On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화

(Memory Hierarchy Optimization in Embedded Systems using On-Chip SRAM)

김 정 원 † 김 승 균 † 이 재 진 †† 정 창 희 ††† 우 덕 균 ††††
(Jungwon Kim) (Seungkyun Kim) (Jaejin Lee) (Changhee Jung) (Duk-Kyun Woo)

요 약 컴퓨터 시스템 분야의 대표적인 문제 중 하나는 메모리의 처리 속도가 CPU의 처리 속도보다 매우 느리기 때문에 생기는 CPU 휴면 시간의 증가, 즉 메모리 장벽 문제이다. CPU와 메모리의 속도 차이를 줄이기 위해서는 레지스터, 캐시 메모리, 메인 메모리, 디스크로 대표되는 메모리 계층을 이용하여 자주 쓰이는 데이터를 메모리 계층 상위, 즉 CPU 가까이 위치시켜야 한다. 본 논문에서는 On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화 기법을 리눅스 기반 시스템에서 최초로 제안한다. 본 기법은 시스템의 가상 메모리를 이용하여 프로그래머가 원하는 코드나 데이터를 On-Chip SRAM에 적재한다. 제안된 기법의 실험 결과 총 9개의 어플리케이션에 대하여 최대 35%, 평균 14%의 시스템 성능 향상과 최대 40% 평균 15%의 에너지 소비 감소를 보였다.

키워드 : 가상 메모리, 메모리 계층, 스크래치패드 메모리, 이종 메모리, 임베디드 시스템, 컴파일러, 데이터 배치, 코드 배치

Abstract The memory wall is the growing disparity of speed between CPU and memory outside the CPU chip. An economical solution is a memory hierarchy organized into several levels, such as processor registers, cache, main memory, disk storage. We introduce a novel memory hierarchy optimization technique in Linux based embedded systems using on-chip SRAM for the first time. The optimization technique allocates On-Chip SRAM to the code/data that selected by programmers by using virtual memory systems. Experiments performed with nine applications indicate that the runtime improvements can be achieved by up to 35%, with an average of 14%, and the energy consumption can be reduced by up to 40%, with an average of 15%.

Key words : Compilers, Code Placement, Data Placement, Embedded Systems, Heterogeneous Memory, Memory Hierarchy, Scratchpad Memory, Virtual Memory

· 이 논문은 2008 한국컴퓨터종합학술대회에서 'On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 서울대학교 컴퓨터공학부
jungwon@aces.snu.ac.kr
seungkyun@aces.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
jlee@cse.snu.ac.kr
††† 정 회 원 : Georgia Institute of Technology, College of Computing
chjung@gatech.edu
†††† 정 회 원 : 한국전자통신연구원 센서네트워크플랫폼연구팀 선임연구원
dkwu@etri.re.kr

논문접수 : 2008년 8월 27일
심사완료 : 2009년 1월 12일

정보과학회논문지: 시스템 및 이론 제36권 제2호(2009.4)

1. 서 론

데스크톱과 마찬가지로 임베디드 컴퓨터 시스템 환경에서 SRAM(Static Random Access Memory)과 DRAM (Dynamic Random Access Memory)은 코드나 데이터를 저장하는 가장 보편적인 메모리다. SRAM은 빠르지만 가격이 비싼 반면, DRAM은 느리지만 가격이 저렴하다. 이 두 메모리의 장점을 동시에 활용하기 위해서는 크기가 큰 DRAM과 크기가 작은 SRAM을 동시에 갖추고 자주 접근하는 코드나 데이터를 SRAM에 할당하는 방법을 사용해야 한다. 이러한 메모리 계층을 이용한 최적화 기법은 SRAM이 매년 60% 속도 향상이 이루어지고 있는 반면 DRAM은 매년 7% 속도 향상이 이루어지고 있음[1]을 고려할 때 앞으로도 매우 효과적인 방

