Multi-Output Instruction 기술 방법 향상을 통한 성능 개선에 관한 연구

윤종희, 안민욱, 김대호, 김호균, 조두산, 권용인, 백윤흥
서울대학교 공과대학 전기컴퓨터 공학부
e-mail : jhyoun, mwahn, dhkim, hkkim, dscho, yikwon, ypaek@compiler.snu.ac.kr

A Study on New Rule Description for Multi-Output Instructions

Jonghee Youn, Minwook Ahn, Daeho Kim, Hokyun Kim, Doosan Cho, Yongin Kwon, Yunheung Paek
School of Electrical Engineering, Seoul National University

요 약

많은 DSP 등에서 Multi-Output Instructions(MOI)를 지원하나 이를 사용할 수 있는 컴파일러가 없다. 그래서 기존연구에서 이 문제를 해결하는 새로운 코드 생성 알고리즘을 개발하여 소개하였다. 하지만, 이 논문에서 제시한 방법은 많은 제약이 있어, 본 논문에서는 기존 논문에서 사용한 MOI 를 위한 compiler grammar rule description 을 확장하고, 알고리즘을 변경하여 기존에 제안된 방법이 해결할 수 없었던 MOI 들까지 모두 컴파일러에서 처리할 수 있도록 하였다.

1. 서론

오늘날 임베디드 프로세서는 파워, 프로세서의 크기, 수행시간 등의 여러 가지 제약 조건에 따라 설계되고 개발 된다. 특히 ASIP(Appication Specific Instruction Set Processor) 이런 조건들을 더 많이 만족해야 한다. 그래서 다양한 형태의 하드웨어 구조들이 개발 되고 있다. 그 중 하나가 Multi-Output Instruction(MOI)이다. MOI 란 같이 하나의 명령어가 두 개 이상의 결과를 만들어 주는 것을 의미한다. 그림 1에서 간단한 MOI 를 보여준다. 이는 ADD 두 개를 동시에 할 수 있는 명령어이다.

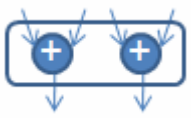

그림 1 Dual ADD

실제로 소니 pDSP 등에서 PLDXY r1, @a, r2 @b 과 같은 레지스터 r1 과 r2 에 메모리 a 와 b 에서 읽은 값을 동시에 저장하는 명령어 들이 존재 한다.

하지만 문제는 이러한 MOI 들을 지원하는 컴파일러의 코드 생성기가 거의 없는 실정이다.

그래서 논문[1]에서 기존에 일반적인 코드 생성기로 많이 사용되어온 OLIVE[2], IBURG[3], BURG[4]등의 code-generator generator 를 바탕으로 하여 새로운 code selection algorithm 을 넣어 이 문제를 해결 하였다. 하지만, 아직 이 알고리즘에서 해결 하지 못하는 문제가 있다. 바로 MOI 에 입력으로 들어가는 operand 들 중에 공유되는 것이 있을 때 이를 처리할 방법이 없다. 이것은 매우 중요한 문제로 실제 프로세서에서는 명령어의 인코딩 문제 등으로 인해 MOI 를 만들 때 source operand 들 중 일부를 공유하게 만드는 경우가 많다. 그리고 Target application 의 소스코드에서도 이러한 형태를 자주 볼 수 있다. 그래서 이번 논문에서는 이를 간단하게 처리할 수 있는 방법에 대해서 논의 해보도록 하겠다. 다음 장에서는 기존 알고리즘을 설명하고, 3 장에서는 새롭게 제시된 compiler grammar rule description 과 이를 지원하기 위한 변형된 알고리즘을 소개하고, 4 장에 결론을 맺게 된다.

2. MOI 를 위한 code selection algorithm

일반적인 compiler 의 code selection 알고리즘은 label 과 cover 두 단계로 이루어진다. 이는 아래 예제와 같은 tree grammar 를 바탕으로 한 것으로, label 에서는 C source 코드를 Tree 형태로 만든 DFT(Data Flow Tree)에서 각 노드에 적용될 수 있는 명령어들을 찾고, cover 단계에서는 이 찾아 놓은 여러 명령어 중에서 최적의 성능을 낼 수 있는 명령어들을 각각의 노드마다 선택하는 방법이다.

NT -> opcode (NT, NT)

여기서 NT 는 Non-terminal 들을 의미하고 opcode 는 ADD, MUL, SUB 등의 명령어의 operation 을 의미한다.

논문[1]에서는 MOI 명령어를 컴파일러에서 처리하기 위해서 아래와 같이 MOI 를 위한 용어를 정의했다.

- **Rule**: represents an instruction pattern in the tree grammar

- **Simple rule** : represents a typical tree pattern
- **Complex rule**: consists of several simple rules
- **Split rule**: is a simple rule that is a member of a complex rule

이를 바탕으로 MOI 를 위한 grammar rule 을 아래와 같이 정의 하였다.

