21-1-R03

# CAM(Copy-Add-Move)을 이용한 바이너리 패치 방법*

## (Binary Patch Method using CAM(Copy-Add-Move))

이민재* · 한경숙* · 우덕균**

한국산업기술대학교* · 한국전자통신연구원**

{wizz, khan}@kpu.ac.kr* · dkwu@etri.re.kr**

요 약

현대의 많은 소프트웨어는 프로그램의 안정성과 기능 향상을 위해서 소프트웨어 업데이트를 사용한다. 초기의 소프트웨어 업데이트는 변경된 파일을 교체하는 방법으로 이루어졌으나 최근에는 업데이트를 제공하는 서버의 네트워크 트래픽과 저장 공간의 부담을 줄이기 위해서 구 버전과 신 버전간의 차이점을 추출한 델타 패키지를 이용하는 방식을 사용한다. 본 논문은 파일의 바이너리를 일종의 문자열로 보고 문자열 매칭 알고리즘을 이용하여 차이점을 추출하는 바이너리 패치 도구를 제안한다. 가장 길고 최대한 근접한 매치를 찾기 위해 zdelta와 같이 윈도우에 존재하는 모든 바이너리 패턴을 인덱싱하는 방식을 이용하였다. vcdiff와 zdelta에서 사용한 복사 기반 방식을 확장하여 1:1 매치되는 구간에 있어서 Move의 개념을 추가한 CAM(Copy-Add-Move) 방식을 새로 도입하고, 기존의 vcdiff에서 사용한 1바이트 델타 명령 체계 세분화를 통해 델타 명령이 사용하는 인수를 줄임으로써 압축률을 향상시켰다. 델타 압축의 최악 경우(worst case)는 두 바이너리 버전 간의 차이가 큰 경우 발생한다는 것을 실험을 통해 밝히고 이와 같은 경우에는 오히려 델타 압축을 수행하는 것보다 데이터 압축을 수행하는 것이 더 좋은 압축률을 얻을 수 있다. 따라서 그 대안으로 wdiff는 RAR32를 이용하여 대상 파일을 데이터 압축하는 정책을 사용하였다.

## 1. 서 론

소프트웨어 업데이트는 이미 배포된 프로그램의 안정성과 기능 향상을 위해 부분적으로 재배포하는 방식이라고 할 수 있다. 초기의 소프트웨어 업데이트는 변경된 파일을 교체하는 방식으로 이루어져왔다. 이는 패치 과정이 간단하고 구현이 용이하다는 장점이 있으나 파일의 크기가 크기 때문에 패치를 제공하는 서버의 저장 공간과 네트워크 트래픽 증가의 부담이 컸다. 이러한 문제점을 보완하기 위해 델타 압축이 등장하였다. 델타 압축이란 두 개의 데이터 셋(구 버전과 신 버전)의 바이너리간의 차이점을 추출하여 인코딩하는 압축 기법을 말하며, 이를 통해 산출된 결과물을 델타 패키지라고 한다[1, 2]. 최근의 소프트웨어 업데이트는 이러한 델타 패키지를 클라이언트로 전송하여 디코딩하는 방식으로 이루어진다.

* 본 연구는 한국전자통신연구원의 "모바일 S/W 원격 업그레이드 도구 기술 분석 연구"과제의 지원을 받아 수행하였습니다.

초기의 바이트 기반의 델타 압축은 구 버전과 신 버전의 파일을 두 개의 문자열로 보고 첫 번째 문자열을 기준으로 삽입, 삭제, 갱신 명령을 조합하여 대상 문자열을 표현하였다. 그러나 이 방법은 같은 주소의 바이너리를 1:1로 비교하여 이루어지므로 높은 성능은 기대할 수 없었다[3].

최근에는 보다 압축률을 높이기 위해 복사 기반 방식이 사용된다. 대상 데이터 셋의 패턴은 위치는 다르지만 참조 데이터 셋에서 찾을 수 있는 경우가 많다. 이러한 사실에 근거하여 대부분의 대상 데이터 셋은 참조 데이터 셋의 복사 조작의 조합으로 표현이 가능하고 없는 패턴에 대해서는 추가하는 방식으로 이루어진다.

