simpleRTJ 임베디드 자바가상기계의 에너지 사용 분석†

양희재
경성대학교 컴퓨터공학과
hjyang@star.ks.ac.kr

Analysis of Energy Usage in simpleRTJ Embedded Java Virtual Machine

Heejae Yang
Dept. of Computer Engineering, Kyungsung University

요 약

휴대폰이나 PDA 등 무선 이동 장치에 내장되는 자바가상기계는 필요 에너지를 모두 밧데리에서 공급받는다. 밧데리의 빈번한 재충전이나 교체 등에 따른 불편을 줄이기 위해서는 임베디드 자바가상기계의 에너지 사용을 최소화하는 것이 매우 중요하다. 자바가상기계의 에너지 사용은 자바 클래스의 저장에 따른 정적 에너지 사용과 바이트코드 실행 및 쓰레기 수집기의 작동 등에 따른 동적 에너지 사용으로 나눌 수 있다. 본 논문에서는 simpleRTJ 상용 임베디드 자바가상기계의 에너지 사용에 대해 분석해보았다. simpleRTJ 의 주요 특징인 프리레졸루션과 ROM 이미지 형식이 에너지 사용에 미친 영향에 대해 분석하였으며, 고정 크기 메모리 할당과 에너지 소비의 상호 관계에 대해서 고찰하였다.

1. 서 론

휴대폰이나 PDA 와 같은 무선 이동 장치에 자바 기술을 적용하는 사례가 급증하고 있다. 대표적 휴대폰 제조업체인 노키아는 연간 1억개 이상 규모의 자바 기반 휴대폰을 단독으로 출시하고 있다.

무선 이동 장치에 자바 기술을 도입하는 가장 큰 이유는 자바가 가진 플랫폼 독립성이다. 각기 다른 하드웨어 규격과 운영체제를 갖는 이동 장치를 위해서는 한번 작성된 프로그램을 임의의 환경에서 사용할 수 있는 (*Write Once, Run Anywhere*) 플랫폼 독립성이 무엇보다 필요하다. 그외에도 자바가 가진 안전성, 수월성, 동적 프로그램 적재 능력 등이 무선 이동 장치에 자바를 적용하게 하는 주요 이유들이다.

무선 이동 장치에서 가장 중요한 요소 중 하나는 에너지 사용이다. 이들 장치는 필요한 모든 에너지를 내장된 밧데리로부터 제공받고 있으며, 빈번한 재충전이나 밧데리 교체에 따른 불편을 줄이기 위해서는 에너지 사용을 최소화할 필요가 있다.

불행하게도 자바 환경은 기존의 C 등 네이티브 언어를 사용하는 환경에 비해 에너지 사용면에서 불리하다 [1]. 자바가상기계에서 자바의 명령어인 바이트코드는 명령어가 아니라 데이터로 취급되어져서 메모리에서 읽혀지게 되며, 이 명령어들은 자바가상기계에 의해 인터프리트되거나 JIT 컴파일 된 후 실행이 이루어진다. 이런 과정은 모두 추가적 에너지를 필요로 한다. 또한 객체의 생성이나 해제, 쓰레기 수집 등의 과정에서 메모리를 빈번히 접근하게 되며, 메모리 접근에 따라 대량의 에너지를 소비하게 된다.

본 논문에서는 임베디드 자바 시스템을 위한 자바가상기계 중 하나인 simpleRTJ 의 에너지 사용에 대해 고찰해 본다. simpleRTJ 는 8/16/32 비트 프로세서 등을 대상으로 한 소규모의 임베디드 자바가상기계이다 [2]. 이것의 중요한 특징 중 하나는 프로그램 실행에 필요한 클래스 파일들을 호스트 컴퓨터상에서 미리 레졸루션 (pre-resolution) 하여 클래스 파일 자체가 아니라 프로그램 실행에 적합한 ROM 이미지로 변형하고, 이 이미지를 자바가상기계에서 읽어 실행하도록 하는 것이다. 레졸루션 과정이 자바가상기계에서 일어나지 않으므로 낮은 성능의 프로세서에서도 빠른 실행이 이루어지며, 적은 양의 메모리로도 실행이 가능하다.

