휴대장치에서 바이너리 코드를 효율적으로 복원하기 위한 압축 기법

이현철, 김강석, 예홍진
아주대학교 대학원 지식정보보안학과
e-mail : deletenim@ajou.ac.kr, kangskim@ajou.ac.kr, hjyeh@ajou.ac.kr

FastD : A Compression Approach for an Efficient Binary Code Decompression in Mobile Devices

Hyunchul Lee, Kangseok Kim, Hongjin Yeh
Dept. of Knowledge Information Security, Graduate School of Ajou University

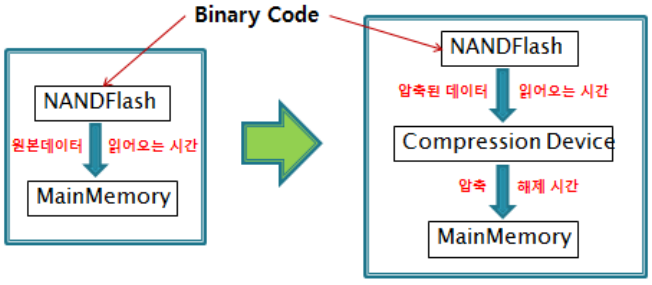
요 약

프로그램 코드는 실행이 되기 전에 반드시 주 기억 장치에 Loading 되어야 하는데, 이때 Loading Time 은 압축 데이터를 NAND Flash Memory 로부터 읽어오는 시간과 압축을 복원하는 시간의 합이 된다. 따라서 빠른 압축 복원 속도는 코드 압축을 사용하는 임베디드 장치에서는 매우 중요한 요소가 된다. 일반적으로 휴대 장치의 경우 일반 PC 와는 달리 적은 배터리 용량 및 프로세서의 한계, 프로그램을 저장하는 NAND Flash Memory 의 크기 때문에 최적의 성능을 발휘할 수 없었다.

본 논문에서는 무 손실 압축 알고리즘에 대한 연구를 진행 함과 동시에 모바일 환경에 적합한 LZCode 을 개선하여 복원속도를 기존 LZCode 보다 1.5 배 향상 시키는 알고리즘을 제시 하고자 한다.

1. 서론

휴대전화와 같은 임베디드 장치에서는 원가 절감을 위해서 NAND Flash Memory[1]를 사용한다. 또한 프로그램 코드를 압축하여 NAND Flash Memory 에 저장함으로써 공간의 절약과 부팅시간의 단축을 기대할 수 있다. 일반적인 임베디드 장치의 경우 NAND Flash Memory 에서 Main Memory 로 데이터를 읽어 오는 시간이 부팅시간에 가장 큰 영향을 준다. 그 과정은 많은 시간을 소요한다. 한마디로 NAND 에서 읽어 오는 시간이 곧 부팅 시간이 된다.



(그림 1) 휴대장치의 Binary Code 접근 방식

그림 1 에서 말하는 Binary Code 는 일반적인 시스템 데이터를 의미하며 이러한 Binary Code 들이 Main Memory 에 올라가 실행되기도 하며, 중요한 데이터로 활용 된다.

압축기법을 사용하는 임베디드 장치의 경우 NAND Flash Memory 에서 Main Memory 로 읽어 올 때 중간에서 압축기가 작동하여 NAND Flash Memory 에서 읽거나 쓸 때 압축 및 복원 루틴이 작동하여 사용하게 된다. 압축기를 사용할 경우 NAND Flash Memory 의 공간을 좀더 효율적으로 사용할 수 있게 된다. 부팅 속도는 압축된 데이터를 NAND Flash Memory 에서 읽어 오는 시간과 압축기가 그 데이터를 복원하는 시간의 합이 된다. 결론적으로 원본 데이터를 그대로 읽어 오는 시간과 압축된 데이터를 읽어 오는 시간에서 복원하는 시간의 합을 비교하여 좀더 효율적인 기법을 사용하는 것이다. 그럼으로 효과적으로 메모리를 사용하면서 빠른 부팅 속도를 가지기 위해서는 최적의 압축 기법을 선택하고 사용하게 되는 것이다.