법일 것이다. 컴퓨터 시스템에서 On-Chip SRAM을 사용하는 방법은 하드웨어가 관리하는 캐시(Cache) 메모리를 사용하는 방법과 하드웨어 도움 없이 소프트웨어가 직접 관리하는 스크래치패드(Scratchpad) 메모리를 사용하는 방법으로 나누어진다. 데스크톱에서는 캐시가 주로 사용되지만, 데스크톱 환경보다 자원이 제한적인 임베디드 시스템 환경에서는 스크래치패드 메모리의 단순한 내부 구조와 작은 크기(Die size), 지연시간의 예측 가능성 때문에 캐시보다 유리할 수 있다. 이러한 스크래치패드의 유리한 점 때문에 Motorola MPC500, Analog Devices ADSP-21XX, Philips LPC2290, Analog Devices ADSP-21160m, Atmel AT91-C140, ARM 968ES, Hitachi M32R-32192, Infineon XC166, Analog Devices ADSP-TS201S, Hitachi SuperH-SH7050, Motorola Dragonball 같은 많은 임베디드 시스템이 스크래치패드 메모리를 갖추고 있다[2]. 스크래치패드를 이용한 임베디드 시스템 최적화는 많은 연구가 되어왔다. Avissar와 Barua[3]는 이종 메모리(Heterogeneous Memory)를 갖춘 임베디드 시스템에서 전역 변수나 지역 변수를 효과적으로 각 메모리에 배치하는 컴파일러 기법을 제안하였다. 하지만 우리가 제안하는 방법과 달리 코드 배치는 할 수 없었고, 또한 운영체제 없이 동작하는 프로그램에 적용한 방법이다. Bernhard Egger와 Cho는 [4-7]에서 메모리 관리 장치(Memory Management Unit, MMU)를 갖춘 임베디드 시스템에서 가상 메모리(Virtual Memory)를 이용하여 운영체제가 없이 동작하는 프로그램에서 코드나 데이터를 효과적으로 스크래치패드 메모리에 할당하는 방법을 제안하였다. 또한 Bernhard Egger는 [8]에서 멀티태스킹 환경에서 동작하는 스크래치패드 메모리 관리 기법을 제안하였다. 하지만 리눅스 같은 일반 상용 운영 체제가 아닌 리얼타임 운영체제에서 구현된 방법이라는 점이 우리의 연구와 차별되는 부분이다. 또한 [4-8]은 시뮬레이터에서 실험되었다. 이러한 부분은 실제 시스템에서 실측한 우리의 방법에 비해 실험의 정확성이 떨어진다. Nguyen[9]는 자바 가상 머신(Java Virtual Machine, JVM) 내부에서 스크래치패드 메모리 할당 기법을 구현하여, 컴파일러의 도움 없이 스크래치패드 메모리를 효과적으로 사용할 수 있는 방법을 제안하였다. 하지만 이는 Java 프로그램에서만 가능한 방법이라는 점과 JVM이 알아서 최적화한다는 점에서 스크래치패드 메모리의 장점인 프로그래머의 의도적인 메모리 할당이 불가능하다는 단점이 있다. 위와 같이 스크래치패드 메모리 할당 기법에 대한 연구는 계속되어왔지만, 아직 리눅스나 윈도우CE 같은 상용 임베디드 운영체제에서의 스크래치패드 메모리 할당 기법에 관한 연구는 전무한 상황이다. 최근 들어 리눅스나 윈도우CE를 탑재한 PDA나 스마트폰 같은 임베디드 시스템이 시장에 출시되고 있는 상황을 볼 때, 이러한 상용 운영체제 기반 임베디드 시스템의 스크래치패드 메모리 연구는 반드시 필요하다. 본 논문에서는 메모리 관리 장치를 갖춘 리눅스 기반 임베디드 시스템에서 On-Chip SRAM인 스크래치패드 메모리를 이용하여 시스템 성능 향상과 에너지 소비 감소를 위한 메모리 계층 최적화 기법을 제안한다. 본 논문은 다음과 같이 구성되어 있다. 2장에서는 메모리 계층 최적화 기법에 사용되는 Marvell PXA310[10], ELF(Executable and Linking Format)[11], 메모리 매핑(Memory Mapping)과 같은 배경 지식을 살펴본다. 3장에서는 On-Chip SRAM를 이용한 임베디드 시스템 메모리 계층 최적화 기법을 제안하고, 4장에서 제안된 메모리 계층 최적화 기법을 실험을 통해 그 성능을 알아본다. 그리고 마지막 5장에서 결론을 맺는다.

2. 배경 지식

2.1 Marvell PXA310

On-Chip SRAM을 이용한 메모리 계층 최적화 기법이 적용될 임베디드 시스템 프로세서는 Marvell PXA310 프로세서다. Marvell PXA310 프로세서의 블록 다이어그램을 그림 1에 나타내었다. PXA310은 32KB 명령어 캐시(Instruction Cache)와 32KB 데이터 캐시(Data Cache)를 내장한 Intel XScale® Core를 사용한다. 그리고 스크래치패드 메모리로 사용 가능한 256KB Internal SRAM(이하 SRAM)이 존재한다. Core가 외부 메모리(External Memory, DDR SDRAM)와 통신하기 위해 메모리 스위치(Memory Switch), 동적 메모리 컨트롤러(Dynamic Memory Controller), 외부 메모리 핀 인터페이스(External Memory Pin Interface, EMPI)를 경유하는 반면, SRAM은 단지 메모리 스위치와 시스템 버스(System Bus)를 통해서 연결되어있다. SRAM 자체가 DRAM보다 빠르고 Core 가까이 위치하기 때문에 SRAM은 외부 메모리보다 Core 접근 속도

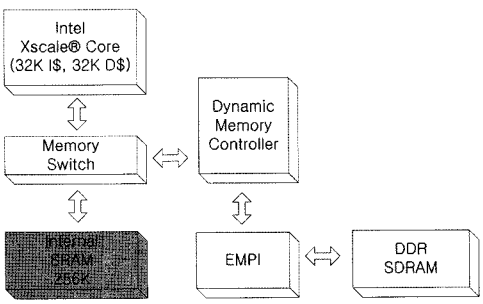


그림 1 Marvell PXA310 블록 다이어그램

표 1 외부 메모리와 내부 SRAM의 성능 차이

코어 클록(MHz)	지연 시간(cycles)	처리량(MB/sec)
104	17	205
208	21	328
312	30	343

코어 클록에 따른 외부 SDRAM 지연 시간과 처리량

코어 클록(MHz)	지연 시간(cycles)	처리량(MB/sec)
104	14	236
208	14	472
312	21	473

코어 클록에 따른 내부 SRAM 지연 시간과 처리량

가 빠르다. SRAM과 외부 메모리의 접근 지연 시간, 처리량을 표 1에 나타내었다. 표 1에서 보다시피 내부 SRAM은 외부 메모리에 비해 약 30% 더 나은 성능을 보인다. 하지만 여타 임베디드 시스템의 스크래치패드 메모리와는 달리 PXA310 프로세서의 SRAM에 할당된 코드나 데이터는 Core 내부의 캐시 메모리에 캐싱된다. 이러한 PXA310 프로세서의 SRAM 특성상 캐시 미스(Cache Miss)가 자주 발생하는 프로그램의 코드나 데이터를 SRAM에 할당해야 최적의 성능 향상을 기대할 수 있다.

