http://dx.doi.org/10.3745/KIPSTA.2012.19A.1.045

Register Pressure를 고려한 다중 출력 명령어를 위한 개선된 코드 생성 방법

윤 종 희[†] · 백 윤 흥[††] · 고 광 만[†††]

요 약

최근 내장형 프로세서가 다양한 휴대 기기에서 사용이 급격히 증가됨에 따라, 빠른 수행 속도와 저전력을 지원하는 내장형 프로세서를 제작하기 위해 대상 응용 프로그램에 최적환 된 설계가 요구되고 있다. 이를 위해, 프로세서들은 다중 출력 명령어형태의, 특별한 명령어들을 추가하고 있다. 여기서 다중 출력 명령어란 다수의 결과값을 병렬적으로 출력하는 명령어를 말한다. 하지만, 이러한 다중 출력 명령어들은 기존 컴파일러에서 생성하지 못하는 문제점이 있다. 그래서, 이를 위한 컴파일 알고리즘이 개발되었지만, 이 알고리즘에서는 register pressure를 고려하지 않아서 최적의 성능을 발휘할 수가 없었다. 본 논문에서는 register pressure를 고려하는 알고리즘을 새롭게 제안하고, 그 결과 기존 알고리즘에 비해서 평균 3%의 코드 사이즈 감소와 2.7% 수행 시간 향상을 더 이룰 수 있었다.

키워드 : 명령어 선택, 컴파일러, 다중 출력 명령어, 응용 특화 프로세서

Register Pressure Aware Code Selection Algorithm for Multi-Output Instructions

Jonghee M. Youn[†] · Yunheung Paek[††] · Kwang-Man Ko[†††]

ABSTRACT

The demand for faster execution time and lower energy consumption has compelled architects of embedded processors to customize it to the needs of their target applications. These processors consequently provide a rich set of specialized instructions in order to enable programmers to access these features. Such an instruction is typically a *multi-output instruction* (MOI), which outputs multiple results parallely in order to exploit inherent underlying hardware parallelism. Earlier study has exhibited that MOIs help to enhance performance in aspect of instruction counts and code size. However the earlier algorithm does not consider the register pressure. So, some selected MOIs introduce register spill/reload code that increases the code size and instruction count. To attack this problem, we introduce a novel iterated instruction selection algorithm based on the register pressure of each selected MOIs. The experimental results show the suggested algorithm achieves 3% code-size reduction and 2.7% speed-up on average.

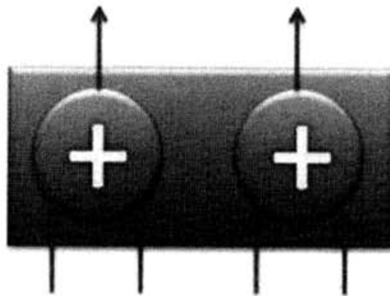
Keywords : Instruction Selection, Compiler, Multi-output Instruction, ASIP

1. 서 론

최근 내장형 프로세서는 다양한 형태의 제품에 포함되고 있다. 특히 스마트폰이나 테이블릿 장치등이 시장을 장악해 나가면서 더욱 더 그 역할이 중요해지고 있다. 이런 내장형 프로세서들은 대체로 휴대용 장치에 사용되기 때문에 한정된 배터리 용량으로 인해서 사용 가능한 시간이 프로세서의 크기와 소비 전력 등에 많은 영향을 받게 된다. 또한 사용자들은 점점 더 고성능, 고기능을 요구하고 있다. 그래서 이렇게 사용자들이 요구하는 저전력 고성능 프로세스를 만들기 위해서, 내장형 프로세서는 파워, 프로세서의 크기, 수행 시간 등의 여러 가지 제약 조건에 따라 설계되고 개발 되고 있다. 그 중 대표적인 형태가 바로 응용 특화 프로세서 (Application Specific Instruction Set Processor)이다. 각각의 프로세서를 활용하려고 하는 분야에 따라 프로세서의 특성을 설계해서 고성능 저전력 요구를 만족시키려는 것이다.