본 논문에서는 vcdiff와 zdelta에서 사용하는 복사 기반 방식을 확장하여 1:1로 매치되는 구간을 Move 명령을 사용하여 인코딩하는 CAM(Copy-Add-Move)방식을 제안한다. 이는 COPY로 판정 중에서 원본과 대상이 1:1로 매치되는 경우 즉, 원본 파일의 포인터와 대상 파일의 포인터가 같은 주소를 참조하는 경우에는 MATCH로 판정하여 참조 주소를 생략하는 것과 자주 발생할 수 있는 작은 명령들을 세분화하여 인수를 생략하는 것이다. 이러한 명령의 세분화와 확장을 통해 압축률을 향상시켰다.

또한 인코딩된 고정 형식의 델타 명령의 사용과 1바이트로 코드화된 명령을 사용하여 간단하게 디코딩할 수 있도록 하였다. 따라서 임베디드 시스템과 같이 열악한 환경에서도 사용하는 데 무리가 없을 것이다. 이 방식을 임베디드를 위한 펌웨어 또는 소프트웨어 업데이트를 위해 사용하다면 큰 효과를 얻을 것으로 기대한다. 그리고 본 논문에서 제공하는 바이너리 패치 도구를 wdiff라고 명명하도록 하겠다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 본 논문과 관련된 다른 유사 도구들에 대해 살펴보고 3장에서 wdiff의 판정과 명령 체계에 대해 설명한다. 4장에서는 wdiff의 델타 추출 방법을 설명한다. 5장에서는 wdiff의 델타 인코딩과 디코딩 방법에 대해 설명한다. 6장에서는 wdiff와 다른 유사 도구들과의 실험을 통해 결과를 비교하고 분석한다. 마지막으로 결론과 향후 과제에 대하여 설명한다.

## 2. 관련연구

본 장은 wdiff와 유사한 대표적인 바이너리 패치 도구들을 살펴보도록 하겠다.

### 2.1 vcdiff

vcdiff[3]는 IETF Standard (RFC3284)에 등록된 델타 압축 방법으로 많은 바이너리 패치 도구들이 이것을 기본으로 하여 만들어진 경우가 많다. vcdiff는 압축률을 향상하기 위해 명령 코드 테이블을 사용하여 델타 명령을 1바이트로 인코딩 한다.

(표 1) vcdiff의 명령 코드 테이블[3]

| | Type | Size | Mode | Type | Size | Mode | Index |
|---|---|---|---|---|---|---|---|
| 1 | RUN | 0 | 0 | NOOP | 0 | 0 | 0 |
| 2 | ADD | 0, [1, 17] | 0 | NOOP | 0 | 0 | [1, 18] |
| 3 | COPY | 0, [4, 18] | 0 | NOOP | 0 | 0 | [19, 34] |
| 4 | COPY | 0, [4, 18] | 1 | NOOP | 0 | 0 | [35, 50] |
| 5 | COPY | 0, [4, 18] | 2 | NOOP | 0 | 0 | [51, 66] |
| 6 | COPY | 0, [4, 18] | 3 | NOOP | 0 | 0 | [67, 82] |
| 7 | COPY | 0, [4, 18] | 4 | NOOP | 0 | 0 | [83, 98] |
| 8 | COPY | 0, [4, 18] | 5 | NOOP | 0 | 0 | [99, 114] |
| 9 | COPY | 0, [4, 18] | 6 | NOOP | 0 | 0 | [115, 130] |
| 10 | COPY | 0, [4, 18] | 7 | NOOP | 0 | 0 | [131, 146] |
| 11 | COPY | 0, [4, 18] | 8 | NOOP | 0 | 0 | [147, 162] |
| 12 | ADD | [1, 4] | 0 | COPY | [4, 6] | 0 | [163, 174] |
| 13 | ADD | [1, 4] | 0 | COPY | [4, 6] | 1 | [175, 186] |
| 14 | ADD | [1, 4] | 0 | COPY | [4, 6] | 2 | [187, 198] |
| 15 | ADD | [1, 4] | 0 | COPY | [4, 6] | 3 | [199, 210] |
| 16 | ADD | [1, 4] | 0 | COPY | [4, 6] | 4 | [211, 222] |
| 17 | ADD | [1, 4] | 0 | COPY | [4, 6] | 5 | [223, 234] |
| 18 | ADD | [1, 4] | 0 | COPY | 4 | 6 | [235, 238] |
| 19 | ADD | [1, 4] | 0 | COPY | 4 | 7 | [239, 242] |
| 20 | ADD | [1, 4] | 0 | COPY | 4 | 8 | [243, 246] |
| 21 | COPY | 4 | [0, 8] | ADD | 1 | 0 | [247, 255] |