PXA 시리즈 최적화 문서인 [12]에서는 On-Chip SRAM을 시스템 성능 향상을 위해 LCD 프레임 버퍼, 캡쳐 인터페이스 버퍼, 컨텍스트 스위치 버퍼, OS가 자주 사용하는 코드 배치 혹은 스크래치패드 메모리로 이용하라는 권고가 나와 있다. 하지만 현재까지 발표된 리눅스 커널 2.6.21까지 On-Chip SRAM을 이용하는 리눅스 모듈은 존재하지 않는다. 따라서 본 논문에서는 이전 리눅스 모듈과 아무런 충돌 없이 On-Chip SRAM을 스크래치패드 메모리로 활용할 수 있다.

2.2 ELF(Executable and Linking Format)

ELF(Executable and Linking Format)는 리눅스에서 지원하는 파일 포맷으로 기존 a.out 파일 포맷의 약점을 보완하고자 나온 파일 포맷이다. 기존의 a.out은 오직 세 가지의 세그먼트(segment)를 가질 수 있었다. 프로그램 코드가 포함된 .text, 전역 변수가 저장되어 있는 .data, 초기화되지 않은 전역 변수가 들어갈 .bss 세그먼트가 그것이다. 이러한 a.out의 약점을 보완하고자 등장한 것이 COFF(Common Object File Format)이다. COFF는 .text, .data, .bss 외의 섹션(section)을 추가할 수 있는 유연성을 제공하였다. ELF는 COFF와 매우 유사하지만 시스템 아키텍처에 투명하다는 장점이 있다. 이러한 ELF의 유연성과 이식성 때문에 리눅스, 솔라리스, FreeBSD, OpenBSD, IRIX같은 운영체제에서 ELF는 a.out과 COFF를 대체하였고, 현재 발표되는

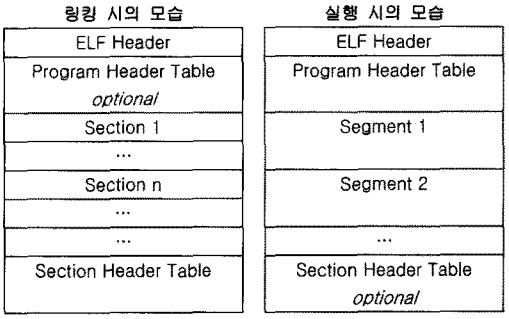


그림 2 ELF 파일 포맷

C 라이브러리와 GCC(GNU Compiler Collection)는 오직 ELF로만 링킹된다. ELF 파일 포맷은 링킹 시의 모습(Linking View)과 실행시의 모습(Execution View)이 다르다. 그림 2에 각각의 모습을 나타내었다. 그림 2에서 주목해야 할 차이점은 링킹 시의 모습에서는 ELF 파일이 섹션 단위로 나누어져 있고, 실행시의 모습에서는 여러 개의 섹션들로 구성된 세그먼트(Segment) 단위로 나누어져 있다는 점이다. ELF 파일 포맷은 섹션의 이름이 미리 정해져 있지 않지만 가장 보편적인 섹션 이름과 그 속성은 표 2와 같다. 섹션 중 bss 섹션은 초기화되지 않은 전역 변수(Global Variable)가 위치하는 곳이다. SHT_NOBITS라는 타입은 프로그램 이미지 파일에서 공간을 차지하지 않는다는 뜻이다. 즉 링킹 시의 모습에서는 해당 섹션에는 아무것도 존재하지 않는다는 의미이다. 대신 프로그램이 실행될 때, 즉 실행시의 모습에서 미리 정해진 크기로 확장되어 섹션 내부를 0으로 초기화한다. bss의 속성은 실행 시 메모리를 차지하고 있고(SHF_ALLOC), 메모리에 쓰기 가능하다(SHF_WRITE). data 섹션은 초기화된 전역 변수가 위치하는

표 2 ELF 포맷의 특별 섹션

이름	타입	속성
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.hash	SHT_HASH	SHF_ALLOC
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	SHF_ALLOC
.symtab	SHT_STRTAB	SHF_ALLOC
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR

곳이다. 타입이 SHT_PROGBITS이라는 것은 프로그램에서 정의된 섹션이라는 의미이다. 또한 bss 섹션과 동일하게 실행 시 메모리를 차지하고 있고, 쓰기 가능하다. 마지막으로 프로그램 코드가 저장되는 text 섹션이 있다. text 섹션은 bss와 data섹션과 달리 실행 가능하다(SHF_EXECINSTR). 프로그램 실행 시에는 bss 섹션과 data 섹션이 데이터 세그먼트(Data Segment)를 구성하고, text 섹션은 텍스트 세그먼트(Text Segment)를 이룬다. 데이터 세그먼트는 "읽기/쓰기 가능"한 속성을 가지고 있다. 하지만 쓰기 가능이라는 속성은 실행 가능이라는 의미를 포함하고 있기 때문에[11] 실제 데이터 세그먼트는 "읽기/쓰기/실행 가능"의 속성을 갖는다. 반면 텍스트 세그먼트는 "읽기/실행 가능"의 속성을 지닌다.

