# 함수 수준에서 프로파일 정보를 이용한 ARM과 Thumb 명령어의 선택

## (Profile Guided Selection of ARM and Thumb Instructions at Function Level)

소 창 호 † 한 태 숙 ††
(Changho Soh) (Taisook Han)

**요 약** 임베디드 시스템에서는 메모리와 에너지의 소비가 중요한 관심사 중 하나이다. 메모리와 에너지의 소비를 줄이기 위해 32비트의 ARM 프로세서는 16비트 Thumb 명령어 세트를 지원한다. 주어진 응용프로그램에 대해 Thumb 코드는 일반적으로 ARM 코드보다 코드 사이즈가 작지만, 실행속도는 느리다. 코드 사이즈가 작으면서도 실행속도가 느리지 않은 코드를 생성하기 위한 방법으로 Krishnaswarmy는 응용프로그램에 대한 프로파일 정보를 이용하여 모듈 수준에서 ARM과 Thumb 명령어 세트를 선택하는 알고리즘을 고안했다. 이 알고리즘은 작은 성능 손실로도 상당한 코드 사이즈 감소 효과를 갖지만, 명령어 세트가 모듈 수준에서 선택되기 때문에 Thumb 코드로 컴파일 하면 코드 사이즈를 줄일 수 있는 함수들도 ARM 코드로 컴파일 되어, 추가적인 코드 사이즈 감소의 기회를 잃게 되는 문제점을 갖고 있다.

본 논문에서는 ARM과 Thumb 코드가 혼합된 코드 사이즈의 감소를 이끌어내기 위해 함수 수준에서 프로파일(profile) 정보를 이용한 명령어 세트 선택 알고리즘을 제안했다. 우리는 성능에서의 페널티는 없이 2.7%의 코드 사이즈를 추가로 줄일 수 있었다.

**키워드** : 프로파일링, 최적화, 코드 사이즈, 사이클 카운트, 임베디드 시스템

***Abstract*** In the embedded system domain, both memory requirement and energy consumption are great concerns. To save memory and energy, the 32 bit ARM processor supports the 16 bit Thumb instruction set. For a given program, the Thumb code is typically smaller than the ARM code. However, the limitations of the Thumb instruction set can often lead to generation of poorer quality code. To generate codes with smaller size but a little slower execution speed, Krishnaswarmy suggests a profiling guided selection algorithm at module level for generating mixed ARM and Thumb codes for application programs. The resulting codes of the algorithm give significant code size reductions with a little loss in performance. When the instruction set is selected at module level, some functions, which should be compiled in Thumb mode to reduce code size, are compiled to ARM code. It means we have additional code size reduction chance.

In this paper, we propose a profile guided selection algorithm at function level for generating mixed ARM and Thumb codes for application programs so that the resulting codes give additional code size reductions without loss in performance compared to the module level algorithm. We can reduce 2.7% code size additionally with no performance penalty

**Key words** : profiling, optimization, code size, cycle count, embedded system

## 1. 서 론

### 1.1 연구 배경

임베디드 시스템(embedded system)이라고 불리는 시스템들이 우리 주변에서 점점 많이 사용하고 있다. 임베디드 시스템은 정보를 처리하는 부분이 더 큰 시스템에 내장(embedded)된 형태의 시스템이다[1]. 이러한 임

· 본 연구는 대학 IT연구센터 육성 지원사업의 연구결과로 수행되었음
† 비 회 원 : 한국과학기술원 전산학과
sonar@pllab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
han@cs.kaist.ac.kr
논문접수 : 2004년 2월 26일
심사완료 : 2005년 1월 19일

베디드 시스템의 예로는 차량에서의 정보처리, 스마트 홈(smart home)에서의 전자제품, 공정 제어 등이 있다.

임베디드 시스템을 실제 구현하는데 있어서 가장 주요한 특징은 효율성이다. 휴대성의 요구, 배터리 기술의 늦은 발전, 대량생산에 있어서의 경제적인 이유 등으로 인해 이러한 효율성을 극대화할 필요성은 점점 커지고 있다. 이런 이유로 인해 기존의 프로세서들에 비해 임베디드 프로세서(embedded processor)들은 메모리와 에너지 사용에 있어서 효율성을 중요한 주제로 다루고 있다.

지금까지 RISC 프로세서들은 낮은 에너지 소비와 높은 수준의 성능으로 인해 임베디드 시스템으로 많이 사용되어 왔다[2]. 그러나 최근 들어 코드가 길어진다는 것이 RISC의 문제점으로 지적되고 있다. 더 나은 성능에 대한 요구 때문에 프로세서들의 처리 비트가 증가함에 따라 소프트웨어에 필요한 메모리의 크기는 점점 커져가고, 이는 결국 비용 문제로 귀착된다. 게다가 RISC의 낮은 코드 밀도는 시스템의 전력 소비 역시 크게 증가시키게 하므로, RISC가 앞으로도 계속 임베디드 시스템에서 사용되려면 낮은 코드 밀도의 개선은 시급한 일이다.

이러한 이유로 어떤 RISC 프로세서들에서는 메모리에 대한 제약을 피하기 위한 방법으로 코드를 압축하는 기법을 사용하기도 한다. 그 중에서도 ARM, MIPS는 기존의 명령어 세트에 추가로 감소된 비트 폭을 갖는 명령어 세트(Reduced Bit-width Instruction Set)를 제공하여 이를 통한 코드 압축을 지원하고 있다[3]. 임베디드 시스템 도메인에서 가장 많이 사용되는 프로세서 코어의 하나인 ARM 프로세서는 기존의 32 비트 ARM 명령어 세트 외에 16 비트 Thumb 명령어 세트를 추가로 제공하고 있다. 추가된 Thumb 명령어 세트를 이용하면 하나의 명령어가 16 비트인 코드로 구성되기 때문에 32 비트인 ARM 명령어 세트로 구성되어 있는 코드에 비해 응용 프로그램의 코드 사이즈가 작아지게 된다. 그러나 Thumb 명령어 세트를 이용할 경우에는 실행 성능이 저하된다는 단점이 있다.

