다수의 레지스터를 확보하기 위한 ARM Thumb 레지스터 뱅크의 제안

이제형° 박진표 문수묵
서울대학교 삼성전자 서울대학교
lordljh@altair.snu.ac.kr° jinpyo11@hotmail.com smoon@altair.snu.ac.kr

Banked Register File for ARM Thumb to Secure More Registers

Je-Hyung Lee° Jinpyo Park, Soo-Mook Moon
Seoul National University, Samsung Electronics, Seoul National University

요 약

ARM 프로세서는 내장형 시스템에서 가장 널리 사용되는 32비트 마이크로 프로세서 중 하나이며, Thumb 명령어 세트는 보다 작은 코드 크기를 위해 제공하는 16비트 확장 명령어 세트이다. Thumb의 약점중의 하나는 줄어든 명령어 길이 때문에 이용할 수 있는 레지스터의 개수가 반으로 줄어든다는 것인데 결과적으로 가용 레지스터의 부족으로 인해 spill 코드가 빈번하게 발생할 수 있다.

우리는 약간의 하드웨어 및 명령어 수정을 통해 뱅크(bank)로 이루어진 레지스터 파일을 제공하고자 한다. 이로 인해 컴파일러는 보다 여유 있는 레지스터를 확보하게 되어 spill 코드가 줄어들게 되므로 보다 작은 크기의 코드를 얻어낼 수 있다. 이 변화된 형태의 레지스터 파일을 운용하기 위한 효율적인 레지스터 할당기법이 요구되며, 제안하는 영역기반 레지스터 할당기법을 통해 이미 최적화된 Thumb 코드 대비 약 5.1%의 코드 크기 감소효과를 볼 수 있었다.

1. 서론

ARM 프로세서는 현재 내장형 시스템에서 가장 널리 사용되는 32비트 마이크로프로세서이다.[1] ARM 프로세서는 내장형 시스템에서의 수요를 위해 성능을 다소 희생하더라도 보다 고밀도의 코드를 생성하기 위해 16비트의 명령어 길이를 가지는 특수 명령어 세트를 제공하는데 이것이 Thumb 확장 명령어 세트이다.[2] Thumb 명령어는 완벽하게 ARM 표준 명령어 세트의 부분집합이며, 제공하는 명령어의 다양성은 떨어지므로 똑같은 역할을 수행하기 위해서는 보다 많은 개수의 명령어가 필요함에도 불구하고 반으로 줄어든 명령어 크기 때문에 평균적으로 약 30%의 코드가 감소한 효과를 갖는 것으로 알려져 있다.[2] 다만 Thumb 명령어는 수행과정에서 ARM 표준명령어로 변환되는 과정을 거치기 때문에 추가적인 해석기가 동원되며, 또한 변환과정 및 늘어난 명령어 개수에 의한 성능의 감소를 짐작할 수 있다. 그렇지만 메모리의 크기가 극히 제약되는 내장형 시스템에는 매우 효과적인 코어로 각광받아 왔다.

줄어든 16비트의 명령어 길이로 인해 원래는 4비트씩 할당되던 레지스터 필드도 모두 3비트로 할당된다. 따라서 ARM이 가지고 있는 16개의 레지스터도 Thumb 명령에서는 8개만을 이용할 수 있다. 결국 이용할 수 있는 레지스터 개수의 감소로 인해 동시에 많은 변수가 이용되는 코드영역에서는 변수를 모두 레지스터에 저장하지 못하고 메모리에 담아 두었다가 필요시에 레지스터로 다시 되가져오는 spill 코드가 많이 필요하게 된다. spill 코드는 전체 코드 크기를 늘릴 뿐만 아니라 수행 속도에도 좋지 않은 영향을 미친다.

우리는 Thumb에서 빈번하게 발생할 수 있는 이 spill코드를 줄이기 위해 이용 가능한 레지스터의 개수를 늘리고자 한다. Thumb에서 이용할 수 없는 나머지 8개의 레지스터에 보다 쉽게 접근할 수 있는 최소한의 하드웨어적인 변경을 가해 결과적으로는 8개씩 2개의 레지스터 묶음(뱅크)을 제공하고, 또한 이러한 레지스터 구조를 효율적으로 사용할 수 있는 레지스터 할당기법을 제안함으로써 프로그램의 코드 크기를 줄이고자 한다.

이 논문은 다음과 같이 구성되어 있다. 2절에서는 Thumb이 가지고 있는 레지스터 구조의 제약 및 그에 따른 연구동기를 소개하고, 3절에서는 제안하는 하드웨어의 변경사항에 대해 기술한다. 4절에서는 제안한 레지스터 뱅크를 효과적으로 이용하기 위한 레지스터 할당기법을 논하고, 5절에서는 실험결과, 그리고 6절에서 결론으로 끝을 맺는다.