2.3 메모리 매핑(Memory Mapping)

리눅스는 가상 메모리(Virtual Memory)를 실현하기 위해 메모리 매핑(Memory Mapping)을 지원한다. 메모리 매핑이란 말은 크게 두 가지로 해석되는데, 물리 주소(Physical Address)를 가상 주소(Virtual Address)로 매핑해주는 것, 그리고 리눅스 파일 시스템에서 파일을 가상 주소로 매핑해 주는 것을 의미한다. 이번 절에서 말하는 메모리 매핑이란 후자의 파일 대 가상 주소의 메모리 매핑을 의미한다. 리눅스 사용자 공간(User Space)에서 메모리 매핑을 수행하기 위해서는 mmap 시스템 콜(System Call)을 사용해야 한다. mmap 시스템 콜의 모습은 그림 3과 같다. 그림 3은 파일 fd에서 off만큼 떨어진 곳부터 길이 len만큼의 공간을 가상 주소 start 번지로 매핑하겠다는 의미이다. Marvell PXA310의 SRAM은 물리 주소 0x5c000000 번지부터 0x5c040000 번지까지 총 256KB(0x40000B)만큼 물리 메모리 공간을 차지하고 있다. 리눅스에서 물리 메모리는 리눅스 파일 시스템의 /dev/mem 파일에 매핑되어있다. 그러므로 다음과 같은 mmap 시스템 콜을 통해 SRAM에 접근할 수 있다.

```
mmap(start, 0x40000, PROT_READ | PROT_WRITE |
PROT_EXEC, MAP_FIXED | MAP_SHARED,
"/dev/mem", 0x5c000000)
```

이와 같은 mmap 시스템 콜은 매핑할 SRAM 공간 속성이 읽기/쓰기/실행 가능하다는 의미다. 이러한 속성으로 인해 SRAM에 데이터와 코드를 모두 할당할 수 있다. 또한 해당 공간을 가상 주소 start로 설정하고 다른 프로세스와 공유 하겠다는 의미이다.

3. 메모리 계층 최적화 기법

On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화는 링킹 단계(Linking Phase)와 메모리 매핑 단계(Memory Mapping Phase)로 구성된다. 전체적인 모습을 그림 4에 나타내었다. 그림 4에서 SRAM에 할당될 코드나 데이터는 음영처리 하였다. 메모리 계층 최적화를 하기 위해 프로그래머는 그림 4와 같이 SRAM에 할당될 데이터(data2)나 코드(func2, func3)를 선택한다. 이러한 선택의 기준은 2.1절에서 설명했듯이 캐시 미스가 가장 많이 일어나는 것을 우선한다. 데이터와 코드 선택 후 링킹 단계를 거치면 선택한 데이터와 함수가 모두 sram 섹션으로 합쳐진 ELF 프로세스 이미지가 생성된다. 생성된 프로세스 이미지 실행 시 메모리 매핑 단계가 진행되는데, 이 단계에서 프로세스 이미지의 sram 섹션이 실제 PXA310 SRAM에 적재되고 SRAM에 매핑된다. 각 단계에 대한 자세한 설명은 다음과 같다.

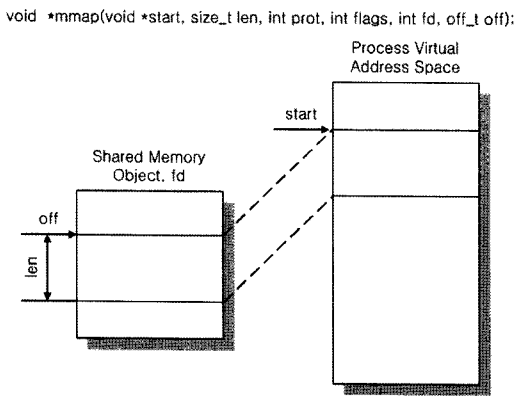


그림 3 mmap 시스템 콜

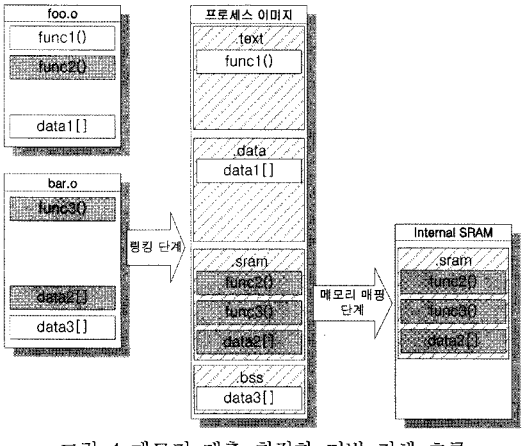


그림 4 메모리 계층 최적화 기법 전체 흐름

3.1 링킹 단계(Linking Phase)