두 개의 명령어 세트를 갖는 아키텍처(Dual Instruction Set Architecture)와 관련하여 초기에는 감소된 비트 폭을 가진 명령어 세트를 이용하여 코드 사이즈를 최대로 줄이되, 실행시간의 저하를 최소화하는 컴파일러의 개발에 대한 연구가 지배적이었다. 그러나 최근에는 점차 두 개의 명령어 세트가 혼합된 코드(mixed code)를 생성해 내는 방법에 대한 연구들이 진행되고 있다.

이런 연구들 중에 프로파일러(profiler)를 통해 얻어낸 정보를 바탕으로 응용 프로그램의 명령어 세트를 모듈 수준에서 결정하여 코드를 생성해 주는 방법[4]이 있다. 프로그램의 각 모듈들은 특성에 따라 가장 적합한 명령어 세트를 선택하고, 선택된 명령어 세트로 코드를 생성한다. 따라서 이 방법은 프로그램 전체를 하나의 감소된 비트 폭을 가진 명령어 세트로 생성해 내는 방법에 비해 성능의 저하는 훨씬 적으면서 코드의 사이즈는 비슷하게 줄일 수 있었다.

### 1.2 문제 제기와 해결책

그러나 앞에서 언급한 Krishnaswary[4]의 문제점은 명령어 세트를 모듈 수준에서 선택하기 때문에, 해당 모듈을 구성하고 있는 함수들 각각에게 유리한 명령어 세트로 코드를 생성하면 얻을 수 있는 성능 향상과 코드 사이즈를 줄일 기회를 놓치게 된다는 것이다.

이러한 문제점을 해결하기 위해서 모듈 수준이 아닌 함수 수준에서 명령어 세트를 선택하는 혼합 코드 생성 기법이 필요하다. 이를 위해 함수 수준에서 명령어 세트를 선택한 후 모듈 분할을 통해 코드를 생성하는 방법을 본 논문에서 제시하고 그 효과를 살펴보기로 한다.

### 1.3 논문 구성

본 논문의 구성은 다음과 같다. 2장에서 관련 연구들을 소개를 하고 3장에서는 새롭게 제안하는 함수 수준에서 명령어 세트를 선택하는 혼합 코드의 생성 기법을 소개하고, 예상되는 특성을 정리한다. 4장에서는 구현 및 실험 결과를 분석하고 마지막으로 5장에서 결론과 향후 연구 과제를 제시한다.

## 2. 관련 연구

두 개의 명령어 세트를 이용하는 코드 사이즈 최적화에 관한 연구 방향은 두 가지로 정리할 수 있다. 초기에는 전체 프로그램을 감소된 비트 폭을 가진 명령어 세트로 구성된 코드로 만들어주는 컴파일러의 개발에 대한 연구가 지배적이었으나, 점차 두 가지 명령어 세트, 즉 보통의 명령어 세트와 추가된 감소된 비트 폭을 갖는 명령어 세트가 혼합된 코드를 생성해 내는 방법에 대한 연구로 변화되어 왔다.

### 2.1 감소된 비트 폭을 가진 명령어 세트를 이용한 코드 사이즈 최적화

RISC 프로세서들 중에 코드 사이즈를 줄이기 위한 방법으로 명령어 세트를 추가로 지원하는 경우 프로세서는 두 개의 명령어 세트를 가지게 된다. 하나는 일반적인 32 비트 명령어 세트이고 다른 하나는 감소된 비트 폭(16 비트)을 가진 명령어 세트이다. 이런 구조를 가진 대표적인 프로세서로 ARM, MIPS, ST100, Trangent 프로세서 등이 있고, 이 명령어 세트를 이용하여 코드를 생성해 내는 연구로는 ARM과 MIPS에서의 실험을 들 수 있다.

ARM에서의 실험[2]은 Thumb 명령어 세트를 이용하

는 컴파일러를 만들어 코드를 생성해 냈는데, ARM 컴파일러에 의해 생성된 코드에 비해 코드 사이즈는 30% 줄어들었지만 실행 시간은 10%~15% 늘어났다. Thumb 코드는 이론상 ARM 코드에 비해 반으로 줄어야 하지만 표현상의 제약 때문에 실제로는 그에 못 미친다. 또한 ARM 명령어 세트였다면 하나의 명령어로 실행할 수 있는 것을 Thumb에서는 여러 개의 명령어로 실행해야 하기 때문에 Thumb 코드에서는 ARM 코드에 비해 실행 시간도 더 걸리게 된다.

MIPS 프로세서의 실험[3]에서는 MIPS 16 ISA(Instruction Set Architecture)를 이용하여 코드 사이즈를 38% 줄였다.

### 2.2 모듈 수준에서 명령어 세트의 선택

모듈 수준에서 명령어 세트를 선택하여 코드를 생성해 내려는 연구[4]는 ARM과 Thumb 명령어 세트의 성질을 잘 이용하여 두 가지 명령어 세트를 적절히 혼합한 코드를 생성하면, 프로그램 전체를 Thumb 명령어 세트로 생성한 코드보다 우수한 성능을 가질 것이라는 가정에서 시작했다. 명령어 세트의 성질을 잘 이용하기 위해 프로파일 정보를 이용하여 자주 사용되는 모듈은 ARM 코드로 생성하고 자주 사용되지 않는 모듈은 Thumb 코드로 생성한다. 이러한 방법을 명령어 세트의 선택 단위가 크다고 해서 조립질 혼합 코드 생성(Coarse grained mixed code generation)라고 부른다.

이 방법은 프로그램 전체를 Thumb 명령어 세트로 컴파일 한 것에 비해 훨씬 작은 실행시간 손실을 보이면서도 전체를 Thumb로 컴파일 했을 때와 비슷한 정도의 코드 사이즈 압축 성능을 보인다.

## 3. 함수 수준에서 명령어 세트의 선택

앞에 소개한 모듈 수준에서 명령어 세트를 선택하는 알고리즘을 개선하여 함수 수준에서 명령어 세트를 선택하는 알고리즘을 새롭게 제안한다.