2. Thumb의 특징 및 연구동기

Thumb의 핵심은 32비트 프로세서에서 작동하는 16비트 길이의 명령어이다. 따라서 줄어든 명령어 표현 비트 수 때문에 대부분의 명령어가 가지는 operand가 2개로 제한되고 (예: ADD R1, R2), 그나마 레지스터 필드도 3비트로 제한된다. 따라서 Thumb에서 자유롭게 이용할 수 있는 레지스터는 8개로 줄게 되었다.

Thumb에서 R0-R7의 8개 레지스터는 일반적인 용도의 범용 레지스터로 사용되고, R13, R14, R15는 각각 Stack Pointer, Link Register, Program Counter로 이용되는 특수 레지스터이

며, R8-R12는 16개의 레지스터를 모두 이용할 수 있는 Format 5 범주에 속하는 명령어들에 의해서만 제한적으로 이용될 수 있는데 메모리보다 빠른 임시 저장소로 이용된다.[2]

<표2>에서 Mediabench, Mibench의 테스트벡터들에 대해 GCC 컴파일러에서 코드의 크기를 최소화 하기 위한 '-Os' 옵션을 주었을 때 어셈블리 명령어의 개수와 spill 코드의 개수를 확인할 수 있는데, 약 13%에 달하는 적지 않은 양의 spill이 발생함을 알 수 있다. R13-R15는 특수 레지스터라 차치하더라도 R8-R12를 자유롭게 쓸 수 없는 상황이 주된 이유라고 할 수 있다.

3. 제안하는 레지스터 뱅크

예제 코드 / 변경된 구조에서의 코드

	Rx	Ry	Rz	PC		Rx	Ry	Rz	Rw
def a					def a				
spill a	a				def b	a			
def b			a		bank	a	b		
spill b		b	a		def c	a	b		
def c			a		def d	a	b	c	
def d	c		a		...	a	b	c	d
...	c	d	a		use d	a	b	c	d
use d	c	d	a		use c	a	b	c	
use c	c		a		bank	a	b		
load b			a		use b	a	b		
use b		b	a		use a	a			
load a			a						
use a	a								

<그림1> 레지스터 뱅크를 통해 spill을 줄이는 예

<그림1>은 4개의 변수 a,b,c,d가 차례로 정의되고, 반대의 순서로 값이 이용되는 경우 각 변수들이 존재하는 위치를 보여준다. 이 그림을 통해 기존 Thumb에서 발생하던 spill의 형태, 그리고 제안하는 레지스터 뱅크가 어떻게 spill을 줄일 수 있는지를 알 수 있다. 여기서는 문제를 간략하게 표현하기 위해 레지스터는 4개만 존재하고, Rx, Ry만이 자유롭게 접근이 가능하며, Rz는 임시 저장소로, 그리고 특수레지스터 PC가 있다고 가정한다.

그림의 왼쪽에서는 범용 레지스터가 2개 뿐이기 때문에 a와 b는 spill 되는데, a는 임시 레지스터 Rz에, b는 메모리 영역에 spill 된다. 결국 a와 b 값을 spill 하기 위해 모두 4개의 추가적인 명령어가 동원되었다.

오른쪽 그림은 Rx, Ry뿐만 아니라 대칭적으로 Rz, Rw도 범용 레지스터로 제공되고 2개의 레지스터는 하나의 뱅크를 구성하며 특수 레지스터 PC는 다른 곳으로 옮긴 경우이다. 이러한 레지스터 구조에서도 일시에 접근 가능한 레지스터는 2개뿐이기 때문에 다른 뱅크에 있는 레지스터에 접근하기 위해서는 뱅크 전환 명령('bank'로 표시)이 사용된다. 결과적으로 이러한 뱅크구조의 레지스터로 인해 2개의 명령어가 절약됨을 알 수 있다.

이와 같이 대칭적인 범용 레지스터 뱅크를 제공하기 위해 다음과 같은 하드웨어적인 변경이 필요하다.

1) 범용 레지스터의 추가 : R13-R15는 더이상 특수 레지스터가 아닌 범용 레지스터가 되어 R0-R7, 그리고 R8-R15는 똑같은 모양을 한 두개의 레지스터 뱅크를 이루게 된다. SP, LR, PC는 따로 특수 레지스터 뱅크를 형성한다.

