휴대장치를 위한 고속복원의 프로그램 코드 압축기법

(A Program Code Compression Method with Very Fast Decoding for Mobile Devices)

김 용 관 † 위 영 철 ††
(Yongkwan Kim) (Youngcheul Wee)

요 약 대부분의 휴대기기는 보조 기억장치로 NAND flash 메모리를 사용하고 있다. 또한, firmware의 크기를 줄이고 NAND flash로부터 주기억장치로 로딩하는 시간을 줄이기 위해서 압축된 코드를 NAND flash에 저장한다. 특히, 압축된 코드는 매우 빠르게 해제가 되어야 demand paging이 적용 가능하게 된다. 본 논문에서는 이를 위하여 새로운 사전식 압축 알고리즘을 제안한다. 이 압축방식은 기존의 LZ형식과는 다르게 현재 압축하고자 하는 명령어(instruction)가 참조된 명령어와 같지 않을 경우, 프로그램 코드의 명령어의 특성을 이용하여 두 명령어의 배타 논리합(exclusive or) 값을 저장하는 방식이다. 또한, 압축 해제 속도를 빠르게 하기 위해서, 비트 단위의 연산을 최소화한 압축형식을 제공한다. 실험결과 zlib과 비교해서 최대 5배의 압축해제 속도와 4%의 압축률 향상이 있었으며, 이와 같이 매우 빠른 압축해제 속도에 따라 부팅(booting) 시간이 10~20% 단축되었다.

키워드 : 코드 압축, 낸드 플래쉬, 휴대기기, 디맨드 페이징

Abstract Most mobile devices use a NAND flash memory as their secondary memory. A compressed code of the firmware is stored in the NAND flash memory of mobile devices in order to reduce the size and the loading time of the firmware from the NAND flash memory to a main memory. In order to use a demand paging properly, a compressed code should be decompressed very quickly. The thesis introduces a new dictionary based compression algorithm for the fast decompression. The introduced compression algorithm uses a different method with the current LZ method by storing the "exclusive or" value of the two instructions when the instruction for compression is not equal to the referenced instruction. Therefore, the thesis introduces a new compression format that minimizes the bit operation in order to improve the speed of decompression. The experimental results show that the decoding time is reduced up to 5 times and the compression ratio is improved up to 4% compared to the zlib. Moreover, the proposed compression method with the fast decoding time leads to 10-20% speed up of booting time compared to the booting time of the uncompressed method.

Key words : Code Compression, NAND flash, Mobile devices, Demand Paging

· 본 연구는 2008학년도 아주대학교 일반연구비 지원에 의하여 연구되었습니다.

† 비 회 원 : 아주대학교 정보통신대학원 컴퓨터공학과
karnies@ajou.ac.kr
†† 종신회원 : 아주대학교 정보 및 컴퓨터공학부 교수
ycwee@ajou.ac.kr

논문접수 : 2010년 8월 17일
심사완료 : 2010년 9월 11일

정보과학회논문지: 소프트웨어 및 응용 제37권 제11호(2010.11)

1. 서 론

휴대전화와 같은 임베디드(embedded) 장치에서는 원가절감을 위해서 프로그램 코드(program code)를 압축하여 낸드 플래쉬(NAND flash)에 저장한다(그림 1 참조). 또한, 주기억장치(DRAM)의 사용량을 줄이기 위하여 디맨드 페이징(demand paging)을 적용한다(그림 2 참조). 프로그램 코드는 실행되기 전에 반드시 주기억장치에 로딩(loading) 되어야 하는데, 이때의 로딩 시간은 압축 페이지(page)를 낸드 플래쉬로부터 읽어오는 시간과 압축을 해제(decoding)하는 시간의 합이 된다. 따라서 빠른 압축해제 속도는 코드 압축을 사용하는 휴대전화와

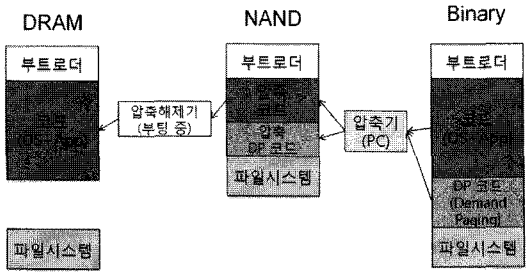

그림 1 압축 코드 처리과정

같은 임베디드 장치에서는 매우 중요한 요소가 된다.