simpleRTJ 의 또다른 큰 특징은 단일 고정 크기 메모리 할당을 한다는 점이다. 임베디드 시스템은 실시간성을 갖는 것이 중요한데, 일반적으로 가변 크기 메모리 할당에 비해 고정 크기 메모리 할당이 실시간성을 갖추도록 하는데 더욱 유리하다. simpleRTJ 에서는 모든 객체마다 동일 크기의 힙 메모리를 할당하도록 하여 메모리 할당 알고리즘을 간단히 하였고, 쓰레기 수집기 (GC) 알고리즘도 빠르게 실행될 수 있도록 하고 있다.

본 논문에서는 이런 특징을 갖는 임베디드 자바가상기계를 에너지 사용면에서 분석해보고자 한다. simpleRTJ 의 특징은 에너지 사용면에서도 여타 자바가상기계와 구별되는 성질을 갖는다. 예상되는 주요 특징은 다음과 같다.

- 레졸루션이 호스트 컴퓨터 상에서 일어나므로 자바가상기계 내에서는 레졸루션에 따른 에너지 소비가 없다.
- ROM 이미지 내에는 레졸루션에 필요한 심볼 등이 없으므로 이미지의 크기는 클래스 파일 크기에 비해 매우 작다. 따라서 이것을 저장할 메모리 양이 작아도 되므로 클래스 메모리 크기를 줄일 수 있고, 따라서 에너지 사용도 줄일 수 있다.
- 메모리 할당 알고리즘이 간단하고, 빠른 GC 동작 등으로 인해 에너지 소비를 줄일 수 있다.
- 반면 고정크기 메모리 할당에 의해 내부 단편화 현

† 본 연구는 과학기술부 목적기초연구(R05-2004-000-10967-0) 지원으로 수행되었음.

상이 일어나게 되며, 이것에 의한 힙 메모리 낭비가 발생하여 필요 이상의 힙 메모리를 유지하기 위한 에너지 낭비가 있다.

본 논문에서는 위의 특징을 고려하여 simpleRTJ 의 에너지 사용에 대해 분석해보았다. 2절에서는 에너지 사용에 대한 관련 연구에 대해 알아보며, 3절에서 simpleRTJ 자바가상기계의 정적 에너지 사용에 대해 분석해 본다. 4절에서는 simpleRTJ 의 동적 에너지 사용에 대해 클래스 메모리, 힙 메모리 등으로 나누어 고찰해본다. 결론 및 향후 연구 방향에 대해 5절에 설명하였다.

2. 관련 연구

일반적인 데스크톱 컴퓨터와 마찬가지로 임베디드 시스템도 프로세서와 메모리, 입출력 장치 등으로 구성된다. 기존 연구에 따르면 이들 중 메모리가 가장 큰 몫의 에너지를 소비하는 것으로 밝혀지고 있다 [3]. 메모리의 소비 에너지는 메모리를 읽거나 쓸 때 소요되는 동적 에너지(E_d)와 메모리 접근에 관계없이 누설전류 등에 의해 소요되는 정적 에너지(E_s)로 구분할 수 있는데, 대개 동적 에너지가 정적 에너지에 비해 월등히 크다. 예를 들어 삼성전자의 휴대 장비용 DRAM 제품 중 K4S561633-1H 경우 대기전류는 6mA 이지만, 접근시 130mA 의 전류가 흐른다 [4]. 따라서 2.7V 입력전압, 100MHz 의 속도에서 E_s 는 10 pJ/bit 이지만, E_d 는 220 pJ/bit 에 해당된다. 즉 에너지 소비를 줄이기 위해서는 무엇보다 동적 에너지를 줄여야 하며, 그렇게 하기 위해서는 메모리에 대한 접근을 최소화할 필요가 있다는 것을 알 수 있다.