링킹 단계에서는 원하는 데이터나 함수를 하나의 섹션(sram 섹션)으로 모으는 작업을 수행한다. 링킹 단계에서 가장 중요한 문제는 "sram 섹션을 ELF 이미지 어느 곳에 위치시키는가"이다. 2.2절에서 살펴봤듯이 ELF 파일 포맷의 대표적인 섹션은 bss, data, text가 있다. bss와 data 섹션은 데이터 세그먼트로 통합되고, text 섹션은 텍스트 세그먼트를 이룬다. 데이터 세그먼트는 읽기/쓰기/실행 가능 속성을 가지고 있는 반면, 텍스트 세그먼트는 읽기/실행 가능 속성을 지니고 있다. 본 논문에서 제안하는 최적화 기법은 데이터(읽기/쓰기)와 코드(읽기/실행)를 동시에 SRAM에 할당할 수 있어야 한다. 그러므로 읽기/쓰기/실행 가능 속성을 가진 데이터 세그먼트에 sram 섹션을 위치시켜야 한다. 이제 데이터 세그먼트 어느 부분에 sram 섹션을 위치시켜야 하는가가 문제가 된다. sram 섹션이 위치 가능한 곳은 data 섹션 앞, bss 섹션 뒤, data섹션과 bss 섹션 사이이다. 우선 data 섹션 앞에 위치시키고자 한다면 문제가 발생한다. 데이터 세그먼트에는 data 섹션과 bss 섹션 외에 ctors, dtors, dynamic, got 등의 섹션이 같이 통합되어있다. 문제는 ctors, dtors, dynamic, got 섹션과 data 섹션 사이에는 위치 순서가 존재하고, 서로 의존 관계가 있기 때문에 sram 섹션이 data 섹션 앞에 위치한다면 그 의존 관계가 깨지게 된다. 두 번째로 bss 섹션 뒤에 위치시키고자 한다면 이 역시 문제가 발생한다. ELF 이미지의 bss섹션의 타입은 SHT_NOBITS이다. 즉 섹션 크기가 0이다. 대신 ELF 이미지가 실행될 때 bss 섹션이 확장되고, 확장된 bss 섹션 뒤로 힙(Heap) 영역이 할당된다. 이 힙 영역은 ELF 이미지에 end라는 이름의 주소부터 시작한다. 만약 sram 섹션이 bss 섹션 뒤에 위치하게 된다면 ELF 이미지 생성시에 bss 섹션 크기가 0이기 때문에 sram 섹션과 end 위치가 겹쳐지게 된다. 이렇게 겹쳐진 상태로 ELF 이미지를 실행하게 된다면 sram 섹션과 힙 영역이 충돌하게 되어 프로그램이 오작동하거나 강제 종료된다. 마지막으로 sram 섹션이 위치할 수 있는 곳은 data 섹션과 bss 섹션 사이이다. ELF 이미지 포맷에 data 섹션과 bss 섹션 사이에 어떠한 의존 관계도 존재하지 않고, 특별한 주소 심볼도 없기 때문에 sram 섹션이 아무 문제 없이 위치할 수 있다. 이제 원하는 데이터나 코드를 해당 sram섹션에 할당해야 한다. 이 작업은 GCC(GNU Compiler Collection)가 제공하는 attribute 기능을 사용해야 한다. 사용 방법은 데이터(전역 변수)나 코드(함수) 정의에 __attribute__((section("section-name")))라는 속성을 부여하는 것으로 가능하다. 그리고 sram 섹션은 주기억장치인 외부 메모리에 할당되는 bss, data 섹션과 다르게 SRAM에 할당되기 때문에 메모리에서 데이터를 가져오는 단위인, 페이지(Page) 크기인 4KB로 정렬(align)해야 한다. 마지막으로 sram 섹션 시작 주소와 끝 주소에 각각 __sram_start와 __sram_end라는 전역 변수를 정의한다. 이 정의된 두 전역 변수는 메모리 매핑 단계에서 사용하게 된다. 링킹 시에 이와 같은 일을 처리하는 링커 스크립트를 그림 6에 나타내었다.

링커 스크립트	설명
.data :	data 섹션
...	...
. = ALIGN(0x1000);	4KB 정렬
__sram_start = .;	sram 섹션 시작 주소 변수 정의
.sram : {	sram 섹션 시작
*(.sram_text)	sram 텍스트 위치
*(.sram_data)	sram 데이터 위치
}	
__sram_end = .;	sram 섹션 마지막 주소 변수 정의
. = ALIGN(0x1000);	4KB 정렬
.bss :	bss 섹션
...	...
end = .;	힙 시작 주소

그림 6 링커 스크립트

3.2 메모리 매핑 단계 (Memory Mapping Phase)

메모리 매핑 단계는 링킹 단계에서 만들어진 ELF 이

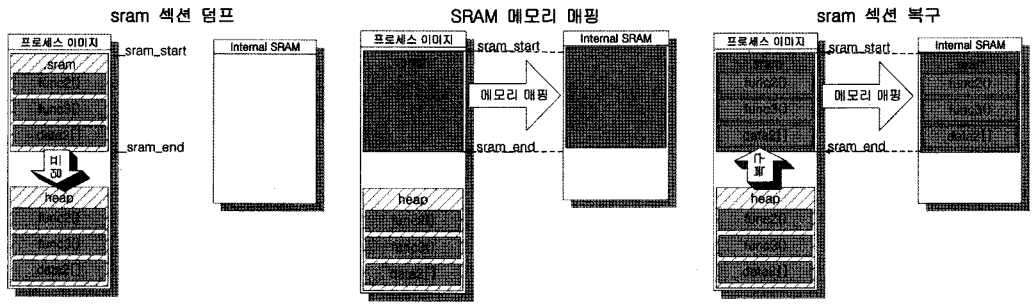


그림 5 메모리 매핑 단계

미지의 sram 섹션을 프로그램 실행 시 PXA310 SRAM에 적재하고 두 공간을 매핑하는 작업을 수행한다. 이 작업을 수행하는 것은 SRAM Allocator 모듈로, 타깃 어플리케이션과 같이 빌드된다. SRAM Allocator가 하는 일을 그림 5에 나타내었다. 우선 프로그램이 시작하기 전에, 즉, main 함수가 호출되기 전에 ELF 이미지의 sram 섹션 전체를 힙(heap) 영역으로 덤프(dump)한다. sram 섹션의 시작과 끝 주소는 링킹 단계 중 링커 스크립트에서 정의한 __sram_start와 __sram_end 전역 변수를 이용한다. 그리고 mmap 시스템 콜을 사용하여 SRAM를 sram 섹션에 메모리 매핑시킨다.