코드 압축기법은 크게 두 가지로 분류할 수 있는데, 하나는 캐쉬(cache)에 코드를 로딩하기 직전에 압축을 해제하는 방식이고[1-6], 다른 하나는 코드 세도윙(shadowing) 과정에서 압축을 해제하는 방식이다[7-12]. 여기서, 세도윙이란 프로그램 코드를 실행하기 전에, 낸드 플래쉬와 같은 비 휘발성 메모리로부터 DRAM과 같은 휘발성 메모리에 복사해 놓는 과정을 말한다.

제안하는 방법은 세도윙 과정에서 채택될 수 있는 압축방법으로, 사전식 압축기법인 LZ형식[7]의 변형된 방법이다. LZ형식 중 하나인 zlib에서는 압축을 해제하는데 소요되는 시간중의 대부분이 비트 단위의 데이터를 해독하는 연산이다. 제안하는 방법은 비트 단위의 연산을 최소화 하여 zlib의 압축방식에 비해 최대 5배 빠르게 압축해제를 수행한다. 또한, 명령어의 특성을 활용하여 zlib의 압축방식에 비해 최대 4%의 압축률을 향상시킨다.

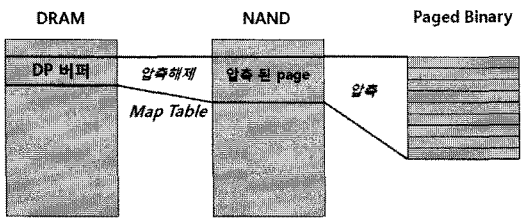

그림 2 압축 디맨드 페이징 처리과정

본 논문은 다음과 같이 구성된다. 2장에서는 제안하는 코드 압축기법을 설명하고, 3장에서는 제안된 방식과 zlib의 압축을 비교 실험하여 성능과 결과를 분석한다. 4장에서는 결론이 주어진다.

2. 제안하는 코드 압축방법

2.1 개요

제안하는 압축방법은 사전식(dictionary based) 방법인 LZ형식[7]의 변형 된 방법으로, LZcode라 부른다. 임베디드 장치에서 코드 바이너리를 압축하여 사용하기 위해서는, 매우 빠른 압축해제 속도와 높은 압축률이 필요하다. LZcode는 압축해제 속도를 향상시키기 위해서, 비트 단위의 연산을 최소화 한다. 그리고 이로 인한 압축률의 감소를 보완하기 위해 2바이트(Thumb mode) 또는 4바이트(ARM mode)로 구성되는 명령어의 특성을 활용하여 압축률을 향상시킨다. LZcode에서 zlib 대비 압축률을 개선하는 기본원리는 그림 3과 같다.

zlib에서 그림 3의 예제에 대한 압축정보는 (12, 8), (literal B), (12, 3)가 되고 이를 호프만(Huffman) 코딩 방식을 사용하여 엔트로피 코딩(entropy coding)을 한다. 코드 (12, 8)의 12는 현재 압축하고자 하는 시퀀스인 22번 위치로부터 참조가 되는 시퀀스인 10번 위치로의 거리를 나타내고, 8은 일치되는 시퀀스의 길이를 나타낸다. 즉, 시퀀스 (22~29)가 시퀀스 (10~17)와 일치한다는 것을 의미한다. 그리고 (literal B)는 3개 이상 일치되지 않는 데이터 이다. 코드 (12, 3)은 시퀀스 (31~33)가 시퀀스 (19~21)와 일치함을 의미한다.

LZcode에서는 기본 압축단위가 2바이트 이므로 그림 3의 예제가 그림 4와 같은 형태가 된다. zlib의 기본 압축단위는 바이트 이지만, LZcode는 기본 압축단위로 2바이트를 사용한다. 이는 휴대기기의 프로그램 코드가 대부분 2바이트 또는 4바이트의 명령어이기 때문이다.

참조 되는 시퀀스 (10~21):

10	11	12	13	14	15	16	17	18	19	20	21
A	B	A	B	A	C	A	B	A	B	C	A

압축 하고자 하는 시퀀스 (22~33):

22	23	24	25	26	27	28	29	30	31	32	33
A	B	A	B	A	C	A	B	B	B	C	A

zlib 압축 형태

(12, 8), (literal B), (12, 3)

그림 3 zlib의 압축예제

참조 되는 시퀀스 (10~15):

10	11	12	13	14	15
AB	AB	AC	AB	AB	CA

압축 하고자 하는 시퀀스 (16~21):

16	17	18	19	20	21
AB	AB	AC	AB	BB	CA

LZcode 압축 형태

(6, 6), (AB ⊕ BB)