vcdiff는 델타 추출을 위해 윈도우 알고리즘을 이용한다. 문자열 매치를 위한 인덱스로서 처음 출현하는 가장 긴 매치에 대한 4바이트의 패턴을 해시 테이블에 저장한다. 이와 같은 방법으로 대상 파일에 대한 모든 매치를 수행했다면 그 결과를 명령어 코드 테이블을 사용하여 인코딩한다.

명령 코드 테이블에 명시되어 있는 명령들은 ADD, COPY, RUN의 세 가지로 나누어진다. ADD 명령은 일치하는 패턴이 없어서 문자열을 추가해야 할 때 사용하며, 그 인수로 는 추가 문자열의 길이와 데이터를 사용한다. COPY 명령은 같은 파일 주소에 원본과 대상 문자열이 일치하거나 다른 부분에 있을 경우 사용하며, 그 인수로는 복사할 바이너리 문자열의 길이와 오프셋을 사용한다. RUN 명령은 같은 문자가 여러 번 반복되어 추가될 경우 사용하며, 그 인수로는 반복되는 문자, 문자열의 길이, 그리고 반복 회수를 사용한다. 이와 같은 방법으로 초벌 델타 명령을 생성한 후에 최적화 과정을 통해 명령들을 더 축약하게 된다.

vcdiff는 모든 패턴을 인덱스에 저장하지 않고 처음 출현하는 가장 긴 매치에 대한 패턴을 저장한다. 그러므로 인덱스에서 검색할 패턴의 수가 적어 실행 속도는 빠를 수 있지만 가장 가까운 패턴을 찾는 데에는 어느 정도 한계가 있다.

## 2.2 zdelta

zdelta[2]는 델타 패키지를 추출하여 인코딩 하는 과정에서 데이터는 LZ77에 의해서 압축하고 생성된 델타 명령들은 허프만 코드를 이용하여 2차 압축하여 패키지를 줄이는 방법을 사용한다.

zdelta에서도 다른 유사 도구들과 같이 패턴을 찾기 위해 인덱스를 생성하고 문자열 매칭 알고리즘을 통해 패턴을 찾는다. 문자열 매치에 있어서는 기본적으로 vcdiff의 방법을 따르고 있으나 가장 길고 가까운 매치를 찾기 위해서 원본 윈도우의 모든 바이너리 문자열에 대하여 존재하는 3바이트 패턴을 인덱스로 만든다.

매치는 길이와 오프셋(포인터, 오프셋 방향)으로 나타내며 길이는 최대 1,026, 오프셋은 0～32,766까지 표현할 수 있으며 세 개 이상 일치하지 않는 매치는 방출하고 다음 문자에서 수행하도록 한다. 그리고 그리디 알고리즘(greedy algorithm)을 이용하여 가장 길고 가장 오프셋이 작은 매치를 선택한다. 그러나 오프셋의 크기가 4,096을 초과 시에는 매치의 길이를 감소시키기 위해 더 가까운 포인터에 매치하도록 한다.

큰 파일에 대한 대비로서 해시 테이블의 크기는 64KB의 윈도우를 사용하며 버킷은 1,024개로 제한을 둔다. 그리고 매치가 윈도우의 크기의 50%이상 진행했을 경우 윈도우를 절반 단위로 슬라이딩하고 해시 테이블을 다시 생성한다.

vcdiff와 달리 zdelta는 모든 패턴에 대한 해시 테이블을 만들기 때문에 매치의 길이를 알아내기 위한 추가 비용이 발생한다. 그러므로 최악 시나리오에서의 속도 면에서 성능 향상은 있을 수 있으나 실 데이터 셋에서는 성능 향상이 크지 않다. 그러나 최적의 매치를 찾을 수 있기 때문에 델타 패키지 용량을 보다 감소시킬 수 있다.