즉 mmap의 시작 주소 start는 __sram_start가 되고, 매핑할 크기 len은 __sram_end - __sram_start가 된다. mmap 시스템 콜이 끝나고 메모리 매핑된 프로세스의 가상 주소인 __sram_start부터 __sram_end까지 접근하게 되면, 그 주소에 해당하는 매핑된 공간, 즉 PXA310 SRAM에 접근하여 코드나 데이터를 가져오게 된다. 하지만 ELF 이미지의 sram 섹션을 PXA310 SRAM에 메모리 매핑을 하게 되면 ELF 이미지의 sram 섹션은 기존에 갖고 있던 데이터를 잃게 된다. 만약 sram 섹션의 데이터가 초기화되지 않은 전역 변수(.bss)라면 무조건 0으로 해당 공간을 채우면 아무런 문제가 없겠지만, 초기화 된 데이터(.data)가 들어가 있었거나 프로그램 코드(.text)가 있었다면 문제가 발생한다. 따라서 앞서 힙 영역에 덤프했던 기존의 sram 섹션 데이터를 다시 ELF 이미지의 sram 섹션에 복사하여 sram 섹션을 복구한다. 이러한 작업을 통해 기존의 ELF 이미지의 sram 섹션의 데이터를 PXA310 SRAM에 복사를 하게 되었다. 이제 메모리 매핑이 완료된 해당 프로세스가 자신의 sram 섹션에 접근하게 되면, 즉 sram 섹션에 있는 데이터를 읽고 쓰거나, sram 섹션에 있는 코드를 수행하게 되면 해당 프로세스에 투명하게 시스템의 SRAM에 매핑되어있는 데이터와 코드를 읽고 쓰고 실행하게 된다. 메모리 최적화가 적용된 프로세스는 적용되기 전과 동일하게 수행되지만, 실제로는 프로세스 모르게 SRAM을 이용하는 것이다. 끝으로 위와 같은 역할을 하는 SRAM Allocator의 함수들은 속성이 constructor로 설정되어 있어, 해당 프로세스가 GNU C Library를 사용할 경우 main 함수 호출 전에 해당 함수를 수행하게 한다. 또한 만약 프로그래머가 PXA310 SRAM의 크기인 256KB보다 더 많은 데이터나 코드에 SRAM 속성을 주어 sram 섹션의 크기가 256KB를 초과해도 단지 ELF 이미지의 sram 섹션의 시작부터 256KB 만이 메모리 매핑되기 때문에 아무런 문제도 발생하지 않게 된다. 이러한 유연성은 프로그래머의 실수로 인한 시스템의 오작동을 방지해준다.

4. 실 험

On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화 실험은 Marvell PXA310 프로세스를 사용한 시스템에서 수행 시간과 에너지 소비를 실측하였다. 실험 시스템 Marvell PXA310은 운영체제로 리눅스 커널 2.6.21을 사용하였다.

4.1 실험 어플리케이션

실험에 사용된 벤치마크 프로그램은 표 3과 같다. cachemiss는 본 실험을 위해 특별히 제작된 벤치마크로 데이터 캐시 미스를 최대한 많이 발생시키는 일을 수행한다. 캐시 미스가 발생하는 데이터를 SRAM에 할당하여 제안된 최적화 기법의 효과를 알아보았다. 그 외 matrixmul, heapsort, mergesort, quicksort[13]는 데이터를 할당하였고, cjpeg, djpeg, iirflt[14], gzip[15]은 데이터와 코드 모두를 할당하여 실험하였다.

표 3 실험 어플리케이션

이름	특징	SRAM 할당
cachemiss	데이터 캐시 미스를 최대한 발생시키는 어플리케이션	데이터
matrixmul	Matrix Multiplication	데이터
heapsort	Heap Sort	데이터
mergesort	Merge Sort	데이터
quicksort	Quick Sort	데이터
cjpeg	JPEG Encoder	데이터 & 코드
djpeg	JPEG Decoder	데이터 & 코드
iirflt	Low Pass Filter	데이터 & 코드
gzip	Compress / Decompress	데이터 & 코드

4.2 실험 세팅

실험 세팅은 그림 7과 같다. PXA310에 전원을 공급해주는 직류 전원 공급기로 Agilent E3634A[16]를 사용하였고 PXA310과 전원 공급기 사이에 저항 R을 직렬 연결하고 R에 흐르는 전압 Vr을 NI PCI-6259[17]로 1KHz 샘플링하여 PXA310에 흐르는 전력 P를 측정

