 유사한 알고리즘을 적용하였다.

```
MOVE                X: ( r 1) +, X0     Y: ( r 5) +, Y0
MOVE                X: ( r 1) +, A      Y: ( r 5) +, Y1
MAC   X0,  Y0,  A   X: ( r 1) +, X1     Y: ( r 5) +, B
MAC   X1,  Y1,  B   A0, X: ( r 1) +
MOVE                A1, X: ( r 1)       B0, Y: ( r 5) +
MOVE                                    B1, Y: ( r 5)
```

(그림 10) 레지스터 배정과 메모리 오프셋 배정 후의 결과

4. 실험 결과

메모리 뱅크 배정 알고리즘의 성능을 측정하기 위해 DSP56000[10]위에서 구현 된 알고리즘과 벤치마크 프로그램을 가지고 실험을 하였다. 성능은 크기와 시간으로 측정되어졌다. 본 장에서는 우리의 실험 결과를 보이고 기존의 연구와 우리의 결과를 비교한다.

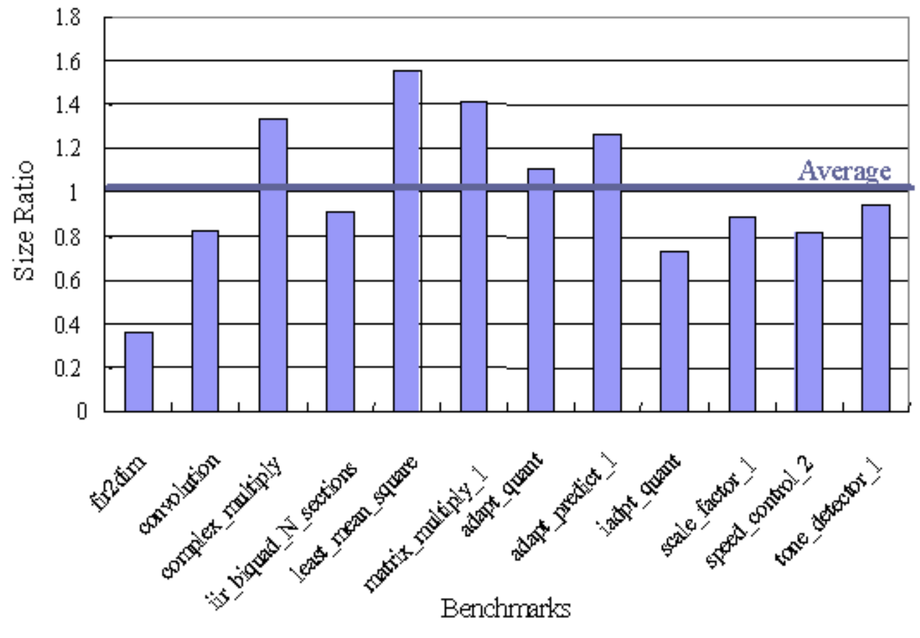
4.1 이전 연구와의 비교

최근까지 ASIP을 위한 코드 생성은 컴파일러 연구에서 많은 관심을 얻지 못했다. ASIP에 대한 가장 두드러진 예는 expression tree를 위한 명령어 선택, 레지스터 할당, 명령어 스케쥴링의 세 가지 기법을 위한 선형 시간 최적 알고리즘을 제안한 Araujo와 Malik[2]의 연구이다. 그러나 ASIP을 위한 대부분의 다른 연구와 같이 그들의 알고리즘 역시 다중 메모리 뱅크 구조를 고려하지 않았다. 우리가 알고 있는 범위한에서 레지스터와 메모리 뱅크 배정의 문제를 처음으로 다룬 연구는 Saghir[12]이다. 그러나 그들 역시 많은 개수의 범용 레지스터를 가지는 프로세서라는 제한을 두었다. 그렇기 때문에 그들의 연구를 이종 레지스터를 가지는 ASIP에 바로 적용하는 것 역시 문제가 있었다.

가장 최근에 이 문제는 Princeton과 MIT[1, 14]에 있는 연구원들에 의해 행해진 SPAM 프로젝트에서 다루어졌다. 사실 SPAM은 현재 우리가 이용할 수 있는 가장 밀접한 관련이 있는 연구이다. 따라서 우리는 SPAM과 비슷한 환경에서 실험함으로서 나온 결과를 SPAM 결과와 비교하려 한다.