※ 본 연구는 2011년도 정부(교육과학기술부)의 재원으로 한국과학재단의 국가지정연구실사업(No. 2011-0018609), 교육과학기술부/한국과학재단 우수연구센터 육성사업(과제번호 2011-0000975), 2011년 정부(교육과학기술부)의 재원으로 한국연구재단의 기초연구사업(No. 2011-0012522)의 지원을 받아 수행되었음.
† 정 회 원 : 강릉원주대학교 컴퓨터공학과 강의전담교수
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수(교신저자)
††† 종신회원 : 상지대학교 컴퓨터정보공학부 교수
논문접수 : 2011년 10월 25일
수 정 일 : 1차 2011년 12월 20일
심사완료 : 2011년 12월 20일

이러한 응용 특화 프로세서에서 대표적으로 사용하는 방법이 새로운 명령어들을 프로세서에 추가하는 것이다. 새롭게 추가되는 명령어들을 다중 출력 명령어(Multi-Output Instruction) 형태를 가지는 경우가 많다. 이 다중 출력 명령어는 보통의 명령어가 두서너 개의 피연산자를 입력으로 받아 하나의 출력으로 결과를 만드는 것과 달리, 여러 개의 피연산자를 입력으로 받아서 두 개 이상의 출력을 만든다. 그리고 그 내부에서 각각의 연산들이 병렬로 수행되는 경우가 많다. 예를 들면, (그림 1) Dual add 명령의 경우 4개의 입력을 받아서 더하고 결과로 2개 값을 출력하게 된다. 이 때 내부에서는 2개의 입력을 받아 더한 후 그 결과를 출력하는 명령어 두 개가 동시에 병렬적으로 수행된다.



(그림 1) Dual ADD

이와 같은 명령어들은 실제 많은 프로세서에 존재한다. 예를 들면, 소니 pDSP 에는 메모리에서 두 개의 값을 동시에 읽어와 두개의 레지스터에 저장하는 PLDXY r1, @a, r2 @b와 같은 명령어가 존재한다. 이렇게 다양한 목적을 성취하기 위해서 프로세서들에서는 여러 다중 출력 명령어들을 지원하고 있다. 하지만 문제는 이러한 다중 출력 명령어들을 기존 컴파일러들이 제대로 지원하지 못하기 때문에 실제 프로세서 위에서 동작할 프로그램에서는 효율적으로 활용되지 못하고 있다[1]. 기존 컴파일러들은 이런 다중 출력 명령어들을 인라인 어셈블리(Inline Assembly) 또는 컴파일러 인지 함수(Compiler Known Function 또는 Intrinsic Function)등을 사용해서 프로그래머가 직접 소스코드를 수정해서 지원하는 방식을 지원하고 있는데, 이러한 방법은 소스 코드를 직접 모두 수정해야 하는 문제와 수정된 소스 코드는 다른 프로세서에서는 사용할 수 없고, 또한 새로운 다중 출력 명령어들이 프로세서에 추가 되었을 경우 또 다시 소스코드를 수정해야만 한다. 그래서 본 논문에서는 먼저 컴파일러에서 다중 출력 명령어들을 처리할 수 있는 알고리즘[4][8]을 소개한다. 이 알고리즘을 통해서 소스코드를 수정하지 않고도 다중 출력 명령어들을 효율적으로 사용할 수 있게 된다. 또한 초기 버전의 알고리즘이 다루지 못하고 있는 부분인 Register Pressure를 고려한 새로운 알고리즘도 소개한다.

본 논문의 구성은 2장에서 기존 관련 연구들을 소개하고 새롭게 제안된 알고리즘을 3장에서 보이고, 4장과 5장에서는 각각 실험 결과와 결론 맺는다.

2. 다중 출력 명령어를 위한 코드 선택(Code Selection) 알고리즘

컴파일러에서 코드 생성 알고리즘은 소스 코드로부터 만들어진 중간 코드(Intermediate Representation, IR)의 각 operation node에 프로세서가 가지고 있는 명령어 중 매칭될 수 있는 것들을 나열해 주는 label과 그 중에서 각 노드마다 최상의 성능을 취할 수 있는 전체 명령어를 선택해주는 cover 두 단계로 이루어진다. 이는 아래와 같은 트리 문법(tree grammar)를 바탕으로 실제 중간코드 그래프(IR Graph)에서 위의 두 단계가 이루어진다.