### 3.1 기본 아이디어

핵심 아이디어는 다음과 같다. 기존에 제안된 방법 중 가장 성능이 좋은 모듈 수준에서 명령어 세트의 선택에 의한 코드 생성의 문제점은 명령어 세트의 선택이 모듈 수준에서 이루어지기 때문에 각각의 함수가 Thumb 명령어 세트를 선택하더라도 ARM 명령어 세트를 선택한 함수가 해당 모듈 안에 존재하면 그 모듈 전체가 ARM 코드로 생성된다는 것이다. 이로 인해 해당 함수가 Thumb 명령어 세트로 컴파일 되었다면 얻을 수 있었을 코드 사이즈를 줄일 수 있는 기회를 놓치게 된다.

따라서 모듈을 구성하고 있는 함수들에 유리한 명령어 세트에 따라 모듈을 분리하여 코드를 생성해 준다면 이러한 기회들을 놓치지 않고 우수한 코드 사이즈 최적화를 할 수 있을 것이다.

### 3.2 개요

위의 아이디어를 바탕으로 함수 수준에서 명령어 세트가 혼합된 코드를 생성해내는 것은 아래의 네 단계로 이루어진다.

1. 프로파일링 하여 빈번하게 실행되는 함수들을 찾는다.
2. 찾은 함수들에 대해 ARM 명령어 세트와 Thumb 명령어 세트 중 유리한 명령어 세트를 선택한다. 빈번하게 실행되지 않는 함수들에 대해서는 Thumb 명령어 세트를 선택한다.
3. ARM 명령어 세트를 선택한 함수를 포함하고 있는 하나의 모듈을 두 부분 - ARM 명령어 세트가 유리한 함수들의 집합인 모듈과 Thumb 명령어 세트가 유리한 함수들의 집합인 모듈 - 으로 분리한다.
4. 분할한 모듈 중에서 ARM 명령어 세트가 유리한 모듈들은 ARM 명령어 세트로 컴파일하고 나머지 모듈들은 Thumb 명령어 세트로 컴파일 해준다.

기존의 방법과 달라진 것은 3번과 4번으로 ARM이 유리한 함수와 Thumb이 유리한 함수가 섞여 있는 모듈을 ARM 모듈과 Thumb 모듈로 분리하여 컴파일 하는 단계이다.

그림 1은 위에서 설명한 단계를 예를 들어 나타내었다.

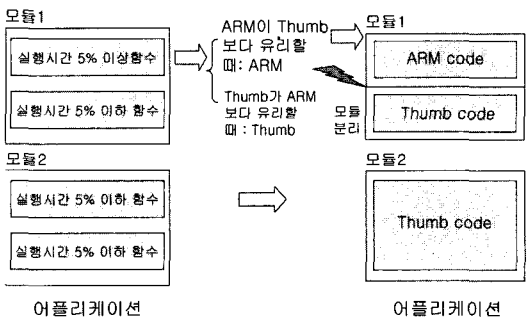


그림 1 함수 수준에서 명령어 세트의 선택

### 3.3 문제점 해결

앞서 제안한 알고리즘을 구현할 때 함수의 경계를 넘어서 이루어지는 최적화, 유리한 명령어 세트의 선택 방법, 모듈 분리와 같은 문제들을 해결해야 한다.

3.3.1 함수의 경계를 넘어서 이루어지는 최적화

[4]의 실험은 모듈 수준에서 명령어 세트를 선택하는 이유가 함수 수준에서 명령어 세트를 적용할 경우 함수의 경계를 넘어서 이루어지는 최적화가 줄어들기 때문이라고 한다. 함수 수준에서의 선택 방법이 효율적이기 위해서는 이러한 최적화가 없거나 크지 않아야 한다.

gcc에서 일반적으로 임베디드 소프트웨어 개발에 사용하는 -O2 레벨에서 적용되는 함수의 경계를 넘는 최적화는 없다. -O2레벨에서 임베디드 시스템을 개발하는 이유는 -O3레벨부터 loop unrolling이나 inlining이 적용되어 코드 사이즈가 늘어나기 때문이다. 또한 -O3 레벨에서도 매우 제한적인 함수의 경계를 넘는 최적화 기법만이 적용되고 있다. 이는 함수의 경계를 넘는 최적화 기법의 비용이 너무 크기 때문이다. 따라서 본 실험에서 함수의 경계를 넘어서 이루어지는 최적화의 영향은 전혀 없음을 알 수 있다.

3.3.2 유리한 명령어 세트의 선택 방법

함수 수준에서 유리한 명령어 세트의 선택 방법은 기존의 모듈 수준에서 제안된 방법 중 ARM과 Thumb 코드 중 사이클 카운트가 적은 코드를 선택하는 첫 번째 방법을 사용한다. 유리한 명령어 세트의 선택이 전체 성능에 미치는 영향이 크지 않으므로 함수 수준의 방법에 적합한 새로운 방법들을 고안하지 않고 앞서 제안된 방법 중에서 가장 간단하고 정확한 방법을 적용했다. 적용한 벤치마크 프로그램 전체 중 2개의 함수만이 Thumb 명령어가 유리한 함수이다.

3.3.3 모듈 분리

모듈 분리는 함수 수준에서 해당 함수에 유리한 명령어 세트를 결정한 후 ARM 명령어 세트가 유리한 함수는 ARM 명령어 세트로, Thumb 명령어 세트가 유리한 함수는 Thumb 명령어 세트로 컴파일 해주기 위한 것이다. 그러나 현재의 컴파일러는 하나의 모듈 안에 있는 두 함수를 두 개의 명령어 세트로 동시에 컴파일하지 못하므로, ARM 명령어 세트로 컴파일 될 코드와 Thumb 명령어 세트로 컴파일 될 부분을 두 개의 모듈로 분리(Module Splitting)하여 준다.

모듈을 분리할 때 분리된 모듈들은 해당 모듈의 데이타 부분과 선언 부분을 external로 공유하고, 실제 정의된 함수 코드 부분만을 ARM과 Thumb 명령어 세트 중 유리한 모듈이 나누어 갖게 된다.