4.2 코드 크기의 비교

(그림 11)로부터 SPAM과 우리의 컴파일러에서 동시에 측정 한 프로그램 리스트를 알 수 있다. 이 벤치마크 프로그램들은 ADPCM과 DSPStone [15]의 프로그램들이다. 몇 가지 이유로 해서 우리는 SPAM 컴파일러를 우리의 머신 플랫폼에 성공적으로 포팅할 수 없었다. 그래서 우리는 우리의 결과를 그들의 논문에 있는 결과와 밖에 비교할 수 없었다. 아래에 있는 SPAM 실험 결과 역시 그들의 논문[14]에서 가져온 것이다.

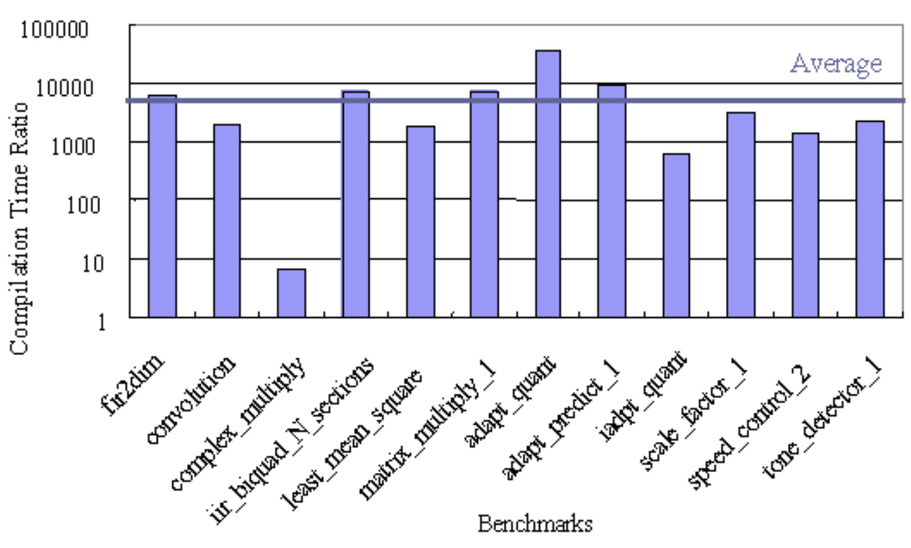


(그림 11) SPAM 코드 크기와 우리의 코드 크기 비율

위 그림은 우리의 코드와 SPAM 코드 사이의 크기 비율을 보인다. SPAM 코드 사이즈를 1로 하였고 우리의 코드 사이즈를 SPAM 코드 사이즈에 대해 정규화 했다. 이 그림으로부터 우리는 코드 크기에 있어서 전반적으로 우리의 결과가 SPAM의 결과에 필적하다는 것을 알 수 있었다. 사실 12개의 벤치마크 중에서 7개는 SPAM 코드 보다 더 작은 크기의 결과를 얻었다. 이 결과는 우리의 메모리 뱅크 배정 알고리즘이 대부분의 경우에 있어서 SPAM의 알고리즘만큼 효율적이라는 것을 보여준다.

4.3 컴파일 시간 비교

코드 크기와 달리 컴파일 속도에 있어서 두 컴파일러는 상당한 성능 차이를 보였다. 그들의 논문[14]에 따르면 SPAM의 모든 실험은 8개의 프로세서와 1GB RAM을 장착한 Sun Microsystems의 Ultra Enterprise에서 행해졌다. 불행하게도 우리는 정확하게 같은 실행 환경을 갖추지 못했다. 대신에 우리는 두개의 프로세서와 2GB RAM을 가지고 있는 같은 Sun Microsystems Ultra Enterprise에서 실험하였다. (그림 12)는 이 환경 하에서 측정한 컴파일 속도를 나타낸다. (그림 12)로부터 우리의 컴파일 시간이 SPAM에 비해 대략 10^3에서 10^4정도 빠름을 알 수 있다. 이렇게 큰 컴파일 시간차가 머신 플랫폼의 차이에도 불구하고 나타나는 것으로부터 컴파일 속도에 있어서 우리의 접근 방식이 그들의 방식보다 이점을 가진다고 생각할 수 있다.