그림 4 LZcode에서의 압축예제

코드 (6, 6)는 시퀀스 (16, 21)가 참조되는 시퀀스 (10, 15)와 6개의 2바이트 거리를 갖고, 6개의 2바이트가 부분매칭이 된다는 것을 나타낸다. LZcode에서는 zlib과는 달리 시퀀스가 완전히 일치가 되지 않아도 일치 되지 않는 위치의 데이터와 참조되는 위치의 데이터 간의 차이 값을 저장하는 방법을 사용하여 하나의 거리 값으로 부분매칭을 유지시킨다. 그림 4에서 보면 20번 위치의 데이터 BB와 참조되는 14번 위치의 데이터 AB가 일치하지 않는다. 따라서 차이 값 (AB ⊕ BB)을 저장하는데 그 위치는 압축하고자 하는 시퀀스에 있는 각 명령어에 플래그를 주는 방법을 사용하여 저장한다. 결국 LZcode는 zlib에 비해서 더 적은 수의 거리정보를 저장하게 되고 길이 값 대신 플래그를 저장한다.

LZcode에서 최적의 부분매칭을 찾는 것은 NP-hard의 문제인, Longest common subsequence problem [13]보다 더 큰 시간 복잡도를 갖기 때문에 압축 시간이 아주 크게 소요된다. 이를 극복하기 위해 최적 값의 근사치를 찾는 방법을 2.4절에서 제시한다.

zlib은 엔트로피 코딩으로 호프만 코딩을 사용한다. 호프만 코딩은 많은 비트연산을 포함하므로 압축해제 속도가 느려지는 주된 원인이 된다. 다음 절에서는 압축해제 속도를 높이기 위하여 비트연산을 최소화하는 기법을 소개한다.

2.2 압축해제 속도의 향상

zlib의 압축방식에서는 거리정보와 길이정보, 그리고 예외 값(literal)을 저장할 때, 호프만 코딩을 사용하기 때문에, 압축된 형식이 가변길이의 비트단위가 된다. 반면에 LZcode에서는 고정된 길이인 4바이트 마다 거리정보를 갖고, 일치가 되는 길이정보는 갖지 않는다. 또한, 거리정보 역시 바이트 또는 쇼트(2 바이트)로 저장되고, 예외 값은 엔트로피 코딩 없이 그대로 저장한다. 따라서, 비트연산이 최소화되어 압축해제 속도가 빨라지게 된다. 다음은 zlib과 LZcode의 압축형태에 대한 비교예시이다.

참조 되는 시퀀스 (0~15):

0				4				8				12			
A	B	A	B	A	C	A	B	A	B	C	A	B	C	A	C

압축 하고자 하는 시퀀스 (16~47):

16				20				24				28			
A	B	A	K	A	B	C	B	B	C	A	C	A	B	C	A
32				36				40				44			
B	C	A	C	A	B	A	K	A	B	C	B	B	C	A	C

Zlib의 압축형태

(14, 3), (literal K), (9, 3), (literal B), (12, 6), (20, 18)

그림 3 zlib의 format

그림 3에서 압축하고자 하는 데이터의 시퀀스를 zlib으로 압축하면 위와 같은 형태가 된다. 코드 (14, 3)은 시퀀스 (16~18)가 시퀀스 (2~4)와 일치함을, 코드 (9, 3)은 시퀀스 (20~22)가 시퀀스 (11~13)와 일치함을, 코드 (12, 6)은 시퀀스 (24~29)가 시퀀스 (12~17)와 일치함을, 코드 (20, 18)은 시퀀스 (30~47)이 시퀀스 (10~27)와 일치함을 나타낸다.

LZcode에서는 기본 압축단위가 2 바이트이므로 그림 3의 zlib 압축형태가 그림 4와 같은 형태가 된다.