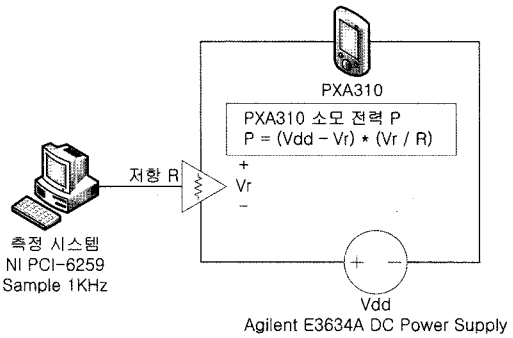


그림 7 실험 세팅

하였다. 최적화 기법을 적용했을 때 시스템이 소비하는 전력 감소를 정확하게 측정하기 위해 LCD의 밝기는 최소로 조정하고, USB, LAN, 시리얼 등의 외부 장치는 모두 제거한 후 실험하였다. 2.1절에 나와 있듯이 기존의 리눅스 커널은 On-Chip SRAM을 전혀 사용하지 않는다. 따라서 실험은 각 어플리케이션마다 On-Chip SRAM 최적화 기법을 사용하지 않았을 경우(ORIG)와 최적화 기법을 사용하였을 때(SRAM)의 수행 시간과 에너지 소비를 측정하였다.

4.3 실험 결과

실험 결과를 표준화(Normalize)하여 그림 8, 9에 각각 나타내었다. 수행 시간 측면에서는 최적화 기법을 사용하였을 때가 사용하지 않았을 때보다 최대 35%, 평균 14%의 향상을 보였다. cachemiss의 경우에는 최적화 기법을 적용했을 경우 약 35%의 성능 향상을 보였다. 이는 그림 1에서 보였듯이 PXA310 SRAM이 CPU 내부 캐시에 캐싱되기 때문이다. 최적화 기법을 사용하지 않았을 경우에 cachemiss 어플리케이션의 데이터는 SDRAM 같은 외부 메모리에 저장되어 있다. 그러므로 캐시 미스가 일어날 때 마다 외부 메모리 접근이 이루어진다. 이러한 데이터 캐시 미스는 CPU의 사이클을 낭비한다. 따라서 캐시 미스가 일어날 때 데이터를 가져오는 속도가 빠른 SRAM에 해당 데이터를 할당함으로써 성능 향상을 이룰 수 있는 것이다. matrixmul, heapsort, mergesort나 quicksort 역시 데이터 캐시 미스가 자주 일어나기 때문에 해당 데이터를 SRAM에 할당함으로써 성능 향상을 꾀할 수 있다. 반면 cjpeg이나 djpeg 같은 경우에는 캐시 미스가 일어나는 횟수가 매우 적다. 이러한 경우에는 아무리 접근이 잦은 데이터나 코드를 SRAM에 할당한다고 해도 큰 성능 향상을 기대할 수 없다. 이는 외부 메모리인 SDRAM에 있는 데이터든, SRAM에 있는 데이터든 CPU 내부 캐시에 캐싱되기 때문이다. 따라서 처음 캐시 미스가 일어났을 때를 제외하고는, 캐시 히트가 발생할 때에는 어떠한 성능 향상을 기대할 수 없다. gzip 같은 경우는 약 22%의 성능 향상을 나타내었다. gzip은 내부 구현에서 압축할 때와, 압축을 풀 때에 각각 32 킬로바이트의 버퍼를 사용한다. 프로파일링을 통해 이 버퍼가 캐시 미스를 자주 일으키는 데이터라는 것을 알 수 있었다. 따라서 이 두 버퍼를 SRAM에 할당하는 것만으로도 매우 큰 성능 향상을 이룰 수 있었다.

에너지 측면에서는 최적화 기법을 사용한 결과가 그렇지 않은 결과에 비해 최대 40%, 평균 15%의 에너지 소비 감소를 나타내었다. 그림 8과 그림 9의 모습을 비교해보면 매우 흡사하다는 것을 알 수 있는데, 이는 바로 수행 시간의 단축으로 인한 에너지 감소라는 것을 알 수 있다. 그리고 수행 시간의 단축에 비해서 에너지 소비가 약간 더 큰 것을 알 수 있다. 이는 외부 SDRAM보다 내부 SRAM이 소비하는 전력이 더 적기 때문에 생기는 효과이다.