자바가상기계의 메모리는 클래스에 대한 정보를 담고 있는 클래스 영역과, 각종 객체들을 저장하는 힙 영역, 그리고 메소드 실행 시 필요한 오퍼랜드 스택과 지역변수 배열 등을 담는 자바 스택 영역으로 나뉘어진다 [5]. 클래스 영역의 정보는 한번 적재되면 변하지 않으며, simpleRTJ 에서는 ROM 이미지의 내용이 이 부분에 해당된다. 힙 영역은 객체의 생성에 따라 그 크기가 계속 커지게 되며, 가용 메모리가 고갈되거나 일정 수준 이하로 떨어지면 자동적으로 GC가 작동하여 더 이상 필요 없는 객체의 메모리를 회수한다. 자바 스택 영역은 메소드 호출 및 복귀 때마다 스택 프레임이 만들어졌다가 사라진다. 본 논문에서는 클래스 영역을 저장하는 메모리를 클래스 메모리, 힙 영역과 자바 스택 영역을 저장하는 메모리를 힙 메모리로 부르기로 한다.

3. 정적 에너지 사용 분석

클래스에 대한 모든 정보는 클래스 영역 메모리에 놓여진다. simpleRTJ 에서는 클래스 파일 내용 자체가 아니라 레졸루션이 완료된 ROM 이미지 형태가 클래스 영역에 놓여지는데, ROM 이미지의 크기는 평균적으로 원래 클래스 파일의 48% 에 불과하다 [2].

저장할 클래스 영역의 크기가 작다면 이것은 곧 에너지 절감을 이룰 수 있는 기회가 된다. Chen [6] 은 메모리 블럭 구조를 설명하면서 누설전류에 의한 정적 에너지 손실을 줄이기 위해 자바가상기계의 코드와 API 클래스 라이브러리를 압축된 형태로 주기억장치에 두는 방식을 제안했다. simpleRTJ 에서는 프리레졸루션 및 고유의 콤팩트한 ROM 이미지 구조에 따라 압축된 것과 같은 효과를 거둘 수 있다. 실제로 압축된 것은 아니므로 Chen 의 접근 방법과 달리 실행 시 감압(decompress)에 따른 에너지 소비도 없게 된다. [2] 의 결론에 따르면 주요 API 클래스 라이브러리에 대해 simpleRTJ 는 평균 48% 절감된 정적 에너지를 사용한다.

4. 동적 에너지 사용 분석

여기서는 메모리 접근에 따른 에너지 사용, 즉 동적 에너지 사용에 대해 분석해본다. 2절의 설명과 같이 자바가상기계에서 메모리 접근은 클래스 메모리에 대한 접근과 힙 메모리에 대한 접근으로 나눌 수 있다.

4.1 클래스 메모리 접근 분석

클래스 영역의 주요 내용은 클래스 정보, 상수 정보, 필드 정보, 그리고 메소드 정보로 나뉘어진다. 그림 1은 simpleRTJ 의 클래스 영역 구조를 보여주고 있다. 여기서 상수풀 오프셋 테이블(constant pool offset table)은 4 바이트 크기를 갖는 포인터 값을 엔트리로 갖는 배열 구조를 가지며, 각 포인터는 클래스, 상수, 필드, 메소드 구조체 중 어느 하나를 가리키고 있다 [2].

클래스 정보에 대한 접근은 `new #i` 라는 바이트코드 명령의 실행에 의해 이루어진다. 여기서 `i`는 상수풀 오프셋 테이블의 인덱스 값이다. 클래스 정보를 읽는 과정은 ① 인덱스 값으로 상수풀 오프셋 테이블 내용 읽기 (4바이트); ② 상수풀 오프셋 테이블이 가리키는 번지에서 `class_T` 구조체 읽기 (32바이트)등 두 단계를 거친다. 1비트 접근에 따른 동적 에너지를 E_d 라고 한다면, new 명령 실행에 소요되는 동적 에너지는 288 E_d 가 된다.