2) BSB(Bank Selection Bit) : 현재 접근할 수 있는 뱅크가 어떤 것인지를 가리키는 flag이며, 이것이 레지스터의 주소와 조합되어 실제 레지스터를 선택하게 한다. BSB는 특수 레지스터 CPSR(Current Program Status Register)의 한 비트를 차지하게 된다. 프로그램 동작시 어떤 레지스터가 실제로 선택되는지에 대한 방침은 <표1>에 나타나 있다.

<표1> 레지스터 주소와 실제로 선택되는 레지스터와의 관계

Register Address	Effective Register (not at Format 5 Inst.)		Effective Register (at Format 5 Inst.)	
	BSB:0	BSB:1	BSB:0	BSB:1
0	R0	R8	R0	R8
1	R1	R9	R1	R9
2	R2	R10	R2	R10
3	R3	R11	R3	R11
4	R4	R12	R4	R12
5	R5	R13	R5	R13
6	R6	R14	R6	R14
7	R7	R15	R7	R15
8			R8	R0
9			R9	R1
10			R10	R2
11			R11	R3
12			R12	R4
13	SP	SP	R13	R5
14	LR	LR	R14	R6
15	PC	PC	R15	R7

3) 뱅크 전환 명령의 추가 : 뱅크 전환을 위해 새로운 명령어 MVB가 추가된다. 이것은 16개의 레지스터를 모두 선택할 수 있는 Format 5 명령어로 등록되는데 bit[9,8]이 11인 영역을 차지하게 되며, MOV 명령과 똑같은 역할을 하되 BSB만 변경하는 역할을 한다. 원래 bit[9,8]이 11로 등록되어 있었던 BX 명령은 bit[9,8]이 00,01로 변경되어 등록된다.

4) 특수 레지스터를 operand로 가지는 일부 Format 5 명령의 변경: 원래 Thumb에서는 R13, R14, R15가 각각 SP, LR, PC 였기 때문에 Format 5 명령에서도 이 레지스터들을 자유로이 사용할 수 있었다. 하지만 제안하는 레지스터 구조에서는 R13-R15가 범용 레지스터로 등록되기 때문에 Format 5 명령에서 특수 레지스터를 이용할 수 없게 되었다. 따라서 컴파일러에서 의미있다고 생각되어 지는 특수레지스터 간의 연산 조합을 일부 선택해서 bit[9,6]이 1000인 미등록 영역에 등록한다.

5) PUSH/POP 행동의 변화 : Thumb에서는 한꺼번에 다수의 레지스터를 PUSH/POP할 수 있으며 8개의 레지스터 중 어떤 조합이라도 선택할 수 있도록 8비트를 할당하고 있다. 하지만 변경된 레지스터 구조에서는 16개의 레지스터를 모두 PUSH/POP할 수 있도록 하나의 비트가 2개의 레지스터를, 즉 n번째 비트는 2n-2와 2n-1번 레지스터를 동시에 지정한다.

4. 영역기반 레지스터 할당

레지스터 뱅크로 인해 비록 전체 가용 레지스터의 개수는 두 배로 늘어나지만 한번에 접근할 수 있는 레지스터는 여전히 8개이기 때문에 필요한 레지스터 요구량을 서로 다른 뱅크영역으로 적절히 분배하지 못하면 아무런 효과도 거둘 수 없다. 우리는 뱅크형태의 레지스터를 효과적으로 이용하기 위한 레지스터 할당 기법으로 영역기반 레지스터 할당을 제안한다. 이 기법의 장점은 일단 뱅크 전환 명령에 의해 코드가 서로 다른 뱅크 영역으로 나뉘어 지고 나면 일반적인 레지스터 할당 기법을 별다른 수정 없이 그대로 적용할 수 있다는데 있다. 따라서 중요한 것은 영역 선택을 신중히 함으로써 레지스터 요구량을 적절히 나누는데 있다.

우리가 추구하는 가장 큰 목적은 spill코드를 제거하는 것이다. <그림1>에서 살펴보았듯이 spill이 발생한 레지스터 요구량이 큰 영역 중에서도 한시적으로 이용되는 변수들이 집중적으로 몰려있는 영역이 새로운 뱅크로 지정되기에 적당하다. 즉, 변수 c,d의 경우와 같이 하나의 뱅크가 가지는 레지스터 개수를 넘지 않는 범위 내에서 비교적 짧은 유효범위를 가지는 변수들을 파악하여 그 앞뒤로 뱅크전환을 시도하면 spill을 줄일 수 있다.

