임베디드 응용 프로그램 성능 분석[1)]

김 선 욱, 오 재 근, 한 영 선, 최 홍 욱, 김 철 우
고려대학교 공과대학 전자컴퓨터공학과
E-mail: {seon,worms97,youngsun,turn2u,ckim}@korea.ac.kr

Performance Analysis of Embedded Applications

Seon Wook Kim, Jaegeun Oh, Young Sun Han, Hongwook Choi, Chulwoo Kim
Department of Electronics and Computer Engineering, Korea University
E-mail: {seon,worms97,youngsun,turn2u,ckim}@korea.ac.kr

Abstract

This paper presents performance analysis of the embedded application, called EEMBC consisting of 5 categories and total 34 applications. We measured various performance metrics, such as code sizes, TLP(Thread Level Parallelism) using OpenMP API and ILP(Instruction Level Parallelism) on ARM-modeled SimpleScalar in detail. We show that the embedded applications have the similar characteristics as integer applications to deliver low ILP and TLP in our environment. They have many small loops, which result in large instruction overhead in TLP and loop control overhead in ILP.

I. 서론

과거 컴퓨터 연구 분야는 대부분 고성능의 워크스테이션이나 서버의 설계, 그리고 이런 고성능 장비에서의 공학 응용 프로그램을 빠르게 수행하는데 초점이 맞추어져 있었다. 하지만 현재의 추세는 이러한 분야에서 멀티미디어 분야로, 또한 고성능의 워크스테이션이나 서버에서 임베디드 시스템을 이용하는 것으로 빠르게 변해가고 있으며, 임베디드 시스템 상에서의 멀티미디어 프로그램의 중요성도 점점 더 늘어나고 있다. 일반적으로 임베디드 시스템에서 사용되는 프로그램은 과거 고성능 컴퓨터에서 사용하던 공학, 과학 프로그램에 비해서 작업량이 적고, 임베디드 시스템의 경우 워크스테이션이나 서버에 비해서 제한된 리소스, 예를 들어 작은 메모리를 갖기 때문에 응용 프로그램의 실행 파일의 코드 크기나 메모리 사용량 부분에서 많은 제한을 갖는다.

본 논문은 일반적인 임베디드 응용 프로그램의 특성과 성능을 분석하여 임베디드 시스템의 최적화 설계의 방향을 제시하고자 한다. 우리는 임베디드 응용 프로그램의 분석을 위해 EEMBC(EDN Embedded Microprocessor Benchmark Consortium) 벤치마킹 프로그램을 사용하였다[1]. 프로그램의 특성 및 성능을 코드 사이즈, ILP(Instruction Level Parallelism)와 TLP(Thread Level Parallelism)를 기준으로 분석하였다. 마이크로 아키텍처에 관한 성능을 분석하기 위해서 ARM 프로세서 구조를 시뮬레이션 하는 SimpleScalar를 사용하고, TLP를 측정하기 위하여 공유 메모리(shared-memory)상에서 OpenMP API를 사용하였다.

결론적으로, 우리가 측정한 임베디드 응용 프로그램에서는 integer 프로그램과 비슷한 특성, 즉 낮은 ILP와 TLP를 갖고 있었다. 이는 기존의 임베디드 프로세서 구조나 컴파일러 기능이 충분한 연산 리소스를 사용하지 못한다는 것을 보여 준다.

1) 고려대학교 특별연구비에 의하여 수행되었음. 또한 본 논문은 부분적으로 2003년 Workshop on Applications and Tools에 게재 되었음.