이렇게 생성된 델타 명령들(오프셋, 매치 포인터, 방향, 길이 계수)은 허프만 코드를 이용하여 2차 압축을 수행하여 압축률을 향상시킨다. 그러나 2차 압축은 임베디드 시스템에서 디코딩 시 메모리와 CPU의 부담이 될 수 있다. xdelta3에서는 2차 압축의 여부를 지정할 수 있도록 되어있다.

## 3. wdiff의 델타 판정과 명령 체계

### 3.1 MATCH 판정에 대한 인코딩

MATCH로 판정된 델타 명령은 원본 문자열과 대상 문자열이 같은 곳에 위치한 경우이다. 이러한 경우에는 참조 주소를 나타낼 필요가 없다. 또한 ADD 명령군 사이에 작은 MATCH 명령들이 존재할 가능성이 크다. 따라서 작은 MATCH 명령에 대하여 차지하는 바이트를 최대한 줄이기 위해 1:1로 일치한 바이트 수가 1~16 바이트인 MATCH 명령의 경우에는 (표 3)의 "MOV0~MOVF" 명령을 사용하여 1바이트로 축약한다. MATCH 판정의 길이가 1바이트로 표현 가능한 경우에는 "XMOV0~XMOVF"를 사용하여 인코딩하고 2바이트를 소요하게 된다. 그리고 MATCH 판정의 경우에는 긴 매치가 존재할 가능성이 크기 때문에 판정의 길이가 2바이트 범위, 3바이트 범위까지도 표현할 수 있어야 한다. 이러한 경우에는 추가 명령 "XMOVX", "XMOVXX"를 사용한다.

### 3.2 ADD 판정에 대한 인코딩

ADD로 판정된 델타 명령은 원본 윈도우에서 같은 패턴을 찾을 수 없는 경우로서 문자열을 추가해야 한다. 특히, .header 섹션은 파일의 정보가 바뀌면서 버전 정보 등의 작은 바이너리가 변경될 가능성이 크며, 컴파일된 바이너리가 들어있는 .text 섹션에서는 주소 값만 변경된 경우도 많다. 이러한 경우에 인덱스의 해당 패턴이 없으면 ADD로 판정되고 적은 양의 바이트를 추가하는 ADD 명령들이 발생할 수 있다. 적은 양의 문자열을 추가하는 ADD 명령에서 인수를 사용하지 않음으로써 압축률을 증가시킬 수 있다. 이를 위해 MATCH 판정에서와 같이 명령들을 세분화하여 인수를 줄였다.

(표 2) 명령 코드 테이블 범례

| 코드 | 내용 | 코드 | 내용 | 코드 | 내용 |
|---|---|---|---|---|---|
| D | 데이터 | Ref | 참조주소 | N | 반복횟수 |
| DL | 데이터의 길이 | 1P | 첫째자리 | F | 플래그 |
| ML | 매치의 길이 | 2P | 둘째자리 | | |
| Ref | 참조 주소 | 3P | 셋째자리 | | |

(표 3) 명령 코드 테이블

| 코드 | 명령 | 인수1 | 인수2 | 인수3 | 인수4 | 소요 바이트 |
|---|---|---|---|---|---|---|
| 0x00 | MATCH | - | - | - | - | - |
| 0x01 | ADD | - | - | - | - | - |
| 0x02 | COPY | - | - | - | - | - |
| 0x03 | XMOVEX | 1P ML | 2P ML | | - | 3 |
| 0x04 | XMOVEXX | 1P ML | 2P ML | 3P ML | - | 4 |
| 0x05 | RUN | - | - | - | - | 2 |
| 0x10 | MOV0 | - | - | - | - | 1 |
| … | … | … | … | … | … | … |
| 0x1F | MOVF | - | - | - | - | 1 |
| 0x20 | XMOV0 | ML | - | - | - | 2 |
| … | … | … | … | … | … | … |
| 0x2F | XMOVF | ML | - | - | - | 2 |
| 0x30 | ADD0 | D | - | - | - | 2 |
| … | … | … | … | … | … | … |
| 0x3F | ADDF | D | - | - | - | 17 |
| 0x40 | XADD0 | DL | D | - | - | 2 + DL |
| … | … | … | … | … | … | … |
| 0x50 | PCOPY | Ref | - | - | - | 2 |
| 0x51 | NCOPY | Ref | - | - | - | 2 |
| 0x52 | XPCOPY1 | Ref | ML | - | - | 3 |
| 0x53 | XPCOPY2 | F | Ref | ML | - | 4 |
| 0x54 | XNCOPY1 | Ref | ML | - | - | 3 |
| 0x55 | XNCOPY2 | F | Ref | ML | - | 4 |
| 0x56 | SAME_PCOPY | Ref | N | - | - | 3 |
| 0x57 | SAME_NCOPY | Ref | N | - | - | 3 |
| 0x58 | SAME_XPCOPY1 | Ref | ML | N | - | 4 |
| 0x59 | SAME_XPCOPY2 | F | Ref | ML | N | 5 |
| 0x5A | SAME_XNCOPY1 | Ref | ML | N | - | 4 |
| 0x5B | SAME_XNCOPY2 | F | Ref | ML | N | 5 |
| 0x60 | XRUN0 | DL | D | - | - | 3 |
| … | … | … | … | … | … | … |
| 0x6F | XRUNF | DL | D | - | - | 3 |
| 0xFF | EOI | - | - | - | - | - |