참조 되는 시퀀스 (0~7):

0	1	2	3	4	5	6	7
AB	AB	AC	AB	AB	CA	BC	AC

압축 하고자 하는 시퀀스 (8~23):

8	9	10	11	12	13	14	15
AB	AK	AB	CB	BC	AC	AB	CA
16	17	18	19	20	21	22	23
BC	AC	AB	AK	AB	CB	BC	AC

LZcode의 압축형태

거리 정보: 5, 7, 6, 10, 10, 10, 10, 10
복사 플래그: 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
예외 플래그: 1, 0
차이 값: AB ⊕ CB
예외 값(exception): AK

그림 4 LZcode의 format

거리정보 5는 매크로블록 (8~9)과 시퀀스 (3~4)의 부분매칭을, 거리정보 7은 매크로 블록 (10~11)과 시퀀스 (3~4)의 부분매칭을, 거리정보 6은 매크로블록 (12~13)과 시퀀스 (6~7)이 일치함을, 거리정보 10들은 나머지 매크로블록 (14~23)들이 시퀀스 (4~13)과 일치함을 나타낸다. LZcode는 압축의 기본단위로 2 바이트를 사용하여 2바이트의 명령어 마다 참조하는 명령어와 같은지를 나타내는 플래그를 저장하게 된다. 거리정보 역시 2 바이트 단위로 증감한다. 그리고 4바이트 마다 거리정보를 갖는다. 이 4바이트(2개의 명령어)를 매크로블록(Macro Block)이라 한다.

2.2.1 거리 정보

거리정보를 4바이트 마다 모두 저장하게 되면 압축률의 심각한 저하를 불러온다. 따라서, 바로 앞의 매크로블록과 거리정보가 같으면 거리정보 플래그(dist flag)만 1로 설정하고 거리정보는 저장하지 않고, 같지 않으면 거리정보 플래그를 0으로 설정하고 거리정보를 저장한다. 이 결과, 거리정보 플래그가 일치하는 시퀀스의 길이를 대체하게 된다. 이를 그림 4의 예제에 적용하면 다음과 같이 된다.

```
거리정보 플래그: 0, 0, 0, 0, 1, 1, 1, 1
거리 정보: 5, 7, 6, 10
복사 플래그: 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
예외 플래그: 1, 0
차이 값: AB ⊕ CB
예외 값(exception): AK
```

2.2.2 차이 값

명령어 w를 압축하고자 할 때, 참조하는 명령어 r이 w와 같지 않으면 배타 논리합 연산 값인 w ⊕ r 을 차이 값으로써 저장하게 되는데, 이 값을 diff(w,r)로 표현한다. 차이 값을 구할 때, 배타 논리합을 사용하는 이유는 프로그램 코드를 구성하는 명령어들이 연산자는 같지만 피 연산자가 다르거나 연산자는 다르지만 피 연산자가 같은 경우가 많이 발생하기 때문에, 이들 명령어를 배타 논리합으로 계산하게 되면 유사한 패턴의 비트 배열이 많이 발생하기 때문이다. 발생 가능한 차이 값들 중에서 가장 많은 빈도수를 가지는 차이 값을 선택하여 테이블을 구성하고, 이 테이블에 diff(w,r)이 존재하면 해당 인덱스를 저장하고 존재하지 않으면 예외(exception)로 처리한다.

일반적인 엔트로피 코딩의 경우 빈도수가 높을수록 적은 비트수의 코드워드를 사용하지만, LZcode에서는 빠른 압축해제 속도를 위해서 테이블 엔트리의 수는 256개로 하고 인덱스는 한 바이트의 동일한 길이로 한다. 여기서 동일한 길이로 엔트로피 코딩을 하여도 압축률이 크게 저하되지 않는데, 이는 앞서 설명한 바와 같이 연산자 (operator)는 같고 피 연산자 (operand)가 다르거나, 피 연산자는 같고 연산자가 다른 명령어들이 많이 발생할 수 있는 프로그램 코드의 특성 때문이다. 반면에 바이트 단위로 테이블에 접근이 가능하기 때문에, 압축 해제 속도를 빠르게 만든다. 이를 2.2.1의 코드에 적용하면 다음과 같이 된다.

```
거리정보 플래그: 0, 0, 0, 0, 1, 1, 1, 1
거리 정보: 5, 7, 6, 10
복사 플래그: 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
예외 플래그: 1, 0
차이 값: idx1, where table[idx1] = diff(AB, CB)
예외 값(exception): AK
```

2.2.3 블록 복사

2.2.2절의 예제에서, 8개의 복사 플래그는 비트연산을 최소화 하는 동시에 압축률을 높이기 위해서 하나의 바이트로 묶이게 된다. 만약 8개의 복사 플래그가 모두 1이고, 4개의 거리정보 플래그 역시 모두 1이라면, 블록복사 플래그(block copy flag)값을 1로 설정하게 되고, 해당 부분의 복사 플래그와 거리정보 플래그는 저장하지 않는다. 여기에서 블록은 4개의 매크로블록을 의미한다. 위의 예제에 블록복사 플래그를 적용하면 다음과 같다.

```
블록복사 플래그: 0, 1
거리정보 플래그: 0, 0, 0, 0
거리 정보: 5, 7, 6, 10
복사 플래그: 1, 0, 1, 0, 1, 1, 1, 1
예외 플래그: 1, 0
차이 값: idx1, where table[idx1] = diff(AB, CB)
예외 값(exception) : AK
```