(그림 12) SPAM 컴파일러와 우리 컴파일러의 컴파일 시간의 비율(로그 스케일)

더욱이 SPAM의 컴파일 시간은 우리와는 반대로 큰 응용프로그램에 대하여 크게 증가할 수 있다. 우리는 이러한 SPAM 컴파일러에서의 긴 컴파일 시간이 레지스터 할당과 메모리 뱅크 배정을 하나의 결합된 단계에서 수행하려고 하는 접근 방식에서 나타남을 알았다.

그들의 접근 방식에서 변수들은 메모리 뱅크에 배정되는 동시에 물리 레지스터를 할당받는다. 그들은 이러한 결합된 접근 방식을 지원하기 위해 여러 제한점을 동시에 표현하는 제약 그래프(constraint graph)를 만든다. 불행하게도 한 그래프를 통해 여러 종류의 제한점을 해결하려고 한 그들의 노력은 그들의 문제를 단지 NP-complete 알고리즘에 의해서만 다룰 수 있는 전형적인 다중 변형 최적화 문제로 만들었다. 더욱이 이종 레지스터와 다중 메모리 뱅크에서의 제한이 최적의 결과를 얻기 위해서 꼭 필요하므로 그들의 결합 방식에서 다중 변형 제한들은 피할 수 없는 요소이다. 결과적으로 그런 많은 시간이 걸리는 알고리즘을 피하기 위해 그들은 Monte Carlo 접근 방식에 기반 한 simulated annealing이라 불리는 학습적 알고리즘을 사용하였다. 그러나 이러한 학습법의 사용에도 불구하고 그들의 방법의 경우 일반적인 코드 크기의 프로그램을 컴파일하기 위해 1000초 이상의 시간이 걸린다는 것을 그들의 논문[13,14]을 통해 알 수 있었다. 이것은 많은 제한점을 가지는 제약 그래프가 코드의 크기 증가에 따라 빠르게 커지고 복잡해지기 때문이다.

그들의 느린 컴파일 속도는 결합된 접근 방식에서 나타나는 내부 복잡성에 원인이 있다. 반대로 우리의 컴파일 시간은 심지어 큰 벤치마크 프로그램에 대하여도 짧게 유지된다. 이는 듀얼 메모리 뱅크 시스템을 위한 코드 생성 과정에서 발생하는 문제를 개별적으로 풀어나가는 다양하고 빠른 학습적 알고리즘을 적용하였기 때문이다. 자세히 설명하자면 우리는 레지스터 할당을 코드 압축과 메모리 뱅크 배정으로 분리 하여 수행하였다. 그리고 이 분리된 두 작업이 모두 수행 된 이후에 물리 레지스터를 배정하는 작업을 수행하였다. 연구 초반에 물리 레지스터 배정과 메모리 뱅크 배정이 분리되어 있는 제한 때문에 오히려 코드 성능이 하락할 수 있지도 않겠냐는 우려의 소리를 들었다. 그러나 본 연구 결과로부터 조심스러운 분리는 종종 산업용 컴파일러에서 중요한 요소인 컴파일 속도는 극대화 하는 반면 위의 약점은 완화시킬 수 있다는 결론을 얻었다.