Ⅱ. EEMBC

우리는 연구에서 임베디드 응용 프로그램의 성능을 분석하기 위해서 현재 약 45개 국가에서 사용되고 있는 EEMBC(EDN Embedded Microprocessor Benchmark Consortium) 벤치마크를 사용하였다. EEMBC는 임베디드 시스템에 사용할 수 있는 다양한 응용 프로그램의 집합으로써 유사한 기능을 하는 프로그램들을 크게 5개의 범주로 나누었고, 그 안에 다양한 알고리즘을 포함하고 있다. EEMBC의 범주에는 가전(Consumer), 통신(Telecommunications), 네트워크(Networking), 사무자동화(Office automation), 자동차와 산업(Automotive & Industrial)이 있고, EEMBC는 총 34개의 독립된 프로그램으로 이루어져 있다. 가전 범주 안의 프로그램들로는 RGB to CMYK/YIQ Conversion, high-pass gray-scale filter, JPEG (de)compression 이 있고, 통신 범주에는 autocorrelation, convolution encoder, bit allocation, FFT, Viterbi decoder가 있다. 네트워크 범주 안에는 Dijkstra's shortest-path, packet flow 와 routelookup 이 있고, 사무 자동화 범주 안에는 Bezier curve calculation, dithering, image rotation이 있으며 마지막으로 자동차와 산업 범주 안에는 table lookup & interpolation, tooth to spark, angle to time conversion, pulse-width modulation, remote data request, road speed calculation, (in)finite impulse response filter, bit manipulation, basic arithmetic, pointer chasing, matrix arithmetic, cache buster, inverse DCT, FFT가 있다. EEMBC 벤치마크는 3부분으로 구성되어 있는데 첫 번째로 입력 장치로부터 데이터를 읽어 들이는 부분, 두 번째는 알고리즘을 이용해 실행하는 부분이 있고, 마지막으로는 출력장치를 통해 실행 결과를 보여주게 되는데, 실행 결과로는 실행 시간과 알고리즘에서의 처리량(iteration/second)을 보여준다.

Ⅲ. EEMBC의 성능 분석

3.1 실험 방법

우리는 프로그램의 특성 및 성능을 파악하기 위해서 코드 사이즈, 실행 시간, ILP(Instruction Level Parallelism), TLP(Thread Level Parallelism)을 기준으로 분석하였다. 우선적으로 EEMBC에서의 프로그램이 임베디드 시스템에 적당한 것인가를 알아보고 성능을 향상시킬 수 있는 방법에 대해 알아봤다.

3.2 성능 분석

3.2.1 코드 사이즈 (Code Size)

코드 사이지는 제한된 메모리 크기를 갖는 임베디드 시스템에서 성능을 결정짓는 가장 중요한 요소들 중의 하나이다. 우리는 프로그램의 코드 사이즈를 분석하기 위해서 인텔 컴파일러 7.0을 사용했다.

표 1. code size (단위 : KB)

Application name	text size	data size	bss size	Application name	text size	data size	bss size
cjpeg	46	238	518	pktflowb2m	21	10	5
djpeg	54	31	773	pktflowb4m	21	10	5
rgbcmy	20	230	6	routelookup	22	14	6
rgbhpg	20	80	6	bezier01fixed	20	24	5
rgbyiq	20	230	6	bezier01float	20	43	5
autcor00data1	21	5	5	dither	20	69	5
autcor00data2	21	7	5	rotate	21	18	18
autcor00data3	21	7	5	text	21	23	45
conven00data1	21	7	5	a2time	22	7	6
conven00data2	21	7	5	aifftr	23	21	32
conven00data3	21	7	5	aifirf	21	10	5
fbital00data2	21	7	5	aiifft	23	21	32
fbital00data3	21	7	5	basefp	21	21	6
fbital00data6	21	7	5	bitmnp	23	6	7
fft00data1	21	12	5	cacheb	22	6	6
fft00data2	21	12	5	canrdr	22	11	5
fft00data3	21	9	5	idctrm	25	13	7
viterb00data1	23	6	5	iirflt	24	10	5
viterb00data2	23	6	8	matrix	26	32	6
viterb00data3	23	6	8	pntrch	21	10	5
viterb00data4	23	6	9	puwmod	22	15	6
ospf	24	6	5	rspeed	20	7	5
pktflowb512k	21	10	5	tblook	21	17	5
pktflowb1m	21	10	5	ttsprk	25	54	6

표 1에서 보여주는 것처럼 EEMBC에 포함되어 있는 프로그램들은 가장 큰 것이 *cjpeg*과 *djpeg*의 경우로써 858KB와 802KB였고 다른 프로그램들의 경우에는 100KB이하로 매우 작다는 것을 알 수 있다.