다음은 위의 블록복사 플래그가 적용된 압축코드를 해제하는 슈도 코드(pseudo code)이다. 여기서 하나의 블록은 8개의 명령어(2바이트)로 이루어져 있기 때문에, 8개의 복사 플래그가 하나의 바이트로 표현이 가능하게 되어 비트연산을 최소화 한다.

```
if (block_copy) block = ref_block;
else {
  mask = (10000000)2;
  for (int i=0; i<8; i++){
    if (copy_flags & mask) block[i] = ref_block[i];
    else if (exception_flag) block[i] = GetValue(expt_list);
    else block[i] = ref_block ⊕ diff_table[GetIdx(idx_list)];
    mask = mask>>1;
  }
}
```

실험 결과 LZcode의 압축해제 속도가 zlib의 압축 방식에 비해 디맨드 페이징을 위한 4K 단위의 압축에서 최대 5.16배 빠른 것을 볼 수 있다(표 2 참조).

2.3 압축률 향상

zlib의 압축방식에서는 압축하는 블록과 참조되는 블록이 일치하는 경우에만 거리정보를 가진다. 반면에 LZcode는 압축하고자 하는 블록과 참조되는 블록의 일정부분이 다르더라도 하나의 거리정보를 가질 수 있다. LZcode에서는 이 블록을 부분매칭 블록(partial matching block)이라 한다. 부분매칭 블록을 zlib에서 압축하고자 한다면, 시퀀스가 일치되는 부분과 일치가 되지 않는 부분이 각기 따로 압축되어야 하고, 저장되어야 하는 거리정보는 최소한 두 개 이상이다. LZcode에서는 현재까지 압축이 되었던 매크로블록들의 거리정보를 현재의 거리정보로 수정하여서 줄어드는 비트수가 최대가 되는 거리정보로 통일시키는 방식을 적용하여, 하나의 거리정보를 가지게 된다. 이를 후방 정련(backward refinement)이라 한다. 후방 정련을 2.2.3절의 예제에 적용하면 다음과 같은 형태가 된다.

```
블록복사 플래그: 0, 1
거리정보 플래그: 0, 1, 0, 0
거리 정보: 5, 6, 10
복사 플래그: 1, 0, 1, 0, 1, 1, 1, 1
예외 플래그: 1, 0
차이 값: idx2, where table[idx2] = diff(CA, CB)
예외 값(exception): AK
```

2.2.3절의 예제에서 두 번째 매크로블록의 거리정보 7을 세 번째 매크로블록의 거리정보 6으로 대체시켰을 경우, diff(CA, CB) 역시 차이 값 테이블에 존재하면, 거리정보 하나를 저장하지 않아도 되므로 코딩 코스트(cost)가 줄어든다. 따라서, 두 번째 거리정보 플래그를 0에 1로 바꾸고 거리정보 7은 삭제한다.

2.4 압축 속도 향상

2.1절에서 언급되었던 것처럼 LZcode에서 가장 적은 정보량을 만들어 내는 부분매칭을 찾는 것은 NP-hard의 문제이기 때문에 압축시간이 아주 크게 소요된다. 따라서 최적 값이 아닌 근사값을 구해 압축속도를 향상시키는 방법을 사용한다.

2.4.1 해쉉

zlib에서는 현재 압축하고자 하는 데이터와 가장 길게 일치가 되는 시퀀스를 찾기 위한 방법으로 윈도우의 모든 위치를 검색하는 것이 아니라, 3바이트의 조합을 키(key)로 사용하는 해쉉기법을 사용한다. LZcode도 이와 유사한 해쉉을 적용하는데 압축의 기본단위는 2바이트이기 때문에, 2바이트의 명령어를 key로 사용하는 해쉉기법을 사용한다. 결국 참조되는 윈도우의 모든 시퀀스를 검색하는 것이 아니라, 해쉉을 통해 선택된 후보군 중에서 최소 정보량을 만들어 내는 부분매칭을 찾는다.