### 3.4 예상 성능 분석

개선 대상 함수들의 코드 사이즈와 사이클 카운트의 점유율을 분석하여 예상 성능을 분석했다. 개선 대상 함수들은 모듈 수준에서 명령어 세트가 선택될 때 ARM 코드로 생성되었던 모듈을 구성하는 함수가 Thumb 명령어 세트를 선택한 경우의 함수들이다.

원래는 Thumb 명령어 세트로 컴파일 할 함수들이 ARM으로 컴파일 된 것이 코드 사이즈의 8%를 차지하고 있고, 사이클 카운트의 2.1%를 차지하고 있다. 이는 원래 ARM으로 컴파일 되어야 할 함수가 ARM으로 컴파일 된 것이 코드 사이즈의 7.5%, 사이클 카운트의 90.6%를 차지하고 있는 것에 비하면 매우 낮은 비율로 이 부분을 Thumb 명령어 세트로 바꿔더라도 성능에 큰 영향을 미치지 않음을 알 수 있다.

제안한 방법은 명령어 세트를 모듈 수준에서 적용하지 않고 더욱 세부화하여 함수 수준에서 적용함으로써 코드 사이즈 절감 효과를 얻게 된다. 또 이러한 코드들은 성능에 영향이 적으므로 실행시간의 저하가 크지 않을 것이다.

## 4. 구현 및 실험 결과

앞서 제안한 함수 수준에서의 명령어 세트 선택 방법을 구현하여 기존의 방법들과 코드 사이즈 및 성능을 비교, 분석하여 보았다.

### 4.1 구현

그림 2와 같이 프로세스를 구성하였다. 먼저 벤치마크 프로그램들을 ARM과 Thumb 모드로 각각 컴파일을 하고, 생성된 실행 코드를 시뮬레이터에서 실행시켜 프로파일러로 분석한다. 생성된 실행 코드와 컴파일 정보, 프로파일링 결과를 분석하여 ARM으로 컴파일 될 함수를 선택한다. 여기까지는 모듈 수준에서와 함수 수준에서의 진행이 동일하고, 다음 단계에 모듈을 분리하는 단계를 추가했다. 앞 단계에서 작성한 함수의 목록을 가지고 해당 모듈을 ARM 모듈과 Thumb 모듈로 분할한다. 선택된 명령어 세트대로 모듈들을 다시 컴파일 하여 실행 코드를 얻어내어 성능을 측정한다.

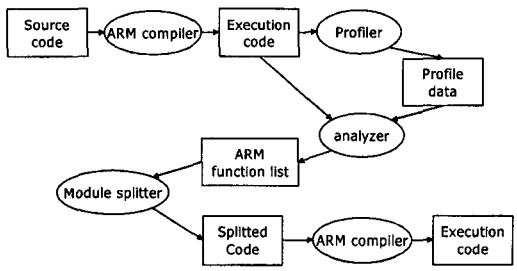


그림 2 구현 프로세스

이 실험을 위해 간단한 함수 수준 분석기(Function level analyzer)와 모듈 절단기(Module splitter)를 작성하였다. 함수 수준 분석기는 실행 코드와 프로파일링 결과를 바탕으로 ARM 모드로 컴파일 되어야 할 대상인 함수들을 결정하여 출력하고, 모듈 절단기는 분석기의 결과를 토대로 모듈을 유리한 함수에 따라 두 개의 모듈로 나눠주는 작업을 한다.

### 4.2 실험 환경

실험을 위해 컴파일러, 시뮬레이터, 프로파일러, 벤치마크 프로그램, 프로그래밍 언어가 필요하다. ARM 프로세서에서 사용 가능한 개발 환경으로 gcc와 ARM

SDT가 대표적인데, 본 연구에서는 ARM사에서 제공하고 있는 ARM SDT(ARM Software Development Toolkit) v2.5[5]를 기본 환경으로 사용하였다.

- 최적화 컴파일러: ARM SDT는 명령어 세트에 따라 아래와 같이 두 가지의 컴파일러를 기본적으로 제공한다.
  armcc: 32비트 ARM 컴파일러
  - tcc: 16비트 Thumb 컴파일러
  두 컴파일러는 함수의 경계에서 다른 모드를 가진 함수를 호출하는 경우를 처리해 주기 때문에 두 가지 모드를 가진 코드를 혼합하여 사용할 수 있다. 이 실험은 임베디드 환경을 대상으로 하기 때문에 코드 사이즈를 크게 하는 최적화를 수행하지 않는 한도에서 최대의 최적화를 수행하는 -O2 -Ospace 수준에서 실험했다. ARM SDT에는 gcc와 달리 -O3 레벨의 최적화가 없고 -O2 레벨이 최대이다. 어셈블러와 Linker는 모드에 상관없이 사용 가능하다
- 프로세서 시뮬레이터: ARM SDT에 포함된 명령어 세트 시뮬레이터(Instruction Set Simulator)인 ARMulator를 사용했다. ARM에서 제작한 여러 프로세서에서 사용 가능하며, 본 실험에서는 프로세서를 ARM7-TDMI 100㎒로 설정하였다.
- 프로파일러: armprof는 ARM SDT에 포함된 프로파일러로 해당 프로그램을 구성하는 함수의 전체 실행 시간에서 차지하는 비율 및 호출 회수 등의 프로파일 정보를 제공한다.
- 벤치마크 프로그램: 임베디드 도메인에 적합한 벤치마크 프로그램인 mediabench[6]를 사용하였다.
- 개발 언어: 함수 수준 분석기와 모듈 분리기를 작성하기 위한 리포팅 언어로 Perl을 사용했다. Perl 프로그램으로 ActiveState Active Perl 5.8[7]을 사용하였다.

### 4.3 실험 결과

함수 수준에서 명령어 세트를 선택했을 때의 코드 사이즈 및 성능을 측정하기 위해서 비교 대상을 설정했다. 프로그램 전체를 ARM 또는 Thumb 명령어 세트로 컴파일 했을 경우와 모듈 수준에서 명령어 세트가 결정될 경우를 비교 대상으로 설정했다. 이를 통해 모듈 수준과의 상대적인 비교에 추가로 전체적인 압축률과 성능도 측정했다.