3.2.2 ILP (Instruction Level Parallelism)

본 논문에서 임베디드 응용 프로그램의 마이크로 아키텍처에 관한 성능을 분석하기 위해서 ARM프로세서의 구조를 시뮬레이션 하는 SimpleScalar 시뮬레이터를 사용했다[3]. 우리는 시뮬레이션을 위해서 EEMBC 프로그램들의 코드를 gcc cross compiler를 사용했고 컴파일 옵션으로는 -O2를 사용했다. 시뮬레이션을 위한 환경으로써 L1 cache로 data cache 와 instruction cache 모두 8KB directed-map을 사용하고 L2 cache로 64KB를 사용하고 out-of-order로 명령을 수행했다.

SimpleScalar로 시뮬레이션 할 경우 4-way issue로 최대 4개의 명령어를 동시에 처리할 수 있다. 하지만 그림 1에서 볼 수 있는 것처럼 시뮬레이션 결과 자동차와 산업 범주의 *pntrch* 프로그램이 가장 큰 값으로 2.4정도의 IPC를 갖고 대부분의 프로그램들이 1과 2사

이의 낮은 IPC를 갖는 것을 알 수 있다. 즉, superscalar 구조를 갖는 시스템에서는 효과를 얻을 수 없다는 것을 보여준다. 그 이유는 대부분의 코드들이 많은 작은 loop으로 구성되어 있어 loop control overhead가 크기 때문이다. loop 자체는 병렬화가 가능하나 loop bound expression에서 pointer를 포함하고 있어 컴파일러가 loop-unrolling에 의한 IPC 증가 방법을 적용할 수 없었다. 그리고 EEMBC의 경우 같은 알고리즘을 반복적으로 수행하기 때문에 L1 cache miss가 매우 작다는 것을 알 수 있었다.

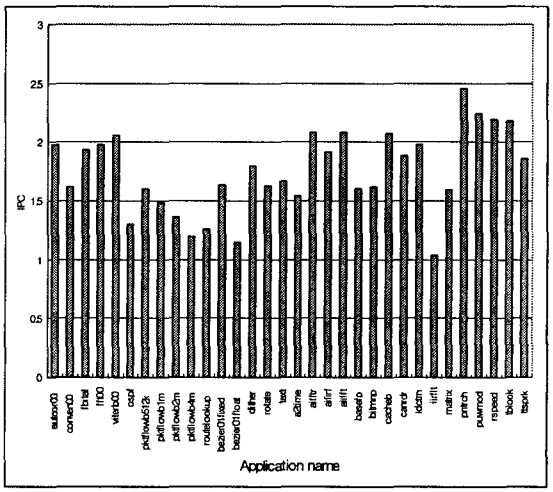

그림 1 ARM-modeled SimpleScalar에 의한 IPC.

3.2.3 TLP (Thread Level Parallelism)

오늘날 고성능 임베디드 프로세서의 요구가 증대되고 이를 위한 하나의 방법으로 SMT(Simultaneous Multithreading)의 기술이 고려되고 있다. 이 기술을 적용함으로써 시스템 전체의 throughput을 향상시킬 수 있다. 그리고 다른 방법으로써 TLP를 이용해 한 프로그램 내에서의 실행 속도를 증가시킬 수 있다. 이런 TLP를 이용한 성능향상의 유용성을 검증하기 위해 우리는 EEMBC프로그램에 OpenMP API를 적용하였다.

OpenMP API는 공유메모리상에서 프로그램 코드에 OpenMP directive를 추가해서 프로그램을 병렬화 시킬 수 있고 우리는 이를 통해 TLP 에 의한 성능향상을 검증하였다[2]. 우리는 실험환경으로써 인텔/Xeon 2.0GHZ로 2개의 프로세서를 갖는 PC를 사용했고 OS는 redhat 8.0 Linux를 사용하였다. 그리고 컴파일러로서 OpenMP API를 지원하는 인텔 7.0 C 컴파일러를 -O2 -openmp 옵션과 함께 사용하였다.

우리는 먼저 EEMBC 프로그램 내에서 OpenMP API를 이용하여 수작업으로 모든 병렬화 가능한 구간을 병렬화 하였다. 우리는 각 병렬화 구간의 성능을 얻기 위하여 -openmp_profile옵션을 이용하였다. 또한 병렬화 시킨 구간 내에서 심한 parallelization overhead가 존재할 경우 OpenMP API를 제거함으로써 선택적으로 실행 시간을 최적화 하였다. 우리는 연구를 위해서 간단한 2개의 병렬화 기법인 induction variable substitution[4]과 privatization[5] 기법을 사용하였다.