2.4.2 부분매칭 검색

zlib에서는 해쉉을 통해 검색된 후보 군 중에서 매칭이 가장 길게 일어나는 위치로의 거리정보를 저장한다. 그러나 LZcode에서는 비록 중간에 일치하지 않는 값이 존재하더라도 해당 위치의 데이터와 참조되는 데이터의 차이 값의 테이블 인덱스 또는 데이터 자체를 저장하여 부분매칭을 유지할 수 있다.

비록 해쉉을 사용하여 후보 군의 수를 줄였더라도, 최적의 부분매칭을 찾는 것은 많은 시간이 소요되므로, LZcode는 원본크기로부터 줄어드는 비트수의 평균이 가장 큰 위치로의 거리정보를 저장한다. 여기에서 줄어드는 비트수란, 하나의 매크로블록의 원본 크기인 32비트(2개의 2바이트 명령어)에서 거리정보를 적용하였을 경우, 감소되는 비트의 양을 의미한다. 예를 들면, 현재 압축하고자 하는 매크로블록과 참조되는 매크로블록의 데이터가 일치하고, 거리정보가 앞의 매크로블록이 참조하는 거리정보와 동일하다면, 해당 매크로블록을 압축하기 위한 비트 수는 복사플래그 2개와 거리정보 플래그 1개를 합친 3이 되고 줄어드는 비트 수는 32-3 인 29가 된다.

부분매칭의 길이는 두 개의 연속적인 매크로블록 예외 값(exception)이 발생 할 때까지가 된다. 여기에서 매크로블록 예외 값이란 줄어드는 비트수가 0인 매크로블록을 의미한다.

위 내용을 정리하면 해쉉을 통해 선택 된 후보 군을 이용하여 두 개의 연속적인 매크로블록 예외 값이 나올 때까지의 부분매칭들 중에서 줄어드는 비트수의 평균이 가장 큰 부분매칭을 선택하여 저장한다.

2.4.3 압축 알고리즘

위 내용들을 적용 한 압축 알고리즘은 다음과 같다.

Step 1: 초기화
- 첫 4개의 매크로블록은 예외 값으로 저장
- 4개 매크로블록의 총 8개의 명령어들의 위치 정보를 해쉬 테이블에 저장

Step 2: 검색리스트 구성
- 검색리스트에 Prev_dist 삽입
- 매크로 블록의 명령어를 키 값으로 하는 해쉬테이블의 위치 값과 현재 압축하고자 하는 데이터의 위치 값과의 거리정보를 검색리스트에 삽입

Step 3: 검색
- 검색 리스트의 각 거리정보에 대하여 2개의 매크로블록 예외 값(exception)이 발생 할 때까지 매칭 진행, 감소하는 비트수의 평균값을 계산
- 감소되는 비트수 평균값이 제일 큰 거리정보와 길이를 저장
- 매칭길이 내의 명령어의 위치를 해쉬테이블에 저장

Step 4: 인코딩(4개의 매크로블록 마다)
- 거리정보 플래그와 거리정보를 저장: 거리정보 플래그는 바이트 단위로 처리 가능하도록 별도로 저장
- 8개의 복사 플래그를 한 바이트로 저장
- 예외플래그 저장
- Diff table index 또는 예외 값을 저장

Step 5: Step2~4를 반복

2.5 zlib과의 차이점 요약

제안하는 방식과 zlib과의 가장 큰 차이점은 비트 단위의 저장을 최소화 함으로써, 압축 해제 속도를 높이는 것이다. 비트 단위로 저장된 데이터를 읽어오기 위해서는 버퍼를 사용하여야 하고 여기에 조건 문이 매 번 수

행되어 바이트 단위로 저장된 데이터를 읽어오는 것보다 시간이 많이 소요된다. 따라서, 압축 해제 속도를 빠르게 하기 위하여 비트 단위의 저장을 최소화하는 압축 형식을 사용한다. 결과적으로 비트 단위로 저장되는 데이터의 수가 zlib보다 5배 이상 줄어들기 때문에 압축 해제 속도가 4배 이상 줄어들게 된다.

비트 단위의 저장을 최소화 하는 대신, 바이트 단위의 저장이 증가하였기 때문에 압축률의 손실이 발생하게 된다. 이를 극복하기 위해서, 배타 논리합을 이용한 차이 값 테이블을 사용하고, 부분매칭을 사용하여 저장하는 거리 정보 값의 수를 최소화 하고, 부분매칭을 효율적으로 계산하기 위하여, 후방향성 정련(backward refinement)을 사용한다.