1~16바이트의 추가 문자열에 대한 ADD 명령은 (표 3)의 "ADD0~ADDF" 명령으로 인코딩하여 바이트 수를 표시하는

인수를 사용하지 않는다. 1바이트 이상의 길이로 표현되는 경우에는 XADD 명령군을 사용하게 되는데, 추가 문자열의 바이트수의 하위 한 바이트는 인수로 사용하고 상위 한 바이트에 대하여는 명령으로서 구분한다. 예를 들면 추가 문자열의 길이가 0x210일 때 하위 한 바이트인 0x10에 대해서는 인수로 사용하고 0x02에 대해서는 명령으로 분리한다. 그러므로 XADD2 명령을 사용하고 추가 문자열의 길이를 나타내는 인수로 0x10을 사용한다. 이와 같이 명령의 세분화를 통해 인수의 사용을 줄임으로써 압축률을 향상시켰다.

## 3.3 COPY 판정에 대한 인코딩

COPY로 판정된 델타 명령은 우선 두 가지 부류로 나누어 질 수 있다. 참조 주소가 데이터가 쓰일 주소(대상 주소)를 기준으로 앞에 있는지 뒤에 있는지에 따라 NCOPY 명령군과 PCOPY 명령군으로 나누어진다. 이렇게 오프셋의 방향을 표시하는 것으로 더 많은 매치를 찾을 수 있고, 참조 주소 표현에 소요되는 바이트 수를 줄일 수 있다.

인덱스 패턴의 길이가 4바이트이므로 COPY의 기본 단위도 4바이트이다. 따라서 4바이트만 차지하는 COPY에 대해서는 바이트 길이를 표시하지 않고 PCOPY와 NCOPY 로 인코딩한다. 4바이트보다 긴 COPY명령을 위해서 1바이트 내에서 바이트수를 표현할 수 있는 경우 XPCOPY1, XNCOPY1로 인코딩한다.

COPY 명령에서의 관건은 매치의 길이를 나타내는 바이트 수만의 문제는 아니다. 참조 주소의 길이를 얼마나 축약할 수 있는가도 큰 관건이다. 그러므로 wdiff는 기본적으로 참조 주소 표현을 위한 바이트를 줄이기 위해 현재 COPY 명령에 의해 데이터가 쓰일 주소를 기준으로 하는 가상 주소로 참조 주소를 표시하고 있으며, 멀리 떨어진 매치를 위해 플래그를 사용한다. XPCOPY2와 XNCOPY2 명령은 1바이트의 플래그를 가지게 되는데, 이 플래그는 니블 단위로 의미를 달리 가진다. 상위 니블은 참조 주소의 상위 숫자, 하위 니블은 바이트수의 상위 숫자로 표현한다. 그리고 이것은 뺀 나머지는 1바이트가 인수로 들어간다. 이와 같이 플래그를 통해서 참조 주소와 바이트 수를 표현하는 COPY 명령의 인수가 차지하는 바이트를 줄임으로서 압축률을 향상시켰다.

COPY 명령 중에서 같은 참조 위치에서 복사해오는 경우가 자주 발생하므로 이러한 경우에는 같은 명령 그룹으로 보고 반복 횟수를 표시하는 한 줄의 명령으로 축약할 수 있다. wdiff는 SAME_COPY 명령군으로 이와 같은 패턴들을 처리한다. 인코딩하는 규칙은 일반 COPY 명령군과 동일하지만 이 명령들이 몇 번 반복되었는지를 나타내는 인수가 하나 더 추가된다.