표 2. 2개의 OpenMP thread를 이용한 speedup.

Application name	unoptimized Speedup	optimized Speedup	Application name	unoptimized Speedup	optimized Speedup
cjpeg	0.29	1	pktflowb2m	1	1
djpeg	0.65	1	pktflowb4m	1	1
rgbcmy	0.86	0.87	routelookup	1	1
rgbhpg	1.2	1.5	bezier01fixed	1.38	1.74
rgbyiq	1.63	1.63	bezier01float	0.59	1
autcor00data1	0.11	1	dither	1	1
autcor00data2	1.32	1.53	rotate	0.93	0.93
autcor00data3	1.56	1.54	text	1	1
conven00data1	1	1	a2time	1	1
conven00data2	1	1	aifftr	0.64	1
conven00data3	1	1	aifirf	0.21	1
fbital00data2	1	1	aiifft	1	1
fbital00data3	1	1	basefp	1	1
fbital00data6	1	1	bitmnp	0.3	1
fft00data1	0.5	1	cacheb	1	1
fft00data2	0.67	1	canrdr	1	1
fft00data3	0.5	1	idctrm	0.13	1
viterb00data1	0.11	1	iirflt	1	1
viterb00data2	0.1	1	matrix	0.08	1
viterb00data3	0.11	1	pntrch	1	1
viterb00data4	0.1	1	puwmod	1	1
ospf	1	1	rspeed	1	1
pktflowb512k	1	1	tblook	1	1
pktflowb1m	1	1	ttsprk	1	1

우리는 34개의 프로그램 중에서 17개의 프로그램 안에서 90개의 병렬화 구간을 찾을 수 있었다. 표 2의 unoptimized speedup에서 보는 것과 같이 우리는 4개의 프로그램 그리고 5개의 테스트 셋(*rgbhpg, rgbyiq, autcor00data2, autcor00data3, bezier01fixed*)에서 속도 향상을 얻을 수 있었다. 하지만 병렬화 구간을 찾을 수 있는 다른 프로그램의 경우 심한 instruction overhead 때문에 실행 시간이 증가하는 것을 볼 수 있다. 또한 한 프로그램 안에서도 병렬 구간에 따라서 속도 향상을 얻을 수 있는 부분과 속도 향상을 얻지 못하는 구간이 있다. 그렇기 때문에 우리는 profiling을 통해 각 병렬 구간 내에서 overhead로 인해 실행 속도가 떨어진 구간을 찾아 OpenMP API를 제거했다. 그렇게 함으로써 우리는 표2의 optimized speedup항목에 있는 속도 증가를 얻을 수 있었다. 표 2에서 보는 것