NT -> opcode (NT, NT)

여기서, NT는 Non-terminal들을 의미하고 opcode는 ADD, MUL, SUB등의 명령어의 연산(operation)을 의미한다. 논문[1]에서는 다중 출력 명령어를 컴파일러에서 처리하기 위해서 아래 용어를 정의했다.

- 일반법칙(Rule): 트리 문법에서 각 명령어의 패턴을 나타냄.
- 간단법칙(Simple rule) : 일반 적인 트리 패턴을 나타냄.
- 복잡법칙(Complex rule): 여러 개의 간단법칙으로 구성된 법칙.
- 분리법칙(Split rule): 복잡법칙을 구성하고 있는 하나의 멤버를 나타내는 간단법칙.

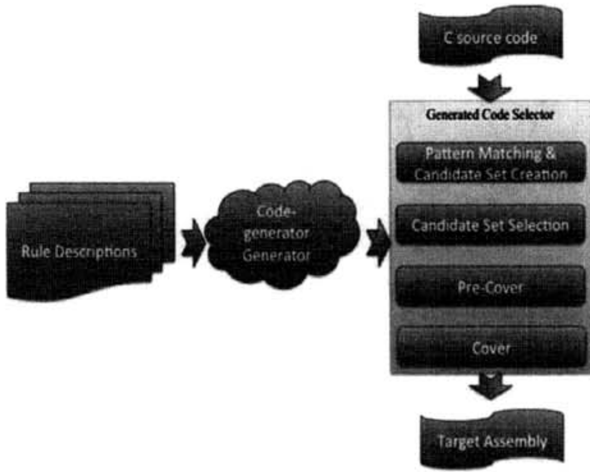
이를 바탕으로 다중 출력 명령어를 위한 트리 문법 법칙을 아래와 같이 정의 하였다.

NT NT ... -> opcode(NT, NT) opcode(NT, NT) ...

즉, 기존의 트리 문법에서는 하나의 명령어당 하나의 opcode와 하나의 NT만을 표현할 수 있었지만, 확장된 문법에서는 원하는 만큼 다양하게 표현 할 수가 있다.

이를 바탕으로 (그림 2)와 같이, 위의 새로운 문법 표현 방식을 바탕으로 코드 생성 알고리즘을 구성해서 다중 출력 명령어를 컴파일러에서 처리 할 수 있도록 하였다. 첫 번째 단계는 앞에서 설명한 트리 문법을 사용해서 모든 명령어들을 기술해 놓은 법칙 서술(rule description)을 입력으로 받아 자동으로 컴파일러를 생성해주는 단계이다. 이것은 기존의 코드 생성기 생성기(code-generator generator) 로 유명한 iburg[5], olive[6] 등의 툴들을 기반으로 만들어졌다. 기존의 툴들은 label와 cover만으로 구성된 코드 생성기를 생성해 주었지만, 본 논문에서 제안하는 툴은 다중 출력 명령어를 컴파일러에서 지원하기 위한 다양한 모듈들을 추가로 자동으로 생성해주게 된다.

생성된 각 모듈들을 간단히 설명하면, 첫 단계인 법칙 서술에서부터 코드 생성기를 생성하는 부분으로, 새로운 표현 방식으로 표현된 복잡법칙을 분리법칙 단위로 나누고, 두 번



(그림 2) 다중 출력 명령어를 위한 코드 선택 알고리즘

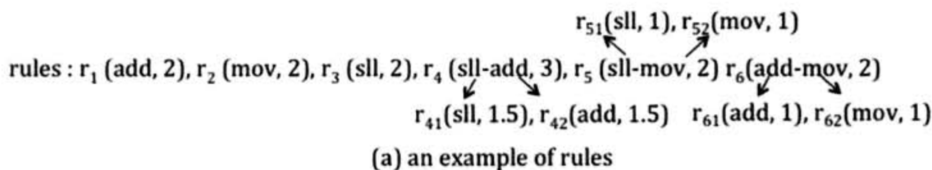
째 단계에서 이 분리법칙을 바탕으로 소스코드로부터 만들어진 DFT를 labeling하게 된다. 그리고 나서 분리법칙이 label된 DFT의 각 노드 간의 의존성(dependency)등을 체크하여 복잡법칙이 될 수 있는 후보군(candidate set)을 만들게 되고, 세 번째 단계에서 만들어진 후보군중에서 최적의 복잡법칙들을 Maximum-independent weight set problem(MWISP)을 이용해서 선택하게 된다. 4번째 단계인 Pre-Cover는 앞에서 선택된 최적의 복잡법칙이 실제로 cover될 수 있는지 없는지를 미리 확인하여 다음 단계인 cover에서 문제가 생기지 않도록 해주고, 마지막 단계인 Cover에서 실제로 복잡법칙에 따라 다중 출력 명령어들을 선택하게 해준다.