4.4 실행 시간의 비교

실행 시간에서 코드 크기 감소의 영향을 측

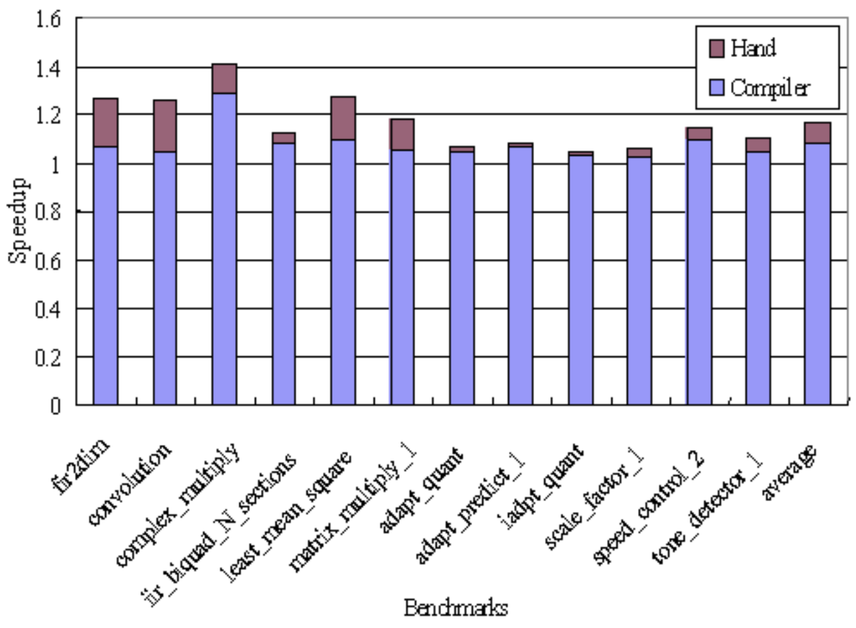
정하기 위해 우리는 다음과 같은 세 가지 버전의 코드를 생성하였다.

최적화되지 않은 코드 그림 2에서 보는 바와 같이 명령어 선택 단계 직후에 생성된 압축되지 않은 코드

컴파일러에 의해서 최적화 된 코드 본 논문 3장의 기법들을 사용하여 컴파일러에 의해 압축된 그림 10과 같은 코드

사람에 의해서 최적화 된 코드 손으로 직접 최적화한 코드. 우리는 DSP56000에서 이상적인 벤치마크 성능을 얻기 위해 컴파일러의 입력으로 제공되는 코드를 직접 최적화하였다.

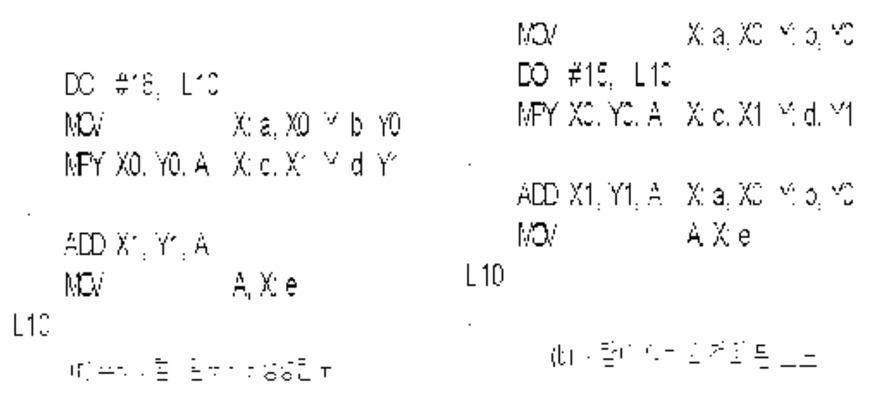
(그림 13)은 실행시간에 대한 비교이다. 이 그림은 최적화되지 않은 코드의 실행 시간에 대하여 컴파일러와 사람에 의해서 최적화 된 코드가 얼마만큼의 속도 향상을 보이는지를 나타내고 있다. 예를 들어 complex_multiply 벤치마크의 경우, 컴파일러에 의해서 최적화 된 코드는 최적화되지 않은 코드에 비하여 23%의 속도 향상을 얻었고 사람에 의해서 최적화 된 코드의 경우에는 부가적으로 9%의 속도 향상을 얻었다. 즉, 최적화되지 않은 코드에 비하여 전체적으로 32%의 속도 향상을 얻었다.



(그림 13) 최적화되지 않은 코드에 대한 컴파일러와 사람에 의해서 최적화 된 코드의 실행시간 속도 증가

(그림 13)으로부터 우리는 최적화되지 않은 코드에 대해 컴파일러와 사람에 의해 최적화된 코드가 각각 7.9%와 16.7%의 평균적인 속도 향상을 내는 것을 알 수 있다. 따라서 컴파일러에 의해 생성된 코드가 사람에 의해 최적화된 코드에 대해 절반의 성능을 보임을 알 수 있다. 비록 이 결과가 만족스럽지는 않지만 우리는 위의 그림으로부터 12개 중 6개의 벤치마크 프로그램에 대해서는 사람에 의해 최적화된 코드와 거의 필적할 만한 성능을 컴파일러가 낸다는 것 역시 확인 할 수 있었다.