#### 4.3.1 프로그램 수준과 모듈 수준에서 명령어 세트의 선택 시 결과 비교

Thumb 명령어 세트는 ARM 명령어 세트에 비해 코드 사이즈는 감소시키지만 실행시간에 있어서 불이익이 있다. [3]의 실험에 따르면 Thumb 명령어 세트를 이용했을 때 코드 사이즈는 30% 줄어들고 실행시간은 10%~15% 증가한다. 하지만 결과는 벤치마크 프로그램의 종류, 하드웨어 특성, 컴파일러의 종류와 최적화 정도에 따라 달라지므로 본 실험에서 사용할 벤치마크와 시뮬레이터, 컴파일러의 환경에서는 어떠한 결과가 나오는지 확인하여 본 실험의 결과 분석의 기초 자료로 활용했다.

실험 결과는 표 1과 같다. 결과는 코드 사이즈의 크기 비교와 사이클 카운트의 비교로 요약된다.

코드 사이즈 및 사이클 카운트의 비교에서 비율을 나타내는 방법으로 비율의 평균과 값들의 합의 비율을 함께 사용했다. 실험 대상 프로그램 전체를 하나의 패키지로 생각하면 값들의 합의 비율에 의미가 있고, 독립된 프로그램들로 보면 비율의 평균이 의미가 있기 때문이다.

또한 코드 사이즈의 측정은 전체 실행 코드의 크기로 측정하였다. 본 실험의 결과는 code segment만을 줄일 뿐, data segment의 크기는 줄이지 못하지만 전체적인 성능의 향상을 확인하기 위해서 data segment를 포함한 전체 실행 코드의 크기로 측정하였다.

전체적으로 보면 코드 사이즈는 약 20% 작아지고 사

표 1 프로그램 수준에서 명령어 세트를 선택 시 결과 비교

| Benchmark | code size | | | cycle count | | |
|---|---|---|---|---|---|---|
| | ARM | Thumb | Thumb/ ARM | ARM | Thumb | Thumb/ ARM |
| adpcm.rawcaudio | 3784 | 3660 | 96.7 | 9499 | 10613 | 111.7 |
| adpcm.rawdaudio | 3892 | 3660 | 94.0 | 27917 | 43600 | 156.2 |
| epic.epic | 18856 | 13416 | 71.2 | 153045317 | 200952474 | 131.3 |
| g721.encode | 6324 | 4704 | 74.4 | 2761147 | 3417774 | 123.8 |
| g721.decode | 6352 | 4716 | 74.2 | 3777855 | 4726282 | 125.1 |
| jpeg.cjpeg | 108040 | 76368 | 70.7 | 21405688 | 26805619 | 125.2 |
| jpeg.djpeg | 104052 | 73592 | 70.7 | 8907053 | 12397004 | 139.2 |
| mesa.osdemo | 478244 | 389684 | 81.5 | 5761626 | 7399585 | 128.4 |
| mesa.texgen | 478452 | 389872 | 81.5 | 128952913 | 174645015 | 135.4 |
| 비율의 평균(%) | | | 80.5 | | | 130.7 |
| 합의 비율(%) | | | 79.4 | | | 132.6 |

이클 카운트는 약 30% 증가했다. 따라서 본 실험의 대상이 되는 실험 환경은 [3]의 실험에 비해 Thumb 명령어 세트에 유리하지 못한 환경이다. 하지만 본 실험의 목적이 ARM 및 Thumb 코드와의 비교가 아니라, 명령어 세트를 모듈 별로 선택하는 것에 대한 성능 비교이기 때문에 이에 따른 영향은 없다.

이 실험으로 프로그램의 특성에 따라 정도는 다르지만 Thumb 명령어 세트를 사용했을 때, 일반적으로 코드 사이즈는 감소하고 실행시간은 증가하는 것을 확인할 수 있다.

4.3.2 모듈 수준과 함수 수준에서 명령어 세트의 선택 시 결과 비교

실험에 있어서 중요한 변인은 해당 함수가 전체 실행시간에서 차지하는 비율이다. 편의를 위해, 전체 실행시간 중 해당 함수가 차지하는 비율이 T% 이상인 함수를 자주 사용되는 함수라고 정의하고 이때의 비율 T를 명령어 세트를 선택할 때 사용되는 "실행시간에서 차지하는 비율"이라고 정의한다.

본 실험은 다양한 실행시간에서 차지하는 비율에서 수행되었으나, 지면 관계상 5%에서의 결과를 자세히 소개한 후 다른 비율에서의 추세를 분석하고 이를 토대로 최적의 코드 특성을 갖는 비율을 찾기로 한다.

**• 실행시간에서 차지하는 비율이 5%일 때**

표 2와 표 3은 각각 실행시간에서 차지하는 비율이 5%일 때의 모듈 및 함수의 코드 사이즈 및 사이클 카운트를 비교하고 있다.

먼저 코드 사이즈를 살펴보면, 전체 프로그램을 Thumb로 생성하면 코드 사이즈가 ARM에 비해 80.5% 작아지는데, 모듈 수준에서 선택하면 84.4% 작아지는데 그쳤다. 그러나 함수 수준에서 선택하면 82.1% 작아져서 모듈에 비해서 2.7% 정도 추가적으로 작아지고 있는 것을 알 수 있다.

이때 작아지는 비율은 앞장에서 살펴봤던 개선 대상이 되는 함수들의 코드 사이즈와 관련이 있다. 개선 대상이 되는 함수들의 코드 사이즈가 크면 클수록 코드 사이즈가 작아질 가능성이 더 크다. 하지만 프로그램의 특성에 따라 코드 사이즈가 줄어드는 비율이 일정치 않기 때문에 반드시 비례하는 것은 아니다.