상수 정보에 대한 접근은 `ldc #i` 라는 바이트코드 명령의 실행에 의해 이루어진다. 마찬가지로 `i`는 상수풀 오프셋 테이블의 인덱스 값이다. 상수 정보를 읽는 과정은 ① 인덱스 값으로 상수풀 오프셋 테이블 내용 읽기 (4바이트); ② 상수풀 오프셋 테이블이 가리키는 번지에서 `const_T` 구조체 읽기 (4바이트); ③ 상수의 형식과 길이를 알고 실제 데이터 읽기 등 세 단계를 거친다. 상수

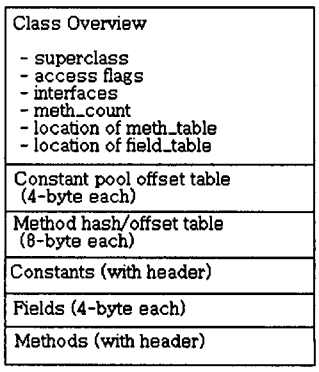

그림 1 simpleRTJ 클래스 영역 구조

의 길이를 cl 바이트라고 하면 상수 정보 접근에 따른 에너지는 $8E_d(8+cl)$ 이 된다. 임베디드 자바 API 클래스에서 사용되는 상수는 대개 정수형임을 감안하면 [2] cl = 4 이며, 따라서 평균 소요 에너지는 $96E_d$ 가 된다.

필드 정보에 대한 접근은 getfield/putfield 등의 명령에 의해 이루어지며, 형식은 getfield #i 등과 같다. 필드 접근에 따른 과정은 ① 인덱스 값으로 상수풀 오프셋 테이블 내용 읽기 (4바이트); ② 상수풀 오프셋 테이블이 가리키는 번지에서 field_T 구조체 읽기 (4바이트); ③ 힙 메모리 상에 놓인 해당 필드 접근 등 세 단계를 거친다. 여기서 ③ 번의 단계는 힙 영역에 대한 접근이므로 나중에 따로 고려하도록 하면 필드 접근에 따른 에너지는 $64E_d$ 가 된다.

마지막으로 메소드 정보에 대한 접근은 invokevirtual #i 등의 명령에 의해 이루어진다. 메소드 접근에 따른 과정은 ① 인덱스 값으로 상수풀 오프셋 테이블 내용 읽기 (4바이트); ② 상수풀 오프셋 테이블이 가리키는 번지에서 method_T 구조체 읽기 (16바이트); ③ 메소드 내의 바이트코드 읽기 등 세 단계를 거친다. 여기서 ③ 번 단계는 바이트코드 개수가 메소드마다 각기 다르며, 반복문이나 조건문 등의 사용 여하에 따라 다르므로 고려에서 제외하도록 한다. 이 경우 메소드 접근에 따른 에너지는 $160E_d$ 가 된다.

4.2 힙 메모리 접근 분석

바이트코드가 실행되는 단계에서 힙 메모리에 대한 접근이 이루어진다. JIT 컴파일 방식이 아니라 인터프리트 방식을 사용하는 simpleRTJ에서는 인스턴스를 생성하는 new 명령에 대해 필드들을 저장할 공간과 ref_T 구조체를 저장할 공간을 각각 확보하게 되는데, 이것은 힙 메모리 상의 빈 공간을 찾아 선형 검색하는 과정을 따른다. 빈 공간이 발견되면 그것을 할당하고, 그렇지 않으면 GC를 작동하고 다시 선형 검색을 시도한다. 이 과정에 소요되는 평균 에너지를 예측하는 것은 현재까지 이루어지지 않았다. 이 선형 검색 과정은 simpleRTJ 의 실시간성을 해칠 뿐 아니라 에너지 소비 예측도 어렵게 만드는 부분 중 하나다.