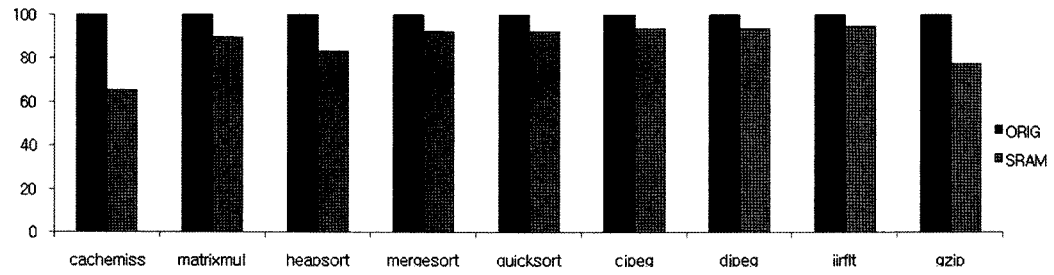


그림 8 수행 시간 실험 결과

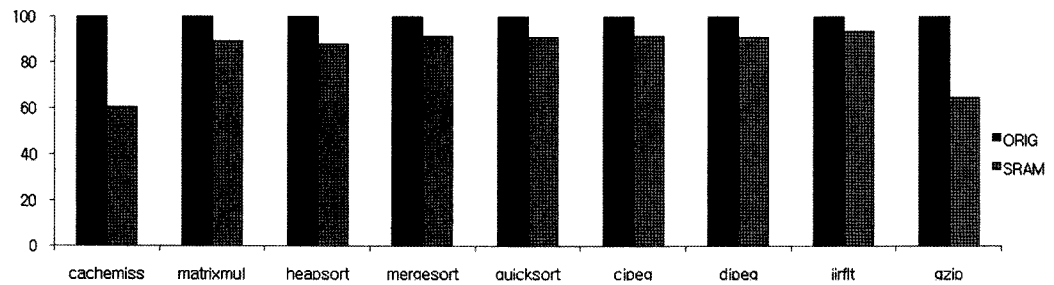


그림 9 에너지 소비 실험 결과

5. 결 론

본 논문이 제안한 On-Chip SRAM을 이용한 임베디드 시스템 메모리 계층 최적화 기법은 프로그램 소스상에서 코드나 데이터에 속성 추가와 SRAM Allocator와의 빌드만으로 시스템의 성능 향상과 에너지 소비의 감소를 이끌어 낼 수 있는 효과적인 방법이다. 또한 본 메모리 계층 최적화 기법은 리눅스 기반 임베디드 시스템에 적용한 최초의 스크래치패드 메모리 관리 기법으로 그 의의가 있다. Marvell PXA310에 메모리 계층 최적화 기법을 적용하여 실측하였으며, 총 9개의 어플리케이션에 대하여 시스템의 성능을 최대 35%, 평균 14% 향상시켰고, 에너지 소비를 최대 40%, 평균 15% 감소시켰다.

참 고 문 헌

[1] John Hennessy and David Patterson. Computer Architecture A Quantitative Approach. Morgan Kaufmann, Third edition, 2002.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad Memory: A Design Alternative for Cache Onchip memory in Embedded Systems. In Proceedings of the 10th International Workshop on Hardware/Software Codesign, pages 73-78, May 2002.

[3] Oren Avissar, Rajeev Barua, and Dave Stewart. An Optimal Memory Allocation Scheme for Scratch-Pad Based Embedded Systems. ACM Transactions on Embedded Computing Systems, pages 6-26, November 2002.

[4] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using postpass optimization. In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems, pages 223-233, October 2006.

[5] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management for portable systems with a memory management unit. In Proceedings of the 6th ACM & IEEE International conference on Embedded software, pages 321-330, October 2006.

[6] Hyeongmin Cho, Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic Data Scratchpad Memory Management for a Memory Subsystem with an MMU. In Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems, pages 196-206, June 2007.

[7] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic scratchpad memory management for code in portable systems with an MMU. ACM Transactions on Embedded Computing Systems, Volume 7, Issue 2, pages 1-38, 2008.

[8] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management in a multitasking environment. In Proceedings of the 7th ACM international conference on Embedded software, pages 265-274, October 2008.

[9] Nghi Nguyen, Angel Dominguez, and Rajeev Barua. Scratch-pad memory allocation without compiler support for java applications. In Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, pages 85-94, October 2007.

[10] Marvell, Marvell® PXA30x Processor and PXA31x Processor, Volume II: Memory Controller Configuration Developers Manual, Doc. No. MV-S301038-02, Rev. B. 2007.

[11] TIS Committee, Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, 1995.

[12] Intel, Intel® PXA27x Processor Family Optimization Guide, 2004.

[13] Mark Allen Weiss. Data Structures and Algorithm Analysis in C, Addison-Wesley, 1997.

[14] EEMBC - The Embedded Microprocessor Benchmark Consortium, http://eembc.org.

[15] gzip, http://www.gzip.org.

[16] Agilent Technologies, Agilent E3634A, http://www.home.agilent.com/agilent/product.jspx?nid=-35690.384003.00.

[17] National Instruments, NI PCI-6259, http://sine.ni.com/nips/cds/view/p/lang/en/nid/14128

김 정 원

2006년 서울대학교 컴퓨터공학부 학사

2006년~현재 서울대학교 컴퓨터공학부 석박사통합과정. 관심분야는 Compiler Optimization, High Performance Computing Systems

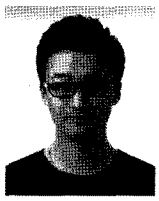
김 승 균

2003년 서울대학교 컴퓨터공학부 학사

2004년~현재 서울대학교 컴퓨터공학부 석박사통합과정. 관심분야는 Compiler Optimization, Programming Language Design, Fast Pure Object Oriented Programming Environment

이 재 진

1991년 서울대학교 물리학과 학사. 1995년 Stanford University, Computer Science 석사. 1999년 University of Illinois at Urbana-Champaign, Computer Science 박사. 2000년~2002년 Michigan State University, Computer Science and Engineering 조교수. 2002년~2004년 서울대학교 컴퓨터공학부 조교수. 2004~현재 서울대학교 컴퓨터공학부 부교수. 관심분야는 Compilers, Computer Architectures, High Performance Computing Systems, Embedded Systems



정 창 희

2003년 충남대학교 컴퓨터공학과 학사 2005년 서울대학교 컴퓨터공학부 석사 2005년~2008년 한국전자통신연구원 연구원. 2008년~현재 Georgia Institute of Technology, Computer Science 박사과정. 관심분야는 Compilers and Computer Architecture for High-Performance and Embedded Systems

우 덕 균

1993년 홍익대학교 컴퓨터공학과 학사 1995년 홍익대학교 전자계산학과 석사 2001년 홍익대학교 전자계산학과 박사 2001년~현재 한국전자통신연구원 센서네트워크플랫폼연구팀 선임연구원. 관심분야는 Compiler, Embedded Software Development Tool, Sensor Network Simulation