사이클 카운트를 살펴보면 Thumb가 ARM에 비해 30.7% 커질 때, 모듈 수준에서는 7.4% 커졌으나 함수

표 2 코드 사이즈 비교(실행시간에서 차지하는 비율이 5%일 때)

| Benchmark | Thumb/ ARM | Module | Module/ ARM | Function | Function/ ARM | 개선 |
|---|---|---|---|---|---|---|
| adpcm.rawcaudio | 96.7 | 3740 | 98.8 | 3720 | 98.3 | 0.5 |
| adpcm.rawdaudio | 94.0 | 3740 | 96.1 | 3708 | 95.3 | 0.9 |
| epic.epic | 71.2 | 14096 | 74.8 | 13840 | 73.4 | 1.8 |
| g721.encode | 74.4 | 5564 | 88.0 | 5132 | 81.2 | 7.8 |
| g721.decode | 74.2 | 5576 | 87.8 | 5144 | 81.0 | 7.8 |
| jpeg.cjpeg | 70.7 | 80000 | 74.1 | 77944 | 72.1 | 2.6 |
| jpeg.djpeg | 70.7 | 76016 | 76.7 | 75604 | 72.7 | 5.2 |
| mesa.osdemo | 81.5 | 399536 | 83.5 | 394100 | 82.4 | 1.4 |
| mesa.texgen | 81.5 | 399732 | 83.6 | 394296 | 82.4 | 1.4 |
| 비율의 평균(%) | 80.5 | | 84.4 | | 82.1 | 2.7 |
| 합의 비율(%) | 79.4 | | 81.8 | | 80.6 | 1.5 |

표 3 사이클 카운트 비교(실행시간에서 차지하는 비율이 5%일 때)

| Benchmark | Thumb/ ARM | Module | Module/ ARM | Function | Function/ ARM | 개선 |
|---|---|---|---|---|---|---|
| adpcm.rawcaudio | 111.7 | 9851 | 103.7 | 9851 | 103.7 | 0.0 |
| adpcm.rawdaudio | 156.2 | 28003 | 100.3 | 28003 | 100.3 | 0.0 |
| epic.epic | 131.3 | 151065364 | 98.5 | 151065386 | 98.5 | 0.0 |
| g721.encode | 123.8 | 2817025 | 102.0 | 2886444 | 104.5 | -2.5 |
| g721.decode | 125.1 | 3984587 | 105.5 | 3961049 | 104.9 | 0.6 |
| jpeg.cjpeg | 125.2 | 23538767 | 110.0 | 23549137 | 110.0 | -0.1 |
| jpeg.djpeg | 139.2 | 11570543 | 129.9 | 11014944 | 123.7 | 6.2 |
| mesa.osdemo | 128.4 | 6070106 | 105.4 | 6096455 | 105.1 | -0.5 |
| mesa.texgen | 135.4 | 143658863 | 111.4 | 144351647 | 111.9 | -0.5 |
| 비율의 평균(%) | 130.7 | | 107.4 | | 107.1 | 0.2 |
| 합의 비율(%) | 132.6 | | 105.5 | | 105.6 | -0.5 |

수준에서는 7.1% 커져서 오히려 모듈에 비해서 0.2% 정도 추가적으로 사이클 카운트가 작아지고 있다. 함수 수준에서 명령어 세트를 선택하는 것이 모듈 수준보다 사이클 카운트를 약간 크게 만들 것이라는 예상과는 다른 결과이다.

이러한 결과는 다른 명령어 세트를 가진 함수들 간의 호출에 있어서 페널티로 설명할 수 있다. 서로 다른 명령어 세트를 가진 함수들 간에 호출될 경우에 컴파일러는 veneer라는 코드를 통해 호출하는데[4], 이 veneer에서 모드 전환을 위해 레지스터의 마지막 비트를 0 또는 1로 세팅한 후, BX 명령어를 수행하게 되고 이때 실행 속도에 페널티가 발생한다. ARM과 Thumb 명령어 세트를 혼합해서 프로그램을 생성할 때 다른 모드를 가진 함수 간의 잦은 호출은 성능에 나쁜 영향을 미치고, 자주 호출되는 함수들이 같은 모드를 가지게 되면 다른 모드를 가지게 될 때 보다 성능이 향상된다. 속도가 오히려 빨라진 g721.decode와 jpeg. djpeg은 함수 별로 명령어 세트가 결정될 때 잦은 호출 관계에 있는 함수들이 같은 모드로 재배열되므로 실행 속도가 빨라졌다.

**• 실행시간에서 차지하는 비율에 따른 변화 추이**

앞서 제시한 실행시간에서 차지하는 비율이 5%일 경우와 같은 방식으로 실행시간에서 차지하는 비율을 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, 10%, 5%, 3%, 1%, 0.5%로 바꿔서 실험하고, 결과의 평균을 그림 3과 그림 4에서 그래프로 나타냈고, 응용 프로그램별 추이를 그림 5와 6에서 나타내고 있다.

그림 3에서 보면 코드 사이즈는 실행시간에서 차지하는 비율이 낮아짐에 따라 처음엔 완만히 증가한다. 모듈과 함수에서의 코드 사이즈 증가 양상은 약간 다른데, 일반적으로 모듈에서의 코드 사이즈가 함수에서 보다 크다. 40% 구간에서 차이가 최대가 되고 점차 줄어들다가 0%에서 최소가 된다. 모듈에서는 50%~40% 구간에서 크게 증가한 후 다시 완만히 증가하다가 5%~0.5% 구간에서 급격히 증가하다가 0.5% 이후에 거의 수직으로 증가한다. 0.5% 이후의 구간에는 모든 함수가 다 포함되기 때문에 코드 사이즈의 상승폭이 매우 크다. 함수에서도 역시 초반에 완만히 증가하다가 30% 구간 이후 조금 더 가파르게 상승한다. 역시 5% 이후에 급격히 증가하고 0.5% 이후 수직으로 증가한다.

그림 4에서 보면 사이클 카운트는 일정하게 줄어들다가 해당 구간에 포함된 함수가 많아지는 50%~20% 구간과 5%~0.5% 구간에서 급격히 줄어든 후, 0.5% 이후 거의 수직으로 줄어든다. 사이클 카운트에서 모듈과 함수의 성능 차이는 40%~30% 구간을 제외하면 크게 나타나지 않는다.