(그림 3(a))는 현재 가지고 있는 명령어들에 대한 법칙과 각 명령어들의 반응시간(latency)를 보여준다. 여기서 r_4, r_5, r_6은 다중 출력 명령어에 해당하고, 자동으로 각각의 분리법칙들이 생성되어 있다. 그리고 (그림 3(b))에는 우리가 컴파일해야 할 코드의 중간코드를 보여주고, 각 노드를 구분하기 위해서 사각형 숫자를 붙여 놓았다. 각 분리법칙에 해당하는 노드들의 테이블을 만들어, (그림 3(c))와 같이 다중 출력 명령어로 선택될 수 있는 노드들의 후보를 만든다. 예를 들어 노드 1번과 노드 4번은 다중 출력 명령어 r_4가 될 수 있다. 그래서 우리는 이런 다중 출력 명령어가 될 수 있는 모든 노드들의 조합들을 앞의 표를 통해서 완성한다. 하지만, 이중에는 절대로 다중 출력 명령어가 될 수 없는 조합이 있는데, 바로 후보 A, C, H와 같은 것들이다. 왜냐하면, 후보 A의 경우 노드 1번과 2번의 조합을 뜻하는데, 노드 1번과 2번 사이에는 데이터 의존성이 존재하기 때문에 다중 출력 명령어가 될 수 없다. 그래서 후보군 중에서 이런 데이터 의존성이 존재하는 것들을 먼저 제외 시킨다.

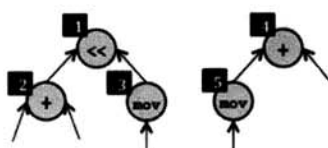
또한, 이 후보들 사이에 두가지 간섭관계가 존재한다. 하나는 후보 B와 D 사이에서 볼 수 있는데, 두 후보 모두 노드 1번을 포함하고 있다. 결국 둘 중 하나를 선택해야 한다. 또 다른 간섭관계는 후보 D와 G 사이에서 나타난다. 두 후보 사이에 공통으로 적용되는 노드가 없어서 앞에서 설명한 간섭관계는 존재하지 않지만, 두 후보는 동시에 선택될 수 없다. 왜냐하면, 만약 후보 D(노드 1번, 5번 조합)가 선택되면, 후보 G(노드 3번, 4번 조합)는 수행 될 수 없게 되는데, 노드 3번의 결과를 노드 1번이 사용해야 되고, 노드 4번은 노드 5번의 결과를 사용해야 한다. 즉 둘 사이에 스케줄링이 불가능한 간섭이 존재한다. 우리는 이런 간섭 관계를 (그림 4)와 같이 그래프를 통해서 표현했다. 첫번째 간섭관계는 실선으로 나타냈고, 두번째 간섭관계는 점선으로 표시하였다. 이 그래프에서 서로 연결된 노드들은 동시에 선택될 수 없다는 것을 나타낸다. 이제 우리는 이중에서 어떤 후보들을 선택해야 하는가하는 문제를 해결해야 한다. 그러기 위해서 우선 각 각 후보 노드들의 가중치(weight)를 아래와 같이 지정해서 최상의 코드를 선택할 수 있도록 하였다.

$$w_m = B(r) = \sum C_s(r_i) - C_c$$

여기서 C_c는 다중 출력 명령어의 반응시간이고, $C_s(r_i)$는 다중 출력 명령어를 구성하는 분리법칙들의 반응시간으로



(a) an example of rules



Split rule	Node Pointer	
r_{41}	1	
r_{42}	[illegible]	[illegible]
r_{51}	1	
r_{52}	[illegible]	[illegible]
r_{61}	[illegible]	[illegible]
r_{62}	[illegible]	5

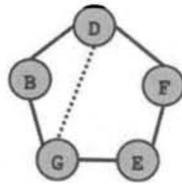
(b) an example of IR tree

Cannot be executed in parallel due to the data dependences

~~A(1,2)~~ B(1,4) ~~C(1,3)~~ D(1,5)
E(2,3) F(2,5) G(3,4) ~~H(4,5)~~

(c) MOI Candidates (MOICs)

(그림 3) MOI Candidates 선택예제



(그림 4) 간섭 그래프(Interference Graph)

이것은 분리법칙과 동일한 형태의 간단법칙의 반응시간을 사용해서 결정하였다.