실제로는 뱅크 전환을 통해 삭제되는 spill명령어의 개수, 추가되는 뱅크 전환 명령 및 뱅크 간의 데이터 이동명령(다른 쪽 뱅크에 있는 값이 필요한 경우 추가됨)을 입력으로 한 비용함수로 정의하여 가장 비용이 적은 쪽으로 뱅크 전환할 영역을 선택하고, 분리된 각각의 영역에 대해 그래프컬러링에 기반한 레지스터 할당[3][4]을 수행한다. 레지스터 할당기는 컴파일러가 생성한 Thumb 어셈블리 파일을 입력으로 변경된 레지스터 구조에서의 영역기반 레지스터 할당을 수행하여 어셈블리 파일로 출력하는 후위 최적화기로 작동하는데, 이러한 구조는 컴파일러의 다른 최적화 단계와의 간섭을 배제하고 레지스터 할당기의 효과를 정확하게 파악할 수 있는 장점이 있다. 구현된 영역기반 레지스터 할당 알고리즘은 <그림2>와 같다.

```
Do CFA & DFA
Rename pseudo live ranges
Compute span of each spilled live range
Find high register pressure blocks(HRPBs) in each span
for each combination of spilled live ranges do
    Find enclosing blocks having HRPBs
    for each enclosing blocks do
        Find secondary bank region with minimal cost
Do global register allocation
```

<그림2> 영역기반 레지스터 할당 알고리즘

5. 실험결과

이 실험은 SPARC 워크스테이션 상에서 GNU toolchain을 수정하여 구현되었으며, 컴파일러는 GCC 3.0 버전을 이용하였다. 영역기반 레지스터 할당기는 컴파일러의 어셈블리 출력을 이용하는 후위 최적화기로 구현되었고, 완성된 실행파일은 수정된 시뮬레이터에서 동작한다.

<표2> 벤치마크의 코드와 spill의 크기 및 코드감소 정도

	Original Thumb instr. count (Spill count)	Banked Thumb instr. count (Spill count)	Instr. count reduction ratio
adpcm	430(66)	407(23)	5.3%
basicmath	972(185)	908(84)	6.6%
g721	2152(230)	2052(63)	4.6%
gsm	10459(888)	10019(176)	4.2%
geomean			5.1%

<표2>는 Mediabench, Mibench에서 추출한 4개의 테스트벡터를 이용한 결과를 보여주고 있다. 모든 수치는 명령어의 개수이다. Original Thumb instr. count는 GCC에서 '-Os' 옵션으로 컴파일 한 결과이며 Spill count는 그 중 spill 코드의 개수를 나타낸다. 여기서 spill코드는 직접 메모리에 load/store하는 것과 더불어 R8-R12를 임시 저장공간으로 이용하는 복사 명령도 포함된다. 평균적으로 줄어든 코드의 크기는 약 5.1%로 나타났다.

현재의 레지스터 할당기에 필요한 보완점 중 가장 중요한 것은 영역을 선택하는 비용함수가 레지스터 할당이 끝난 이후에 발생하는 spill의 크기를 정확하게 예측할 수 없다는 것이다. 따라서 줄어드는 spill 을 예측하기 위한 보다 풍부한 정보를 제공한다든가, 새로운 heuristic이 필요할 것이다. 뱅크 전환이 빈번하게 발생하는 경우 추가되는 뱅크전환 명령의 부담 또한 간과할 수 없다. 마지막으로 데이터의 유효범위가 복잡하게 얽혀있는 경우 뱅크간의 데이터 이동명령이 큰 부담이 될 수 있다. 이 뱅크 전환 명령과 뱅크간 데이터 이동명령의 부담은 비용함수 및 영역 선택과 밀접한 관계가 있으므로 같은 맥락에서의 보완이 필요하다.

6. 결론

Thumb에서 8개로 줄어든 가용 레지스터를 각각 8개의 레지스터를 가지는 두 개의 레지스터 뱅크 형태로 변환함으로써 빈번하게 발생하는 spill 명령을 줄이고자 하였다. 그러기 위해서는 레지스터 파일의 변화와 일부 명령어의 추가 및 행동의 변화가 필요하며, 뱅크 레지스터 구조를 효과적으로 이용하기 위해 영역기반의 레지스터 할당 방법을 제안하였다. 이러한 수정을 통해 이미 컴파일러로 크기 최적화를 행한 코드에 비해 약 5.1%의 추가적인 코드 감소 효과를 볼 수 있었다.

참고문헌

[1]http://japan.cnet.com/news/tech/story/0,2000047674,20069388,00.htm

[2]ARM Ltd. "An Instruction to Thumb", 2.0 edition, March 1995

[3] Jinpyo Park, Je-Hyung Lee, Soo-Mook Moon. "Register Allocation for Banked Register File", ACM Workshop on LCTES, June 2001

[4]P. Briggs. "Register Allocation via Graph Coloring", PhD thesis, Rice Univ., 1992