일단 압축 기법을 사용하는 임베디드 장치는 압축 데이터를 저장 하기 때문에 공간을 효율적으로 사용할 수 있고, 복원 속도에 따라 부팅 속도가 결정 되기 때문에 부팅 속도는 압축기에 의해 최적화된 속도를 얻을 수 있다. 이것은 곧 복원 속도가 향상될수록 일반적인 임베디드 장치보다 더 좋은 효과를 낼 수 있게 된다는 것이다. 그래서 본 논문에서는 임베디드 장치에서의 최적화된 사전식 압축 기법을 제안 하고자 한다. 압축 방식은 기존의 LZ 형식[2]을 활용하여

만든 LZCode[3] 에서 변형된 방법을 사용한다.

본 논문은 이 압축 방법을 복원 속도 향상에 중점을 두었기 때문에 FastD(Fast Decompression)라고 부르도록 하겠다.

FastD 는 기존의 LZ 형식의 기반으로 만들어진 LZCode 의 특징인 베타 논리합(exclusive OR) 값을 이용 하며, 복원 속도를 높이기 위해 LZCode 에서 사용하고 있는 Table 및 비교연산을 제거하고 그 제거된 데이터의 상응 하는 알고리즘으로 대체하여 압축 속도를 높인 기법이다.

2. 관련연구

2.1 Zlib

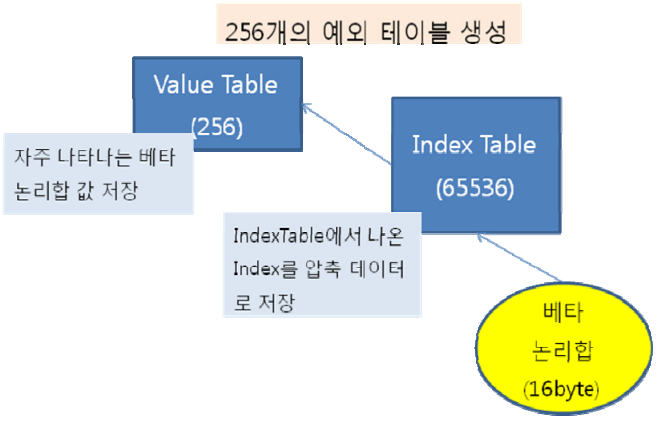
Zlib[4]는 LZ77[2] 알고리즘과 Huffman[6] 알고리즘을 접목시킨 deflate[7] 알고리즘을 사용하는 압축 기법으로 사전식 압축 기법을 이용하여 압축을 시도한 후에 비교문자를 찾기 못한 특정 문자열들에 대해 Huffman coding 압축기법을 사용하여 2 중 압축 효과를 보는 압축 기법이다.

2.2 LZCode

LZCode 는 현재 압축하고자 하는 데이터가 참조된 데이터와 같지 않을 경우 프로그램 코드의 명령어 특성을 이용하여 두 데이터의 베타 논리합 값을 저장하는 방식을 사용한다.

가장 빈번하게 나타나는 예외 값에 대해서 Table 을 만들고 Table 안의 데이터가 예외 값으로 나타나게 될 경우 Table 을 참조하여 압축 및 복원을 한다. 결국 16bit 에서 8bit 로 변경되면서 8bit 을 절약할 수 있게 된다. 또한 Table 을 이용하기 때문에 특별한 연산 시간 없이 데이터에 접근할 수 있다는 장점이 있다.

실제 LZCode 는 압축기법 중에 일반적으로 쓰이는 Zlib 와 비교했을 때 최대 5 배의 압축 복원 속도와 4%의 압축률 향상이 있으며 LZCode 압축 기법을 모바일에 적용했을 경우 부팅 시간이 10~20% 단축된다.[3]



(그림 2) LZCode Table 사용 방식

Table 사용 루틴은 다음과 같다.