## 3.4 RUN 판정에 대한 인코딩

같은 문자의 반복으로 구성된 문자열을 추가하는 RUN 명령에 대한 인코딩은 4바이트 데이터의 반복을 표현하는 RUN 명령과 그 이상의 반복을 표현하는 XRUN 명령군으로 나누어진다. XRUN 명령군의 경우 XADD 명령군과 같이 차지하는 바이트수를 나타내는 인수를 축약하기 위해 하위 1바이트를 제외한 상위 숫자로 명령을 세분화하여 사용함으로써 압축률을 증가시켰다.

## 3.5 명령 코드 테이블

wdiff의 델타 명령은 (표 2), (표 3)와 같이 명령 코드와 인수로 구분되어져 있으며,

모든 명령 코드는 1바이트로 되어 있다. 인수로는 참조 주소, 데이터의 길이, 반복횟수, 데이터(바이너리 문자)가 사용된다. 표 1과 같이 vcdiff에서도 명령 코드 테이블을 사용하지만 두 개의 명령을 묶어서 하나의 코드로 사용하는 연계 명령 방식이다. 그러나 wdiff는 한 단위 명령만을 사용하고 연계 명령을 사용하지 않는 대신에 자주 발생할 수 있는 명령은 세분화하였다. 같은 참조 주소를 사용하는 COPY 명령의 경우에는 SAME_COPY 명령군으로, 대상 파일과 1:1 매치되는 구간의 COPY 명령은 MOV 명령군으로 확장하였다. 이러한 세분화와 확장을 통해 인수의 사용을 최소화함으로써 압축률을 향상시켰다. 그러나 인수를 사용을 줄이기 위해 인수를 분할하거나 생략하여 명령으로 분리하는 다소 복잡한 연산이 필요하다.

## 4. wdiff의 델타 추출

wdiff의 델타 압축을 위한 처리 흐름은 다른 유사 도구들과 같이 윈도우 알고리즘을 이용한다. 원본 파일과 대상 파일을 읽어서 4,096 바이트의 윈도우 버퍼와 원본 파일에 대한 윈도우를 생성한다. 그리고 생성된 윈도우에 대해서 델타를 추출하게 된다. 윈도우의 50%이상 진행되었을 경우 1,024 바이트만큼 슬라이딩하여 다음 작업을 진행한다. 이와 같이 파일의 끝까지 윈도우를 이동하면서 매치를 수행하고 델타 명령을 생성하게 된다. 그리고 생성된 델타 명령은 명령코드테이블의 형식에 준하여 명령어는 1바이트로 인코딩하고 그에 필요한 인수(주소, 데이터, 반복 회수)도 같이 인코딩하게 된다. 그러나 만약 인코딩한 결과, 압축률이 55%미만일 경우에는 두 버전 간의 차이점이 많아서 델타 압축이 비효율적인 경우가 많다(실험 참조). wdiff는 이러한 경우 대안으로 대상 파일을 데이터 압축한다.

효율적인 델타 추출을 위해서는 윈도우 내에 존재하는 패턴을 최대한 많이 그리고 가장 길게 매치되면서 가장 근접한 매치를 찾을 수 있어야 한다. 본 장에서는 이를 위해서 wdiff에서 구현된 인덱스 구조와 매치에 대한 판정에 대해서 설명하고자 한다.

### 4.1 인덱스 및 윈도우 운용

관련 연구에 있는 여러 도구들과 같이 wdiff도 문자열 매칭 알고리즘을 사용한다. 최대한 길이가 길고 현재 위치에서 가장 짧은 거리를 가지는 매치를 찾기 위해 zdelta와 같이 윈도우 내에 있는 모든 패턴에 대한 인덱스를 패턴 검색 전에 생성하여 사용한다[2].