또한 우리의 컴파일러는 fir2dim, convolution 그리고 lest_mean_square와 같은 여러 벤치마크에 대해서 성능 향상의 많은 여지를 보이고 있다. 우리의 분석에 따르면 컴파일러와 사람에 의해 최적화 된 코드에서 발생하는 성능 차이의 주요 원인은 루프처리와 관련되었다. 즉 루프를 효율적으로 다루는 능력의 부족으로 인해 실행시간 차가 발생하였다. 이를 보이기 위해 그림 14와 같은 예제를 제시한다. 이 예제는 루프 최적화에서 필요로 되는 전형적인 소프트웨어 파이프라이닝을 보인다.



(그림 14) 컴파일 된 코드와 사람에 의해서 최적화된 코드 사이의 압축 비교

(그림 14)에서 변수 a와 b를 위한 병행 이동은 그들과 MPY 명령어 사이에 존재하는 의존관계 때문에 ADD를 포함하는 명령어 워드로 압축(compaction)될 수 없다. 그러나 루프의 시작 전에 병행 이동의 복사 본을 놓는다면 ADD와 병행이동은 하나의 명령어로 압축 될 수 있다. 비록 이 최적화 기법이 전체 코드 사이즈를 감소시키지는 않지만 루프 내에서 하나의 명령어를 제거함으로서 전체 실행시간은 크게 줄인다. 특히 대부분의 실행 시간이 루프에서 소요

되기 때문에, 이 예제는 우리에게 소프트웨어 파이프라이닝과 같은 더 향상된 루프 최적화 없이는 실행 속도에 있어서 우리의 컴파일러가 사람에 의해서 최적화된 코드를 따라갈 수 없다는 사실을 말해준다. 현재 이 작업은 우리의 추후 작업으로 진행 중에 있다.

5. 결론과 앞으로의 작업

본 논문에서 우리는 듀얼 메모리 구조를 지원하기 위해 여러 단계로 분리된 접근 방식을 사용하였다. 또한 부가적인 기법으로서 이름 분리와 통합이라는 두 가지 방법을 제시하였다. 그리고 우리의 연구 결과를 SPAM의 결과와 비교함으로서 우리의 접근 방식이 가지는 장점과 단점에 대하여 언급 하였다. 실험 결과로부터 우리의 컴파일러가 코드 크기에 대하여 기존 연구와 필적할 만한 결과를 가진다는 것을 알 수 있었다. 또한 여러 단계로 분리된 접근 방식이 듀얼 메모리 뱅크를 위한 데이터 할당 알고리즘을 단순화하여 상당히 빠른 컴파일 속도를 얻게 한다는 것 역시 확인 할 수 있었다. 이를 통하여 메모리 뱅크로 잘 배정된 스칼라 변수가 실행 속도 향상이라는 결과를 낸다는 것을 확인 할 수 있었다.

그러나 결과 분석을 통하여 현재의 기법이 가지고 있는 몇 가지 제한 사항 역시 발견되어 졌다. 이 중 하나는 우리의 방식이 스칼라 변수에 제한되어 있다는 것이다. 따라서 많은 프로그램에서 나타나는 배열위에서 수행되는 연산에 대해서는 큰 성능향상을 기대할 수 없다.이것은 (그림 11, 13)에서 보인 결과로부터 확인할 수 있다. 이 그림을 보면 코드 크기에 있어서는 눈에 띄는 차를 볼 수 있지만 실행 시간의 경우에서는 심지어 직접 사람이 최적화한 코드의 경우에도 큰 성능 향상을 보이지 않는 다는 것을 알 수 있다. 이것은 성능에 있어서 스칼라 변수의 영향은 코드에서 변수들이 차지하는 공간에 비하여 상대적으로 적기 때문이다. 다른 제한점 중 하나는 함수 호출과 관련 있다. 함수 호출시 메모리를 통해서 매개 변수를 전달할 때 호출된 함수에 맞게 메모리 뱅크들을 배정해야 한다. 그러나 이 기능은 호출자가 호출되는 함수의 메모리 접근 방식을 알아야 하기 때문에 함수간의 분석이 필요로 되어진다. 이 작업은 아직 구현되지 않은 것으로 추후 추가될 예정이다. 또 다른 제한점으로는 4.4절에서 보인바와 같이 출력 코드의 실행 시간을 향상하기 위해 루프 최적화 기법이 구현되어야 한다는 것이다.