1. 16bit 의 베타 논리합 값을 구한다.
2. 베타 논리합 값을 Index Table 의 첨자로 넣고 Index 값을 확인한다. (Index Table[베타 논리합 값])
3. Index 값이 0 이 아닐 경우 그 값을(8bit) 저장하게 된다.
4. 값을 복원하는 방법은 Value Table 에 Index 값을 첨자로 넣게 되면 해당 16bit 값이 나오게 된다. (Value Table[Index])

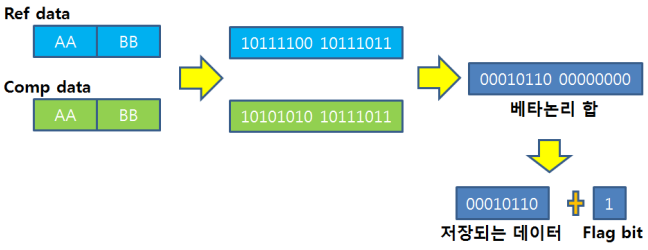
3. 제안된 연구(FastD)

3.1 FastD 압축 기법

FastD 는 기본적으로 동적 사전식 압축 기법을 사용하는 LZCode 의 변형이라고 볼 수 있다. LZCode 의 특징 중 하나는 베타 논리합 값을 이용하여 데이터를 저장하는데 이 데이터들 중에서 자주 나오는 값들을 Table 에 저장 하여 사용한다. 또한 비교 단위가 16bit 단위로 비교 하고 있기 때문에 베타 논리합의 값도 당연히 16bit 가 된다. 그 16bit 을 Table 에 저장 하고 그 인덱스 값을 가지고 있기 때문에 실제로 데이터에 저장되는 메모리 공간은 8bit 가 되는 것이다. 하지만 FastD 같은 경우는 Table 을 사용하지 않고 베타 논리합 값 16bit 중에서 0 이 아닌 값이 있는 8bit 만을 저장하여 사용한다. 절차는 다음과 같다.

1. 16bit 의 베타 논리합 값을 구한다.
2. 베타 논리합 값을 8bit 으로 나눈 후 한쪽 8bit 가 0 인경우를 찾는다.
3. 0 으로 채워진 8bit 은 제거 되고 남은 8bit 을 Flag bit 와 함께 저장한다.
4. 앞쪽 8bit 경우는 Flag bit 1 을 저장하고 뒤쪽에 있을 경우는 0 을 저장하게 된다.

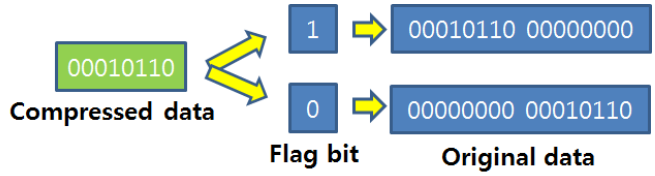
이런 식의 루틴을 따르게 되면 Table 에 접근하는 연산과정이 줄어들고 Table 이 가지고 있던 메모리 공간도 절약하게 된다.



(그림 3) FastD 압축 기법

베타 논리합 값 중에 00010110 을 압축 데이터로 저장 하고 flag bit 을 두어 앞쪽 8bit 인지 뒤쪽 8bit 인지를 구별할 수 있게 된다. 이와 같은 방법을 사용할 경우 Table 을 이용하여 16bit 값을 8bit 로 만든 효과와 비슷한 효과를 볼 수 가 있게 된다. 또한 명령어 코드가 4byte 나 2byte 시스템을 사용하는 바이너리

코드 같은 경우 2byte 나 1byte 는 명령코드로 자주 사용 되기 때문에 반은 같은 경우가 많이 발생하게 된다. 그런 점을 고려 봤을 때 압축률은 Table 을 사용했을 때와 많은 차이가 나지는 않는다. 그 대신 Table 을 사용하는 공간과 연산의 사용을 줄일 수가 있으므로 복원 속도의 향상을 기대할 수 있게 된다.