아래와 같이 우리는 각 후보들의 가중치를 계산하였다.

$$B(r_4) = (C_s(r_{41}) + C_s(r_{42})) - C_c = (2+2)-3 = 1$$
$$B(r_5) = (C_s(r_{51}) + C_s(r_{52})) - C_c = (2+2)-2 = 2 \quad w_B = 1$$
$$B(r_6) = (C_s(r_{61}) + C_s(r_{62})) - C_c = (2+2)-2 = 2 \quad w_D = w_E = w_F = w_G = 2$$

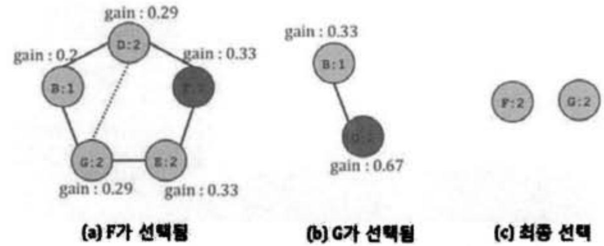
이중에서 우리는 weight의 총합을 최대로 하는 서로 간섭하지 않는 후보를 선택하여야 한다. 이를 위해서 우리는 MWISP의 문제의 솔루션을 통해서 이를 해결하였다[9]. 이 알고리즘은 아래와 같은 이득(gain)함수를 통해서 각 노드이 이득을 계산하고 그중 가장 큰 이득을 가지는 노드를 선택하고, 선택된 노드와 연결관계가 있는, 즉 간섭관계가 있는 노드들을 그래프에서 제거하고 다시 이득을 계산하고 선택하는 과정을 반복한다.

$$\sum_{v \in V(G)} \frac{W(v)}{\sum_{u \in N_{G^+}(v)} W(u)}$$

$W(v)$: the weight of node v
$V(G)$: the set of nodes in graph G
$N_{G+}(v)$: the neighborhood of v & v

(그림 5)에서는 우리의 예제에서 최종 후보를 MWISP를 통해서 선택하는 예제를 보여주고 있다. 우리의 예제에서는 최종적으로 후보 F,G가 다중 출력 명령어로 선택되었다.

이후에는 기존의 코드 생성 알고리즘과 동일하게 labeling과 covering 단계를 통해서 최종 코드가 생성된다. 그중에 pre-cover라는 단계가 추가 되는데, 이는 미리 covering을



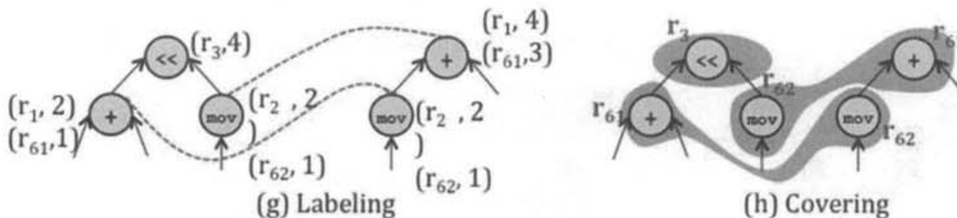
(그림 5) MWISP를 통한 최종 후보 선택 예제

시도해서 우리가 선택한 다중출력명령어의 조합이 실제로 가능한지 판단하는 과정도 추가되어있다.