3. 실험 결과

실험은 현재 휴대 전화에서 사용중인 퀄컴 기반의 세 가지의 프로그램 코드 바이너리를 1M byte, 4K byte 블록단위로 압축률과 압축해제 시간을 두 가지 환경에서 측정하였다. 그 중 하나는 CPU가 ARM9 624MHz, RAM이 128MB일 때, 1MB를 NAND로부터 읽어오는 시간이 평균 86ms이다(표 1 참조). 다른 환경은 CPU가 ARM9 146MHz, RAM이 64MB일 때, 1MB를 NAND로부터 읽어오는 시간이 평균 487ms이다(표 2 참조).

압축률은 1M 단위에서 zlib 보다 약 4% 높고 4K 단위에서는 zlib과 유사하다. 압축해제 시간은 ARM9 624MHz, RAM 128MB에서 1M 단위: 1.8배, 4K 단위: 3.5배 zlib 보다 빠르고 ARM9 146MHz, RAM 64MB에서 1M 단위: 4.6배, 4K 단위: 5.1배 zlib 보다 빠르다. 4K 단위에서 압축해제 속도 향상이 1M 단위에서 보다 좋으며 이는 더 빠른 압축해제 속도를 요구하는 압축 디맨드 페이징에 적합 함을 보여준다.

압축코드 시스템의 부팅(booting) 시간은 압축파일 로딩 시간과 압축해제 시간의 합이 되는데, 압축파일 로딩 시간이 압축률에 비례하여 줄어들고 압축해제 시간이 매우 빠르므로 압축코드 부팅 시간이 비 압축코드 부팅 시간에 비하여 10~20% (CPU의 성능과 로딩 성능에 따라 편차를 보임) 향상을 보였다. 표 1의 환경에서 1MB를 읽어오는데 걸리는 시간은 86ms이다. 압축률이 약 50%라면 압축된 코드를 읽어오는데 43ms가 걸리고 이를 압축해제 하는데 35ms가 걸린다. 결국 압축을 하지 않았을 때보다 약 8ms 정도 빠르게 읽어올 수 있다. 표 2의 환경에서는 1MB를 읽어오는데 압축을 하지 않았을 때 보다 약 130ms 정도 빠르게 읽어올 수 있다.

종합하여 보면 본 방법을 사용한 코드 압축은 NAND 사용량을 약 반으로 줄이면서 부팅 시간과 디맨드 페이징을 포함한 시스템 성능을 10~20% 향상시킨다.

표 1 테스트 환경: ARM9 624MHz, RAM 128MB
1MB NAND Read Time: 86ms

Code Binary 1		size	69MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	48%	61.244
	4K	45%	0.503
LZcode	1M	52%	34.595
	4K	44%	0.146

Code Binary 2		size	72MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	51%	59.955
	4K	47%	0.468
LZcode	1M	55%	33.820
	4K	46%	0.135

Code Binary 3		size	65MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	49%	61.908
	4K	46%	0.477
LZcode	1M	54%	34.227
	4K	45%	0.130

표 2 테스트 환경: ARM9 146MHz, RAM 64MB
1MB NAND Read Time: 487ms

Code Binary 1		size	69MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	48%	511.125
	4K	45%	2.241
LZcode	1M	52%	111.625
	4K	44%	0.434

Code Binary 2		size	72MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	51%	492.342
	4K	47%	2.120
LZcode	1M	55%	104.231
	4K	46%	0.421

Code Binary 3		size	65MB
Method	unit	comp ratio	avg dec time (ms)
zlib	1M	49%	508.167
	4K	46%	2.183
LZcode	1M	54%	109.980
	4K	45%	0.423

압축률의 조절은 2.3절에서 제안한 후방향성 정련의 범위를 조절하거나, 2.2.2절에서 제안한 차이 값 테이블의 크기를 변경하고, 테이블에 접근 할 때, 고정된 한

바이트의 인덱스로 접근하는 것이 아니라, 비트 단위의 가변 길이의 인덱스로 접근 하는 것이다. 후 방향성 정련의 범위를 증가시키면 표 3과 같이 압축 시간이 대폭 늘어나는데 비하여 압축률은 크게 개선되지 않는다. 또한, 가변 길이의 인덱스를 이용하는 방법을 사용하면, 표 4와 같이 테이블의 크기에 따라 압축률이 증가하지만 압축 해제 속도 역시 증가하게 된다. 표 4에서 보면 256T는 바이트 단위의 인덱스를 이용하여 차이 값 테이블에 접근하는 방법이고, 508T, 1020T, 2044T는 비트 단위의 가변 길이 인덱스를 이용하는 방법이다.