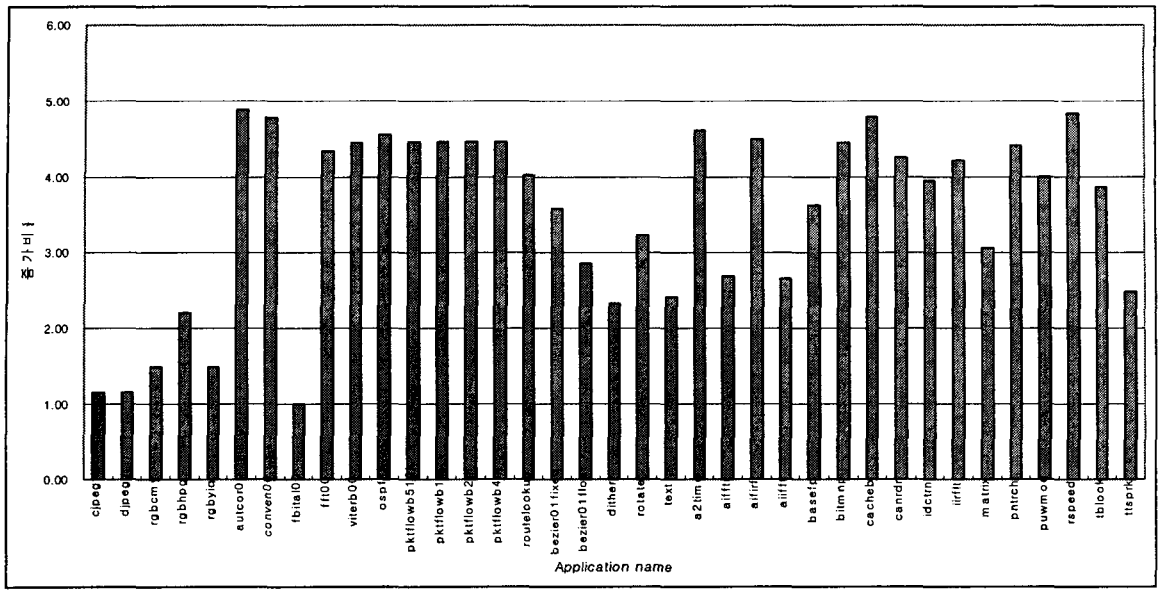

그림 2 OpenMP API를 사용하였을 경우에 사용하지 않았을 경우와 비교한 코드 사이즈의 증가 비율.

과 같은 5개의 프로그램 중에서 3개의 프로그램 (*rgbcmy*, *autcor00data2*, *bezier01fixed*)안에서 더욱 많은 속도 향상효과를 얻을 수 있었다.

우리가 OpenMP API를 이용하여 TLP에 의한 속도 향상을 알아보았다. 하지만 우리는 EEMBC전체 프로그램 중에서 오직 일부분에서 의미 있는 TLP를 찾을 수 있었다. 그 이유는 EEMBC 에서의 프로그램의 경우 크기가 작은 loop이 계속적으로 수행되기 때문에 병렬화 시킬 경우 instruction overhead가 병렬화 효과에 비해서 너무 크다는 것을 알 수 있다. 또한 그림 2에서 알 수 있듯이 OpenMP API를 이용할 경우 프로그램의 코드 사이즈가 *autcor00*의 경우 최대 5배 이상 증가하게 된다. 실제 코드를 text section, data section, bss section으로 나누어서 볼 경우에는 각 section에서 최대 10배까지 크기가 증가하는 것을 볼 수 있었다. 다시 말해서 OpenMP API를 추가해서 컴파일 할 경우 추가 되는 OpenMP 라이브러리에 의해서 병렬화 시키지 않을 경우보다 코드 사이즈가 매우 커지기 때문에 multithreading 기법은 메모리 크기에 제한을 갖는 임베디드 시스템에 적용하기에는 많은 문제가 있다는 것을 알 수 있다.

Ⅳ 결론

본 논문에서는 임베디드 응용 프로그램에 대한 성능을 분석하였다. 실험 결과에서 임베디드 응용프로그램은 기본적인 코드 사이즈에서는 문제가 없었다. 하지만 EEMBC의 프로그램들의 경우 시뮬레이션 결과에서 알 수 있듯이 loop control overhead 때문에 IPC가 매우 작았다. 임베디드 응용 프로그램에서 multithreading을 이용하는 것 또한 문제가 있다는 것을 알 수 있었다. 그 이유는 병렬화 부분이 너무 작아 instruction overhead가 컸으며, multithreaded library에 의한 코드 사이즈가 최대 5배까지 증가하는 것을 볼 수 있다. 이것은 제한된 메모리 사용을 갖는 임베디드 시스템에서는 사용 제약이 된다는 것을 알 수 있다.

Reference

[1] EEMBC(EDN Embedded Microprocessor Benchmark Consortium). http://www.eembc.org/.

[2] OpenMp Forum. http://www.openmp.org/. *OpenMP:A proposed Industry Standard API for Shared Memory Programming, October*, 1997.

[3] SimpleScalar http://www.cs.wisc.edu/~mscalar/simplescalar.html/.

[4] William M. Pottenger. Induction variable substitution and reduction recognition in the polaris parallelizing compiler. Technical Report UIUCDCS-R-98-2072, 1998

[5] Peng Tu and David A. Padua. Automatic array privatization. In *Compiler Optimizations for Scalable Parallel Systems Languages*, pages 247-284, 2001.