3. 코드 생성 알고리즘의 한계

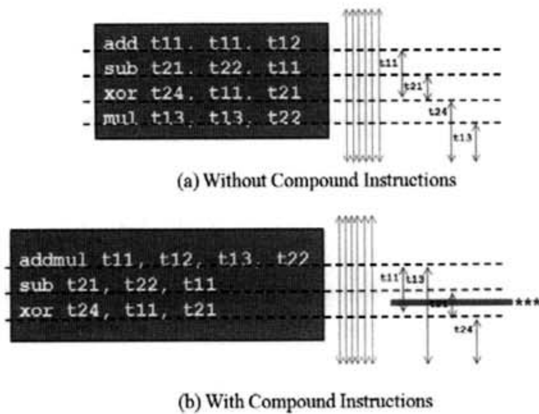
2장에서 제안한 알고리즘을 사용하면, 효율적으로 다중 출력 명령어를 컴파일러에서 생성할 수 있다. 하지만, 기존 알고리즘은 다중 출력 명령어를 선택할 때 Register Pressure를 전혀 고려하지 않는다. 이럴 경우 선택된 다중 출력 명령어로 인해서 성능이 개선되는 것이 아니라, 레지스터 대피(register spill)이 발생해서 성능을 더 나쁘게 할 수 있다[7].

(그림 7)은 다중 출력 명령어가 spill을 발생시키는 경우의 예제를 보여주고 있다. 일단, 사용 가능한 전체 레지스터 개수가 8개이고 다른 변수(variable) 6개가 현재의 기본 블록(basic block)에서 생존(live) 한다고 가정하면, (그림 7(a))에서 볼 수 있듯이, 다중 출력 명령어가 없을 경우에는 register pressure가 최대 8이 되기 때문에 8개의 레지스터로 충분하다. 하지만, 첫 번째 add 명령어와 네 번째 mul 명령어를 합쳐서 다중 출력 명령어가 선택되면, mul 명령어가 add명령어의 위치만큼 올라가게 되어 결국 t13의 생존범위(live range)가 길어지게 되고, *** 부분에서는 register pressure가 9가 되어 대피 정책에 의해서 생존 변수 중 하나를 대피해야 한다. 이 경우 대피를 위한 메모리에 저장하는 명령어(store instruction)와 재적재(reload)를 위해 메모리에서 레지스터로 값을 읽어 오는 명령어(load instruction)




```
addmov R3, R2, R8, R7  // R3+=R2; R8 = R7;
addmov R8, R9, R5, R4  // R8+=R9; R5 = R4;
sll R6, R3, R5         // R6 = R3 << R5;
```

(그림 6) 최종 코드 생성 예제



(a) Without Compound Instructions

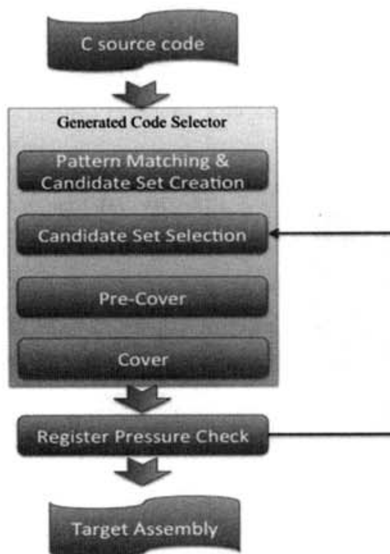
(b) With Compound Instructions

(그림 7) Register Pressure를 증가 시키는 다중 출력 명령어의 예제

가 전체 코드에 추가 되기 때문에 다중 출력 명령어를 사용하지 않았을 때 보다 코드의 성능이 더 나빠지게 된다. 이전 논문들에서는 이 문제를 전혀 고려하지 않고 단지 다중 출력 명령어와 단일 출력 명령어 들 중 cost가 낮은 것을 선택하는 방식이었다. 그래서 이번 논문에서는 코드 질(code quality)에 큰 영향을 줄 수 있는 대피 관련 다중 출력 명령어를 찾아, 그런 종류의 다중 출력 명령어는 생성되지 않도록 하여 성능 향상을 이룰 수 있게 한다.

4. 레지스터 대피를 방지하는 다중 출력 명령어 선택 알고리즘

3장에서 본 것처럼, 기존 다중 출력 명령어 선택 알고리즘에서는 register pressure 전혀 고려하지 못한다. 그래서 실제로 대피/재적재 코드를 양산할 수 있는 후보를 선택하게 된다. 우리는 이러한 문제점을 개선하기 위해서 (그림 8)과 같이 기존 알고리즘을 개선하였다.