표 3 테스트 환경: Intel E5300(2.6GHz)

Code Binary 1		size	69MB
refinement range	unit	comp ratio	avg comp time (ms)
2	1M	50.5%	1214
4	1M	51.5%	2143
8	1M	52%	4410
16	1M	52.2%	8207

표 4 테스트 환경: ARM9 146MHz, RAM 64MB

Code Binary 1		size	69MB
Table size	unit	comp ratio	avg dec time (ms)
256T	1M	52%	111.625
508T	1M	53.1%	125.239
1020T	1M	53.4%	126.164
2044T	1M	53.5%	125.784

4. 결 론

본 논문은 비트 연산을 최소화하는 새로운 형식의 효율적인 프로그램 코드 압축방법을 제안하였다. LZcode는 zlib의 압축 방식에 비해 최대 5배 가량 압축해제 속도가 빠르고, 압축률 또한 최대 4%의 향상이 되었다. 특히, 본 방법은 NAND 사용량을 절반으로 줄이면서 부팅 시간을 10~20% 향상시킨다.

향후 연구과제는 압축시간 또한 대폭 개선하여 무선으로 코드를 update 하는 FOTA(Firmware Over The Air)에도 압축코드가 사용될 수 있게 하는 것이다.

참 조 문 헌

[1] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," *Proc. 25th Ann. International Symposium on Micro architecture*, pp.81-91, December 1992.

[2] S. Seong and P. Mishra, "Bitmask-based code compression for embedded systems," *IEEE Trans. CAD*, vol.27, no.4, pp.673-685, 2008.

[3] H. Lekatsas, J. Henkel, and W. Wolf, "Code Compression for Low Power Embedded System Design," *Proc. ACM/IEEE Design Automation Conference*, Los Angeles, CA, June 2000.

[4] X. Xu, S. Jones, C. Clarke, "ARM/THUMB code compression for embedded systems," *15th International Conference on Microelectronics (ICM 2003)*, 9-11 December 2003, Cairo, Egypt.

[5] N. Aslam, M. J. Milward, A. T. Erdogan, and T.Arslan, "Code Compression and Decompression for Coarse-Grain Reconfigurable Architectures," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol.16, no.12, December 2008.

[6] M. Kozuch and A. Wolfe, "Compression of embedded system programs," *Proc. Int. Conf. on Computer Design*, 1994.

[7] 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.

[8] J. N. Larsson, and A. Moffat, "Offline dictionary based compression," *Proc. Data Compression Conference '99*, pp.296-305, 1999.

[9] C. Nevill-Manning and I. Witten, "Compression and Explanation using Hierarchical Grammars," the Computer Journal, vol.40, no.2/3, pp.103-116, 1997.

[10] H.E. Yang and C. J. Keiffer, "Efficient universal lossless data compression algorithm on a greedy sequential grammar transform," *IEEE Trans. Inf. Theory*, vol.46, no.3, pp.755-777, 2000.

[11] A. J. Storer J, and G. T. Szymanski, "Data Compression via Textual Substitution," JACM vol.29, no.4, pp.928-951, 1982.

[12] C. Park, J.-U. Kang, S.-Y. Park and J.-S. Kim, "Energy-aware demand paging on NAND flash-based embedded storages," in Proceedings of the 2004 International Symposium on Low Power Electronics and Design (ISLPED'04), pp.338-343, Newport, USA, August 2004.

[13] Hirschberg, D.S., "'Algorithms for the Longest Common Subsequence Problem," *Journal of ACM*, vol.24, no.4, pp.664-675, 1977.

김 용 관

2008년 2월 아주대학교 정보 및 컴퓨터 공학부(공학사). 2008년 1월~2009년 8월 큐램 연구원. 2009년 9월~현재 아주대학교 일반대학원 컴퓨터 공학과 통합과정 중. 관심분야는 영상처리, 정보 압축, 컴퓨터 그래픽스

위 영 철

1982년 12월 State Univ. of NY at Albany 전산학과(학사). 1984년 12월 State Univ. of NY at Albany 전산학과(석사). 1989년 12월 State Univ. of NY at Albany 전산학과(박사). 1990년 3월~1995년 4월 삼성종합기술원 수석연구원. 1995년 5월~1998년 2월 현대전자 기획부장. 1998년 3월~현재 아주대학교 정보 및 컴퓨터공학부 교수. 관심분야는 영상처리, 정보 압축, 컴퓨터 그래픽스