참고 문헌

[1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang, and A. Wang. *Challenges in Code Generation for Embedded Processors*, pages 48-64. In Marwedel and Goossens [9], 1995.

[2] G. Araujo and S. Malik. Code Generation for Fixed-point DSPs. *ACM Transactions on Design Automation of Electronic Systems*, 3(2): 136-161, April 1998.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computers*, Nov. 2001.

[4] G. Chaitan. Register Allocation and Spilling via Graph Coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 201-207, June 1982.

[5] J. Cho, J. Kim, and Y. Paek. Efficient and Fast Allocation of On-chip Dual Memory Banks. In *6th Workshop on Interaction between Compilers and Computer Architectures*, Feb. 2002.

[6] S. Jung and Y. Paek. The Very Portable Optimizer for Digital Signal Processors. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 84-92, Nov. 2001.

[7] R. Leupers and P. Marwedel. Algorithms for Address Assignment in DSP Code Generation. In *Internaltional Conference on*

Computer-Aided Design, 1996.

[8] C. Liem. *Retargetable Compilers for Embedded Core Processors*. Kluwer Academic Publishers, 1997.

[9] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluwer Academic Publishers, 1995.

[10] Motorola Inc., Austin, TX. *DSP56000 24-Bit Digital signal Processor Family Manual*, 1995.

[11] R. Prim. Shortest Connection Networks and Some Generalizations. *Bell Systems Technical Journal*, 36(6):1389-1401, 1957.

[12] M. A. R. Saghir, P. Chow, and C. G. Lee. Exploiting Dual Data-Memory Banks in Digital Signal Processors. *ACM SIGOPS Operating Systems*, pages 234-243, 1996.

[13] A. Sudarsanam. *Code Optimization Libraries For Retargetable Compilation For Embedded Digital Signal Processors*. PhD thesis, Princeton University Department of EE, May 15, 1998.

[14] A. Sudarsanam and S. Malik. Simultaneous Reference Allocation in Code Generation for Dual Data Memory Bank ASIPs. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):242-264, April 2000.

[15] V. Zivoljnovic, J.M. Velarde, C.Schager, and H. Meyr. DSPStone-A DSP oriented Benchmarking Methodology. In *Proceedings of International Conference on Signal Processing Applications and Technology*, 1994.

조 정 훈

학사, (1996), EECS, KAIST

석사, (1998), EECS, KAIST

박사, (2003), EECS, KAIST

현재 Hynix 반도체 재직

관심분야 : DSP 컴파일러, 재겨냥성 최적화 컴파일러, Dependence Analysis

백 윤 흥

서울 대학교 컴퓨터공학과에서 학사와 석사 학위를 받음

제3사관학교에서 육군 소위 임관 직후 미국 일리노이 주립 대학교 전산학과에 입학하여 박사학위 받음

1997년, 미국 일리노이 주립대 전산과 연구 교수

1997년～1999년, 미국 뉴저지 주립 공대 전산과 조교수

1999년～2003년, 한국과학기술원(KAIST) 전기 및 전자공학과 부교수

2003년～현재, 서울대학교 전기공학부 부교수

관심분야 : 재겨냥성을 지닌 최적화 컴파일러, 프로세서 설계 소프트웨어, 내장형 프로세서 구조

최 준 식

1999, 서울대학교 전기·컴퓨터 학부 학사

1999~2002, 파인디지털(주), 연구원

2003～현재, 서울대학교 전기컴퓨터 학부 석사과정

관심분야 : 재겨냥성 최적화 컴파일러, 내장형 프로세서