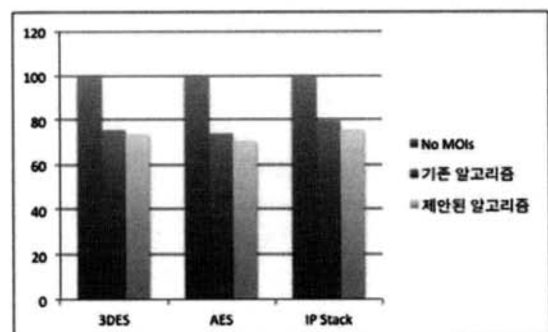


(그림 8) 개선된 다중 출력 명령어 선택 알고리즘

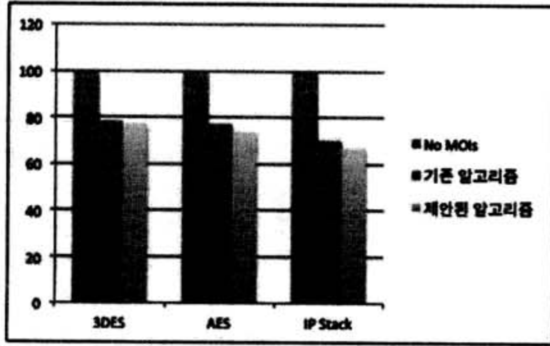
개선된 알고리즘에서는 최종적으로 선택된 후보들의 register pressure를 측정하고, 만약 그 register pressure가 타겟 프로세서가 가진 전체 register 개수를 초과하는 후보들을 찾고, 만약 이러한 후보들이 존재한다면, Candidate Set Selection 단계로 다시 돌아가게 되고, 간섭 그래프를 다시 그리게 되는데, 이때 register pressure가 높은 후보들을 제외하게 된다. 이렇게 하면, 기존에 위의 후보들로 인해서 선택되지 않은 후보들이 다시 선택될 수 있는 기회가 부여되기 때문에 최적의 코드를 생성할 수 있게 된다. 이러한 반복은 register pressure가 전체 레지스터 개수를 하는 최종 후보들이 없을 때까지 계속된다.

5. 실험 결과

이장에서는 개선된 알고리즘과 기존 알고리즘을 통해 실제 코드를 생성 비교해서 성능을 비교한다. 우리는 먼저 다중 출력 명령어 선택 알고리즘을 Little C Compiler(LCC)[2]에 구현하여 비교 하였다. 또한 실험을 위해 MIPS32의 ISA에 10가지 다중 출력 명령어를 추가한 명령어 레벨 시뮬레이터를 사용하였다. 실험 코드는 암호화 알고리즘인 Data Encryption Standard(3DES)[3], Advanced Encryption Standard(AES)와 internet protocol stack인 IP Stack을 사용하여, 코드 사이즈와 dynamic instruction count를 비교하였다. (그림 9)와 (그림 10)에서 볼 수 있듯이, 평균적으로 코드사이즈는 기존 알고리즘보다 3% 감소하였고, 실행 속도는 2.8% 빨라졌음을 알수 있다. 실제 결과가 예상보다 저조한데 이것은 우리가 사용하는 LCC 컴파일러의 레지스터 할당기의 특성 때문이다. LCC 컴파일러는 모든 컴파일 모듈의 실행 단위를 기본 블록으로 한정하고 있고, 작은 컴파일러를 지향하기 때문에 대부분의 최적화 모듈들도 제외 되어 있다. 또한 레지스터 할당 알고리즘 자체도 거의 대피가 생기지 않도록 작성되어 있다. 그래서 실제 다중 출력 명령어로 인한 대피가 많지 않기 때문에 기존 알고리즘과 비교했을 때 크게 성능 향상을 보이지는 않지만, 실제로 여러 분야에서 사용되는 최적화 컴파일러에서는 대피의 영향이 더 커지게 되고, 반드시 다중 출력 명령어를 선택할 때 register pressure를 고려해야 한다.