매치를 위한 사전 작업으로 다른 유사 도구들과 같이 인덱스와 윈도우 버퍼를 사용한다. 인수를 통해 시작 주소와 종료 주소를 받고 그 범위만큼 원본 파일에 대해서는 인덱스를 생성하고, 대상 파일에 대해서는 윈도우 버퍼를 사용한다. 원본 파일에 대해서만 인덱스를 생성하는 이유는 대상 파일을 이용한 COPY 명령을 수행할 수 없기 때문이다. 대상 파일은 클라이언트에서 디코딩 작업을 통해 만들어지는 파일이므로 클라이언트에서는 참조할 경우 오버헤드가 증가한다. 따라서 디코딩 과정에서 대상 파일의 패턴을 사용하지 않는다. 그리고 차이점 추출 과정에서 대상 파일을 통해 이루어지는 작업은 1:1 매치의 여부를 검사하는 것이다. 따라서 대상 파일에 대한 인덱스는 불필요하다.

원본 파일에 대해서는 zdelta와 같이 원본 파일 내에 존재하는 모든 패턴에 대해서 인덱싱하기 때문에 원본 윈도우의 길이가 N이라고 하면 wdiff의 인덱스 패턴의

길이는 4바이트이므로, 최대 N−3개의 패턴에 대한 인덱스 노드가 생성된다. 각 인덱스는 패턴이 위치한 주소를 저장하기 위한 체인을 가진다. 그러나 파일 전체에 대한 인덱스를 만드는 것은 많은 메모리를 필요로 한다. wdiff에서 인덱스 생성 범위는 윈도우 단위는 4,096 바이트이므로 최대 인덱스 노드와 최대 체인의 길이는 4,093이 된다.

인덱스의 생성은 윈도우의 시작 주소와 종료 주소를 인수로 받아서 윈도우 버퍼 끝까지 순회하면서 작업을 수행한다. 우선 원본 파일로부터 1바이트의 바이너리를 읽어서 그 바이너리가 EOF이면 작업을 종료하고 그렇지 않으면 지금 읽어온 1바이트 바이너리를 윈도우 버퍼에 저장한다. 그리고 나머지 3바이트를 더 읽어서 4바이트의 패턴으로 만든다. 이 패턴이 현재 인덱스에 존재하는지 여부를 검사하여 존재하는 경우에는 체인에 항목을 추가하고, 체인 길이 카운터 변수를 증가시켜 그 길이를 알 수 있도록 한다. 만약, 그 패턴이 인덱스에 없으면 새로운 노드로 추가한다.

대상 파일에 대해서는 버퍼만 생성한다. 윈도우의 시작 주소와 종료 주소를 인수로 받아서 윈도우 버퍼 끝까지 대상 파일로부터 1바이트 바이너리를 읽어온다. 그 바이너리가 EOF이면 작업을 종료하고, 그렇지 않으면 1바이트씩 버퍼에 계속 저장하는 작업을 수행한다.

## 4.2 매치와 판정

(그림 1)과 같이 매치는 윈도우 단위로 이루어지며 대상 윈도우 버퍼가 소진되거나 파일의 끝에 도달할 때까지 대상 윈도우에 패턴을 읽어 온다. 그 패턴이 4바이트의 온전한 패턴이 아니거나 인덱스의 해당 패턴이 없다면 “not match” 상황이다. 이

```
델타 판정 (시작주소, 종료 주소) {
 if (대상 윈도우 위치 > 파일의 크기) {
   함수 종료;
 }
 <대상 윈도우 패턴>을 읽어옴;
 if (대상 윈도우 패턴이 4바이트 미만 || 인덱스에 패턴이 존재하지 않으면) {
   if(대상 윈도우 패턴가 4바이트 && 원소가 같으면) RUN 판정;
              else ADD 판정;
 } else {
   복사가 가능한 패턴 후보들을 복사 리스트에 저장;
   while (true) {
     if (모든 <복사 리스트>의 <매치 종료 플래그>가 매치종료 상태라면)
     break;

     if (연계 매치인가?) {
       <매치 포인터> 1증가;
       <매치 종료주소> 1증가;
      } else {
        <매치 종료 플래그>를 매치종료 상태로 설정;
      }
    }
 <최적 복사 패턴> = 복사 리스트 중에서 가장 큰 매치길이, 가장 작은
   옵셋인 요소를 선택;
 if (<최적 복사 패턴>의 <매치 시작 주소> == 현재 파일 주소) MATCH
  판정;
   else COPY 판정;
 }
 명령 그룹핑 함수 호출;
 다음 패턴을 읽음;
 }
```

(그림 1) 델타 판정 함수

때, 4바이트의 온전한 패턴이고 패턴의 모든 원소가 같은 경우에는 한 문자로 표현이 가능하므로 “RUN”으로 판정하고 그렇지 않으면, “ADD”로 판정한다.