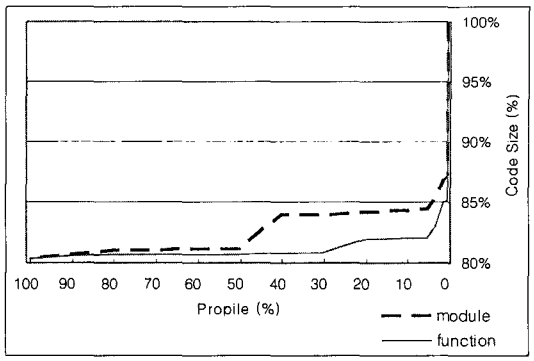


그림 3 실행시간에서 차지하는 비율에 따른 코드 사이즈의 변화 추이

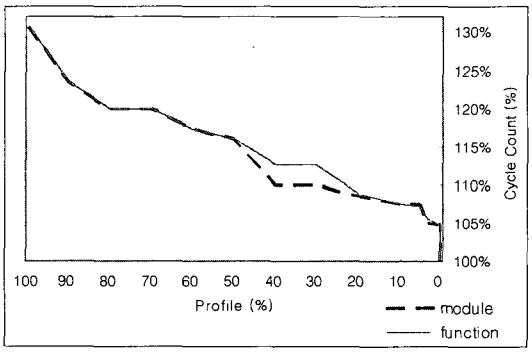


그림 4 실행시간에서 차지하는 비율에 따른 사이클 카운트의 변화 추이

설정된 실행시간에서 차지하는 비율의 구간에 포함되는 함수들은 초반에는 코드 사이즈에 비해 실행 시간이 매우 큰 함수들이고 0%로 가면 갈수록 코드 사이즈에 비해 실행시간을 적게 차지하는 함수들이 된다. 이때 함수 별 코드 사이즈는 실행시간에서 차지하는 비율과 관계없이 해당 구간에 해당하는 함수의 개수에만 비례한다. 이러한 특성들 때문에 실행시간에서 차지하는 비율이 큰 초반에는 포함되는 함수의 수가 적으므로 코드 사이즈가 크게 증가하지 않다가, 후반부에 들어 대부분의 함수가 포함되면 코드 사이즈가 급격히 증가하는 양상을보인다. 사이클 카운트는 초반부터 영향이 매우 큰 함수들로 구성되므로 처음부터 성능의 향상이 꾸준히 이루어지게 된다.

이러한 코드 사이즈와 사이클 카운트의 추이를 고려하면 각 응용 프로그램의 사이즈 및 사이클 카운트의 변화도 앞서의 평균 곡선과 같이 코드 사이즈는 모듈이 함수의 위에 있고 사이클 카운트는 아래에 있는 형태를 따르고 있으므로 코드 사이즈가 급격히 증가하기 전에

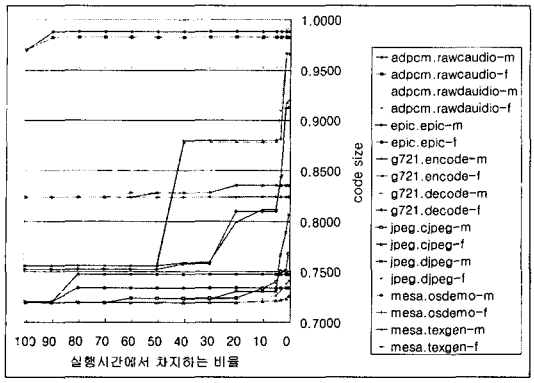


그림 5 응용 프로그램별 실행시간에서 차지하는 비율에 따른 코드 사이즈의 변화 추이

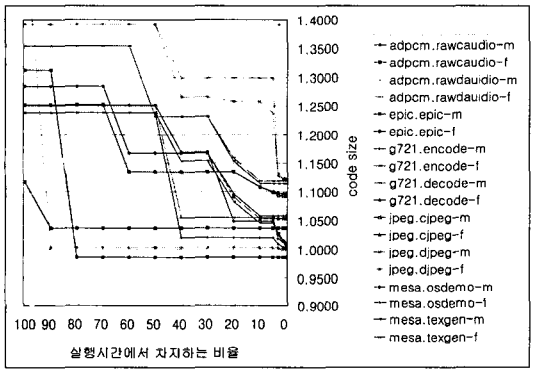


그림 6 응용프로그램별 실행시간에서 차지하는 비율에 따른 코드 사이즈의 변화 추이

사이클 카운트가 개선되는 곳에서 최적의 코드를 찾아낼 수 있다.

**• 최적의 비율 : 5% 모듈 대 3% 함수**

일반적으로 통용되는 가장 적합한 실행시간에서 차지하는 비율을 찾는 것은 매우 어려운 일이다. 프로그램의 종류 및 특성에 따라 가변적이기 때문이다. 그러나 해당 프로그램 세트가 고정되면, 그 세트에 대한 최적의 실행시간에서 차지하는 비율을 찾을 수 있다. 이런 배경에서 본 벤치마크를 하나의 패키지로 보고 최적의 비율을 찾아보았다.

모듈의 경우 5% 이전에는 실행시간은 적당히 감소하면서 코드 사이즈는 완만히 증가하지만 5% 이후에는 코드 사이즈가 급격히 증가하고 실행시간도 급격히 감소한다. 하지만 원래 기본적인 코드 사이즈가 크므로 약 5% 대가 최적의 비율이라고 할 수 있다. 함수의 경우 5%~3%에서 코드 사이즈가 급격히 증가하지만 모듈에 비해 기본적으로 코드 사이즈가 작기 때문에, 5%~3% 대에서 얻어지는 가파른 실행시간의 향상을 추가적으로 얻어 낼 수 있다. 함수의 최적 비율은 약 3%이다.