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

즉, 기존의 tree grammar 에서는 하나의 명령어당 하나의 opcode 와 하나의 NT 만을 표현할 수 있었지만, 확장된 grammar 에서는 원하는 만큼 다양하게 표현 할 수가 있다.

논문[1]에서는 **그림 2**와 같이, 위의 새로운 grammar 표현 방식을 바탕으로 5 단계로 code selection algorithm 을 구성해서 MOI 를 컴파일러에서 처리 할 수 있도록 하였다.

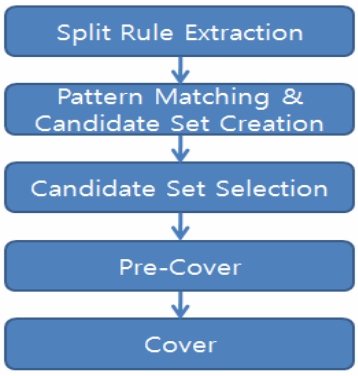

그림 2 MOI 를 위한 Code Seletion 알고리즘

간단히 설명하면, 첫 단계인 Split rule extraction 에서는 새로운 표현방식으로 표현된 Complex rule 을 split rule 단위로 나누고, 두 번째 단계에서 이 split rule 을 바탕으로 소스코드로부터 만들어진 DFT 를 labeling 하게 된다. 그리고 나서 split rule 이 label 된 DFT 의 각 노드간의 dependency 등을 체크하여 complex rule 이 될 수 있는 candidate set 을 만들게 되고, 세 번째 단계에서 만들어진 candidate set 중에서 최적의 complex rule 들을 Maximum –independent weight set problem 을 이용해서 선택하게 된다. 4 번째 단계인 Pre-Cover 는 앞에서 선택된 최적의 complex rule 이 실제로 cover 될 수 있는지 없는지를 미리 확인하여 다음 단계인 cover 에서 문제가 생기지 않도록 해주고, 마지막 단계인 Cover 에서 실제로 complex rule 에 따라 MOI 들을 선택하게 해준다.

3. 확장된 Complex rule Description

논문[1]에서 제안한 MOI 를 위한 complex rule 기술 방법과 알고리즘은 상당히 실용적이나, 현실적이지 못한 부분을 가지고 있다. 실제로 target application 이나 프로세서에서는 여러 개의 독립된 operation 을 동시에 수행하기는 하나, operation 간에 source operand 를 공유하는 경우가 많기 때문이다. 즉 그림 3과 같이 두 개의 ADD 가 하나의 operand 를 공유 하는 경우가 상당이 많이 존재한다. 코드로 예를 들면 *R1 = R2 + R3; R4 = R3+R5;* 와 같은 경우가 될 것이다.

그림 3 Operand 를 공유한 Dual ADD

이러한 경우를 표현하기 위해 본 논문에서 새롭게 확장한 grammar rule 은 아래와 같다.

NT NT ... -> opcode(NT, NT[n]) opcode(NT[n], NT) ...

즉 각 operand 를 기술 할 때 [num] 형태를 사용하여 서로 같은 operand 라는 것을 표현 해 주는 방식이다. 즉 아래와 같이 그림 3은 표현 될 수 있다.

NT NT -> ADD(NT, NT[0]) ADD(NT[0], NT)

또한 새로운 complex rule 기술에 따라 candidate set creation 도 변경되어야 한다. 즉, 기존에는 complex rule 에 따라 split rule 의 조합을 만들고 이들간이 dependency 를 확인하는 방법으로 complex rule 에 부합하는지를 결정하고 candidate set 을 만들었지만, 새롭게 제안된 방법에서는 기존 방법에 더해, operand 의 공유가 complex rule 에 있는지를 확인하고, 만약 있다면 각 split rule 들의 operand 들을 체크하여 complex rule 에서 정의한 operand 공유에 따라 서로가 하나의 operand 로 사용 될 수 있는지를 확인하고 candidate set 을 만들어야 한다.

4. 결론

많은 프로세서들이 여러 가지 제약조건들을 만족시키기 위해서 MOI 와 같은 특별한 명령어들을 지원한다. 하지만 이를 제대로 활용할 수 있는 컴파일러 기술이 많이 존재하지 않았다. 하지만 논문[1]에서 새로운 방법을 제시해 MOI 의 활용을 극대화 하려고 했지만, MOI 를 표현하는 방법의 문제로 인해 그 활용도가 떨어졌다. 이번 논문을 통해 거의 모든 MOI 를 표현할 수 있도록 compiler grammar rule description 방법을 확장하고, 그에 따른 알고리즘 변경을 제시하였다. 이를 바탕으로 한 새로운 컴파일러 코드 생성기는 프로세서가 지원하는 다양한 MOI 를 지원할 수 있게 되었다.

참고문헌

[1] Hanno Scharaeschter, Jonghee Youn et al., “A code generator Generator for Multi-Output Instructions”, The International Conference on Hardware-Software Codesign and System Synthesis 2007(CODES+ISSS2007)

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

[3] C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG–fast optimal instructionselection and tree parsing. Technical Report CS-TR-1991-1066, 1991.

[4] 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.