상수를 읽는 ldc 명령은 클래스 영역에 놓인 상수를 오퍼랜드 스택으로 옮기며, 스택 접근에 따른 에너지 소비는 4바이트 기준으로 $32E_d$ 가 된다.

필드 접근 명령인 getfield/putfield 등 명령은 실제 필드들이 힙 영역에 존재하므로 추가 에너지 소비가 필요하며, 임베디드 시스템의 경우 거의 모든 필드들이 32비트 크기임을 감안할 때 $32E_d$ 의 에너지가 소요된다.

메소드 접근 명령인 invokevirtual 등 명령은 힙 메모리에 자바 스택을 생성한다. 자바 스택은 오퍼랜드 스택, 지역변수 배열, 프로그램 카운터와 스택 포인터 등을 포함하는 frame_T 구조체로써 이중연결 구조를 이루고 있다. 자바 스택의 할당은 new 명령 실행 때와 마찬가지로 선형 검색을 통해 이루어지며, 메모리 고갈시 GC 가 작동된다. 이것에 소요되는 에너지 분석은 아직 이루어지지 않았다.

표 1 클래스 주요 정보 접근에 소요되는 에너지

접근 정보	바이트코드	소요 E_d 량			이상상태 소비율
		클래스	힙	계	
클래스	new	288	α	288+α	3.3
상수	ldc	96	32	128	1.3
필드	getfield / putfield	64	32	96	1.0
메소드	invokevirtual	160	β	160+β	2.0

표 1에 클래스와 관련된 주요 정보들을 접근하는데 필요한 에너지를 정리하였다. 여기서 α와 β는 각각 인스턴스와 ref_T 공간을 할당받는데 소요되는 에너지와 자바 스택을 할당받는데 소요되는 에너지를 의미한다. 언제나 힙 메모리의 여유가 있다면, 즉 힙 메모리의 크기가 무한대인 이상상태를 가정한다면 α와 β 는 최소값을 가지며, $\alpha_{min} = \beta_{min} = 32E_d$ 이다. 필드 접근에 따른 에너지 소비량을 1.0 이라 한다면 상수 접근은 1.3, 메소드와 클래스 접근은 이상 상태에서 각각 2.0, 3.3 이며 실제로는 더 큰 값을 필요로 한다.

5. 결론 및 향후 연구 방향

simpleRTJ 는 클래스 파일 대신 프리레졸루션이 끝난 ROM 이미지를 클래스 영역으로 사용하므로 클래스 파일을 직접 저장하는 것보다 평균 48%의 정적 에너지 절감을 얻을 수 있었다. 상수와 필드 접근에 대해서는 동적 에너지의 소비가 일정하지만 클래스와 메소드에 대한 접근은 동적 메모리 할당에 따라 가변적이다. 이상적 상태를 가정한다면 클래스, 상수, 필드, 메소드 접근에 대해 각각 3.3, 1.3, 1.0, 2.0 의 비율로 동적 에너지 소비가 일어난다. 향후 동적 메모리 할당 및 GC 에 따른 에너지 소비를 정량적으로 분석하고자 한다.

참 고 문 헌

[1] N. Vijaykrishnan, et al., "Energy Behavior of Java Applications from the Memory Perspective", *JVM '01*, April 2001

[2] 양희재, "simpleRTJ 임베디드 자바가상기계의 ROMizer 분석 연구", 한국정보처리학회 논문지, 제 10-A권, 4호, 2003년 10월, pp.397-404

[3] N. Vijaykrishnan, et al., "Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower", *Proc. Int'l Symp on Computer Architecture*, June 2000

[4] J. Fryman, et al., "Energy-Efficient Network Memory for Ubiquitous Devices", *IEEE Micro*, Sept-Oct 2003, pp.60-70

[5] T. Lindholm and F. Yellin, *The Java Virual Machine Specification*, Addison-Wesley, 1997

[6] G. Chen, et al., "Energy-Savings Through Compression in Embedded Java Environments", *10th Int'l Symp on Hardware/Software Co-design*, May 2002