위와 같이 명령어 세트를 함수 수준에서 선택할 경우에는 최적의 실행시간에서 차지하는 비율을 낮출 수 있으므로 이를 통한 개선 효과를 얻을 수 있다. 개선 효과를 알아보기 위해 5% 모듈과 3% 함수를 비교해보면, 코드 사이즈는 1.8% 줄어들면서 사이클 카운트 역시 1.6% 줄어들고 있다.

## 5. 결론 및 향후 연구 과제

### 5.1 결 론

기존의 모듈 수준에서 명령어 세트를 선택해서 코드를 생성하는 알고리즘은 우수하지만 명령어 세트의 선택이 모듈 수준에서 이루어지기 때문에 모듈이 포함하고 있는 여러 함수들의 특성을 최적으로 반영한 코드를 생성하지 못한다.

이를 해결하기 위해 함수 수준에서 명령어 세트를 선택하는 방법을 고안하였다. 모듈을 유리한 명령어 세트에 따라 두 부분으로 분리시켜서 프로그램의 모든 함수들은 각 함수에서 가장 유리한 명령어 세트를 가진 코드로 컴파일 될 수 있었다.

함수 수준에서 명령어 세트를 선택했을 때 개선의 대상이 되는 함수들의 코드 사이즈는 약 8%, 사이클 카운트의 2% 정도를 차지하고 있었는데, 새로 제안한 방법을 사용하여 이 부분에 대한 추가적인 코드 사이즈 최적화를 진행한 결과, 전체 코드 사이즈가 2.7%가 줄어들고, 전체 사이클 카운트는 0.2% 줄었다. 이렇게 명령어 세트의 선택을 모듈 수준에서 함수 수준으로 바꿈으로써 실행시간의 손실 없이 코드 사이즈를 추가적으로 줄일 수 있었다.

또한 실행시간에서 차지하는 비율의 조절을 통해서 전체 패키지에 대한 최적의 실행시간에서 차지하는 비율을 구할 수 있었는데, 함수 수준에서 최적의 비율을 반영할 경우 모듈 수준에서의 선택보다 코드 사이즈에서 1.8% 추가 감소와, 사이클 카운트에서 1.6%의 성능 향상을 얻을 수 있었다.

### 5.2 향후 연구 과제

향후 연구과제로는 몇 가지를 들 수 있다.

라이브러리에 대해 본 실험을 적용해야 한다. 임베디드 시스템에서 보통 라이브러리가 -static 옵션에 의해 배포되는 코드에 포함되는 것을 고려하면 이러한 연구는 매우 중요하다. ARM의 SDT에서는 혼합된 명령어 세트를 가진 코드는 Thumb 라이브러리만을 사용하도록 고정되어 있기 때문에, 이 연구를 위해서는 새로운 실험 환경을 구성해야 한다.

코드 사이즈는 전체적으로 작아지는 반면, 사이클 카운트는 불규칙적으로 커지거나 감소하는 경향을 보이는

데, 이는 서로 호출 관계에 있는 함수들 간의 명령어 세트가 동일한지 여부에 영향을 받기 때문으로 보인다. 호출되는 함수간의 명령어 세트의 영향과 호출 회수를 고려하여 각 함수의 명령어 세트를 선택할 수 있다면 추가적인 실행시간과 코드 사이즈 향상을 얻을 수 있다.

본 논문에서는 함수 수준에서 명령어 세트를 선택하여 생성시킨 코드가 모듈 수준에서 선택한 경우보다 우월한 성능을 갖는다는 것을 실험하는데 초점을 두었기 때문에, 모듈을 분리하는 부분을 간단히 구현했다. 분리 컴파일, 모듈을 정확히 분리하고 분리된 모듈의 동일성을 증명하는 문제도 매우 중요한 과제이다.

컴파일러의 처리 단계 중 명령어 선택 단계에 본 아이디어를 적용해 본다. 두 개의 명령어 세트가 혼합된 코드를 생성하기 위한 방법으로, 현재의 컴파일러들은 두 개의 컴파일러로 구현되어 있다. 이러한 방법은 구현의 복잡성을 줄여 주는 효과는 있으나, 혼합된 코드를 생성해낼 때 얻을 수 있는 장점들을 최대로 활용하지 못한다. 따라서 명령어 선택 단계에 함수 혹은 베이직 블록 단위로 명령어 세트를 혼합하여 코드를 생성한다면, ARM과 같이 두 개의 명령어 세트를 가진 프로세서의 최적화에 큰 도움이 될 것이다.

## 참 고 문 헌


[1] P. Marwedel, S. Steinke and L. Wehmeyer, "Compilation techniques for energy-, code-size-, and run-time-efficient embedded software," Int. Workshop on Advanced Compiler Techniques for High Performance and Embedded Processors, Bucharest, 2001.

[2] L. Goudge and S. Segars, "Thumb: Reducing the cost of 32-bit risc performance in portable and consumer applications," In Precedings of COMPCON 96, 1996.

[3] A. Halambi, A. Shrivastava, P. Biswas, N. Dutt, and A. Nicolau, "An efficient compiler technique for code size reduction using reduced bi-width ISAs," In Proc. of Design Automation and Test in Europe(DATE), 2002.

[4] A. Krishnaswarmy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," CM SIGPLAN Joint Conference on Languages Compilers and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES/SCOPES), Berlin, Germany, June 2002.

[5] ARM, "ARM Software Development Toolkit Version 2.50 User Guide," ARM DUI 0040D.

[6] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," IEEE/ACM International Symposium on Microarchitecture (MICRO), Research Triangle Park, North Carolina, December 1997.

[7] ActiveState Perl Homepage, http://www.activestate.com/



소 창 호

1998년 고려대학교 컴퓨터학과 학사 1998년~2001년 신세기통신 정보시스템실. 2004년 한국과학기술원 전산학과 석사. 2004년~현재 삼성전자 무선사업부 북미SW랩 선임연구원



한 태 숙

1976년 서울대학교 전자공학과 학사. 1978년 KAIST 전산학과 석사. 1990년 Univ. of North Carolina at Chapel Hill 박사 현재 KAIST 전산학과 교수 관심분야는 프로그래밍 언어론, 함수형 언어, 임베디드 시스템 명세언어 등