(그림 9) Normalized Code Size



(그림 10) Normalized Dynamic Instruction Count

6. 결 론

응용 특화 프로세서(Application Specific Instruction-set Processor)가 여러 분야에서 최적의 성능을 내기 위해서 다양하게 개발되고 있는데, 이러한 프로세서에서는 각각의 응용 어플리케이션의 특성에 맞는 다중 출력 명령어를 추가하게 된다. 하지만 지금까지의 일반적은 컴파일러 코드생성 알고리즘은 이러한 다중 출력 명령어를 처리하지 못했다. 그래서 새로운 알고리즘을 개발하여 다중 출력 명령어를 컴파일러에서 생성하도록 하는 방법이 제안되었는데, 이 알고리즘에서는 register pressure를 고려하지 않아 다중 출력 명령어가 생성됨에 따라 실제 결과 코드에 레지스터 대피/대적재 코드가 추가되는 현상이 발생하였고, 이로 인해 성능이 떨어지는 상황이 발생하였다. 그래서 본 논문에서는 register pressure를 측정해서 다중 출력 명령어를 선택할 수 있는 새로운 알고리즘을 제안하였고, 최종적으로 실험을 통해서 평균 3%의 코드 사이즈 감소, 2.7%의 speed-up이 됨을 확인 할 수 있었다. 또한 본 알고리즘을 LCC 컴파일러가 아닌, 좀 더 최적환 된 컴파일러에 적용할 경우 이 이상의 성능 향상을 이룰 수 있을 것으로 기대된다.

참 고 문 헌

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code Generation Using Tree Pattern Matching and Dynamic Programing. ACM Transactions on Programming Languages and Systems., 11(4):491 - 516, Oct., 1989.

[2] C. Fraser and D. Hanson. A Retargetable C Compiler : Design and Implementation. Benjamin/Cummings Publishing Co., 1994.

[3] C. Devine. http://xyssl.org, 2007.

[4] Hanno Scharwaechter , Jonghee M. Youn , Rainer Leupers , Yunheung Paek , Gerd Ascheid , Heinrich Meyr, A code-generator generator for multi-output instructions, Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, September 30-October 03, 2007, Salzburg, Austria.

[5] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering Efficient Code Generators Using Tree Matching and Dynamic Programming. Technical Report TR-386-92, 1992.

[6] S. W. K. Tjiang. An Olive Twig. Technical report, Synopsys Inc., 1993.

[7] G. J. Chaitin. Register allocation and spilling via graph coloring. Symposium on Compiler Construction, 17(6):98 - 105, 1982.

[8] Jonghee M. Youn, Jongwon Lee, Yunheung Paek, Jongeun Lee, Hanno Scharwaechter, Rainer Leupers, Fast graph-based instruction selection for multi-output instructions, Software: Practice and Experience Vol.41, Issue6, pp.717-736, May, 2011.

[9] S. Sakai, M.Togasaki, and K. Yamazaki. A Note on Greedy Algorithms for the Maximum Weighted Independent Set Problem. Discrete Applied Mathematics, 126:313 - 322, 2003.

윤 종 희

e-mail : jhyoun@gwnu.ac.kr
2003년 경북대학교 전자전기공학부(학사)
2011년 서울대학교 전기컴퓨터공학부(박사)
2011년~현 재 강릉원주대학교 컴퓨터공학과 강의전담교수
관심분야 : 임베디드 시스템, 컴파일러 최적화, 소프트웨어 최적화, 아키텍처, MPSoC, GPGPU

백 윤 흥

e-mail : ypaek@snu.ac.kr
1988년 서울대학교 컴퓨터공학과(학사)
1990년 서울대학교 컴퓨터공학과(석사)
1997년 UIUC 전산과학(박사)
1997년~1999년 NJIT 조교수
1999년~2003년 KAIST 전자전산학교 부교수
2003년~현 재 서울대학교 전기정보공학부 교수
관심분야 : 임베디드 소프트웨어, 컴파일러, MPSoC, CGRA

고 광 만

e-mail : kkman@sangji.ac.kr
1993년 동국대학교 컴퓨터공학과(공학석사)
1998년 동국대학교 컴퓨터공학과(공학박사)
1998년~2001년 광주여자대학교 컴퓨터과학과 교수
2003년 QUT(호주) 방문연구
2008년 UQAM(캐나다) 방문연구
2010년 UC Irvine(미국) 방문연구
2001년 9월~현 재 상지대학교 컴퓨터정보공학부 교수
관심분야 : 컴파일러, 프로그래밍 실행 환경