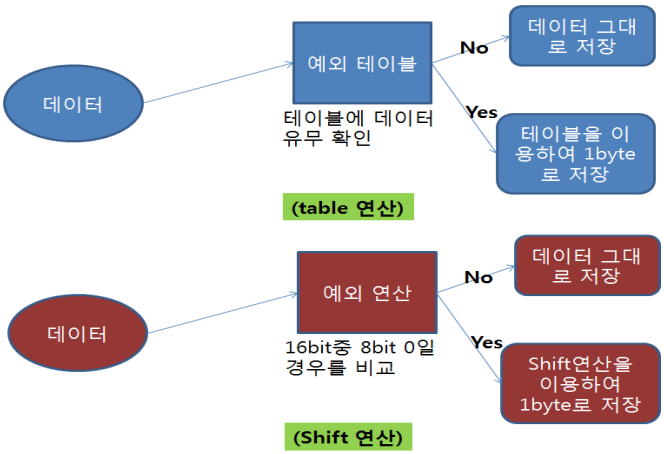
(그림 4) FastD 복원 기법

3.2 관련 연구와 비교 분석

	Zlib	LZCode	FastD
비교단위	Byte	Short	Short
저장공간	단일버퍼	단위별 버퍼	단위별 버퍼
비트연산	Bit 단위접근	Byte 단위접근	Byte 단위접근
부분매칭	매칭 사용	부분 매칭 사용	부분 매칭 사용
Align	미사용	사용	사용
예외처리	허프만코딩	테이블 사용	Shift 연산 사용

<표 1> Zlib, LZCode, FastD 특징 비교 표

FastD 와 LZCode 는 많은 부분에서 같은 것을 볼 수 있다. 그것은 일단 LZCode 을 기반으로 해서 FastD 가 만들어 졌기 때문이며, FastD 와 LZCode 의 다른 점은 예외처리 부분이며, 예외처리 방식은 Table 처리 방식을 Shift 연산으로 변경하여 예외를 처리함으로써 LZCode 보다 빠른 복원 속도와 압축기의 무게를 줄일 수가 있다.



(그림 5) 알고리즘 비교

4. 실험결과

실험 환경은 Microsoft Windows XP Professional 의 운영체제 기반으로 진행 되었으며, 컴퓨터의 CPU 및 RAM 의 사양은 Intel® Core(TM) i3 CPU 540 @3.07 4GB 이다.

시간 측정 부분에서는 ms(Millisecond) 단위로 측정을 하기 위해 Win32 API 에서 제공하는 QueryPerformanceFrequency(), QueryPerformanceCounter() 함수를 이용 하여 시간을 측정 하였다. 이 두 함수는 CPU 의 Tick Count 를 이용해 경과 시간을 알아낼 수 있어 좀더 정확한 실험을 할 수 있었다.

원본 데이터		Zlib			LZCode			FastD		
데이터이름	원본 크기	속도	압축크기	압축률	속도	압축크기	압축률	속도	압축크기	압축률
amss_ver1.000(1MB)	1,048,576	0.021	374,948	64%	0.003	429,056	59%	0.002	433,664	59%
amss_ver1.006(1MB)	1,048,576	0.026	504,339	52%	0.006	611,840	42%	0.004	619,008	41%
amss_ver1.013(1MB)	1,048,576	0.029	661,666	37%	0.005	654,848	38%	0.004	663,552	37%
amss_ver1.020(1MB)	1,048,576	0.024	517,645	51%	0.005	643,584	39%	0.003	648,704	38%
amss_ver1.027(1MB)	1,048,576	0.016	194,966	81%	0.002	284,160	73%	0.002	286,720	73%
amss_ver1.034(1MB)	1,048,576	0.029	681,663	35%	0.005	688,128	34%	0.004	697,344	33%
amss_ver1.041(1MB)	1,048,576	0.028	632,740	40%	0.005	620,032	41%	0.004	629,760	40%
amss_ver1.048(1MB)	1,048,576	0.028	626,778	40%	0.006	635,904	39%	0.004	644,608	39%
amss_ver1.055(1MB)	1,048,576	0.025	624,087	40%	0.003	731,136	30%	0.002	733,184	30%
amss_ver1.062(1MB)	1,048,576	0.019	339,220	68%	0.003	443,392	58%	0.002	444,416	58%
평균	1,048,576	0.025	515,805	51%	0.004	574,208	45%	0.003	580,096	45%

<표 2> 복원속도 및 압축률 실험 결과 표

실험 결과 표를 보게 되면 FastD 가 LZCode 에 비해 압축률이 1% 정도 감소 하였지만 복원속도에서 평균적으로 1.5 배 정도 빠른 것을 확인 할 수가 있다. 또한 Zlib 의 복원 속도와 비교해 보았을 경우 7 배 정도 빠르며, 데이터의 양이 커질 수록 복원 속도와 압축률이 Zlib 에 비해 더 좋은 성능을 내는 것을 확인 할 수 있었다.

5. 결론 및 향후 과제

LZCode 는 베타의 논리합을 이용하여 압축률의 향상과 복원 속도의 향상을 가져왔다. 본 논문에서는 LZCode 에서의 복원 속도를 높이는 다른 방안을 제시하였고 압축률은 조금 떨어지지만 그 압축률에 상응하는 복원 속도를 올릴 수 있었다.

결국 FastD 와 LZCode 을 비교했을 때 서로의 장단점이 있으므로 상황에 맞게 알맞은 압축기법을 사용한다면 좀더 효과적인 메모리 사용을 할 수 있을 것이다.

현재는 PC 상에서의 실험 결과로 FastD 의 복원속도가 LZCode 에 비해 향상된 것을 확인 할 수 있다.

앞으로는 PC 가 아닌 실제 휴대장치가 사용하는 ARM CPU 환경에서 실험 하고 분석 및 연구를 함으로써 현재 FastD 의 복원 속도보다 좀더 빠른 복원 속도를 구현 및 연구가 필요가 있다.

현재 ARM9 에뮬레이터에서 FastD 에 대한 실험 및 연구가 진행 중이며, 개선할 점을 찾아 현재의 복원 속도보다 더 빠른 복원속도를 구현하는데 목적을 두고 있다.

앞으로 높은 복원 속도가 필요한 곳은 Flash Memory 를 쓰는 곳 일 것이다. 현재 많은 스마트폰의 영향으로 많은 이슈가 생겨나고 있는데 그 중 가장 이슈가 되고 있는 것은 FOTA Update[8]이다. 외국에서는 예전부터 많은 관심을 가지고 연구가 되고 있지만 국내에서는 많은 관심을 같지 않은 분야이다. 이 기법은 무선으로 NAND Flash Memory 에 있는 데이터를 업데이트 시켜주는 방식을 말한다. 때문에 무선으로 업데이트를 하기 위해서는 쓰기 위한 빠른 압축 속도와 읽어 오기 위한 빠른 복원 속도를 필요로 한다. 이런

점을 생각해 봤을 때 복원 속도와 압축 속도 두 가지 모두를 높일 수 있는 대안이 필요하다고 생각된다.

또한 현재 사용되는 스마트 폰에서 전력의 사용을 줄이기 위해 Hibernation[9] 기법을 사용하고 있는데 이 Hibernation 에서도 압축된 이미지를 NAND 에 저장하여 전력을 아끼는 방법이 나오고 있다 이런 방법들이 적절하게 사용되기 위해서는 압축률을 높이고 복원 속도를 빠르게 하는 기법들이 많이 등장해야 할 것이다.

참고문헌

[1]Hyojin Kim, Youjip Won, Yohwan Kim. “MNFS : Design of Mobile Multimedia File System based on NAND FLASH MEMORY Memory”, 정보과학회논문지 : 시스템 및 이론 제 35 권 제 11 · 12 호 497~579, 2008.12.

[2]E.-h. Yang and J. C. Kieffer, “On the redundancy of the fixed database Lempel-Ziv algorithm for mixing sources”, IEEE Trans. Inform. Theory, vol.43, pp.1101-1111, July 1997.

[3]Y. Kim, Y. Wee, "A Program Code Compression Method with Very Fast Decoding for Mobile Devices," Journal of KIISE : Software and Applications, vol.37, no.10, pp.851-858, November 2010.

[4]Zlib, “http://www.zlib.net/”, 2010.

[6]Hyuncheol Kim, Jaekyoung Ryu, Sunny Jung, Jinwoo k Jung “The Concatenated Algorithm of Compression an d Secrecy Using Huffman Codes”, 한국정보과학회 : 1991 년 도 봄 학술발표논문집 제 18 권 제 1 호, pp 3~568, 1991.4.

[7]P.Deutsch, “DEFLATE Compressed Data Format Specification version 1.3”, In http://www.ietf.org.IETF RFC1951, March 1996.

[8]Taehwa Kim, Youngcheul Wee, “A New Code Compression Method for Compressed Firmware Over the Air on Mobile Devices”, IEEE Transactions on Consumer Electronics, Vol. 56, No. 4, November 2010.

[9]Hibernation,“https://wiki.archlinux.org/index.php/Suspending_to_Disk_with_hibernate-script”, 2011.

[10]A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture”, Proc. 25th Ann. International Symposium on Micro architecture, pp.8 1-91, December 1992.