읽어 온 대상 윈도우 패턴이 4바이트의 온전한 패턴이고 인덱스에 그 패턴이 존재한다면 그 패턴과 같은 문자열을 복사할 수 있으므로 “COPY”로 판정한다. “COPY” 판정은 연계된 매치를 더 찾기 위해 그 패턴의 체인을 가지고 COPY 리스트를 생성한다. COPY 리스트는 구조체 배열로 복사 작업을 위해 패턴의 시작 위치와 종료 위치, 매치의 길이, 현재 주소와 떨어진 거리로 구성되어 있다. 가장 긴 매치의 패턴을 찾기 위해 COPY 리스트의 모든 노드를 순회하면서 원본 파일의 바이너리와 대상 파일의 바이너리를 비교하여 연속된 “COPY” 판정들을 찾아내는 작업을 수행한다.

처음 “COPY”로 판정된 패턴 이후의 바

이너리를 계속 진행하면서 비교하기 위해 이동 인수를 사용하여 파일 윈도우에서 패턴 윈도우를 이동시켜서 패턴이 연속적으로 일치하는지 여부를 검사한다. 일치하는 경우에는 매치 종료 주소와 매치의 길이를 1만큼 증가하고, 일치하지 않으면 다음 COPY 리스트로 넘겨 같은 과정을 수행하게 된다. 이와 같은 과정을 통해 완성되어진 COPY 리스트는 복사해올 수 있는 패턴의 최대 바이트와 복사 패턴과의 거리를 알 수 있게 한다. 모든 가능성이 있는 매치를 구한 COPY 리스트는 그리디 알고리즘을 사용하여 가장 긴 매치 길이, 가장 작은 오프셋을 가진 패턴으로 최적의 매치를 선택한다.

복사에 대한 판정은 “MATCH”와 “COPY”의 두 가지로 나누어질 수 있다. 현재 파일 주소와 같은 매치는 “MATCH”로 판정하며 참조 주소를 생략하고 매치된 바이트로만 인코딩하여 소요되는 바이트 수를 줄인다. 그리고 현재 파일 주소와 다른 경우에는 “COPY”로 판정한다.

하나의 판정이 완료될 때마다 그룹핑 함수를 수행하게 된다. 그룹핑 함수는 이전 명령과 현재 명령이 같은지 여부를 판단하여 같은 판정일 경우 그룹으로 묶어준다. 그리고 COPY 판정 중에서 이전 명령과 참조 주소, 데이터의 길이가 같을 경우에는 SAME_COPY로 분류하는 작업을 수행한다. 이와 같은 과정을 통해서 델타 명령 하나가 완성된다.

## 5. wdiff의 델타 인코딩과 디코딩

차이점 추출을 통해 생성된 델타 명령들은 델타 패키지에 저장하게 되고 델타 패키지의 용량을 줄이기 위해 인코딩하게 된다. wdiff는 기본적으로 복사 기반 방식에 준하며 vcdiff와 같이 명령 코드 테이블을

```
인코딩_함수()
{
  for (모든 델타명령 순회) {
    if (현재델타명령.명령그룹 != 이전명령그룹) {
      if (이전명령그룹 != 0) {
        switch (현재델타명령.명령코드) {
        case MATCH:
          if (델타명령[인덱스 - 1].일련번호 - 1 < 0x10) {
            MOV명령군으로 인코딩;
          } else {
            if (델타명령[인덱스 - 1].일련번호 > 0xFFFF) {
              XMOVEXX 명령으로 인코딩;
            } else if (델타명령[인덱스 - 1].일련번호 > 0xFFF) {
              XMOVEX 명령으로 인코딩;
            } else {
              XMOV명령군으로 인코딩;
            }
          }
          break;
        case RUN:
          if (총_바이트 > 4) {
            XRUN명령군으로 인코딩;
          } else {
            RUN으로 인코딩;
          }
          break;
        case ADD:
          if (현재델타명령.일련번호 - 1 < 0x10) {
            ADD명령군으로 인코딩;
          } else {
            XADD명령군으로 인코딩;
          }
          break;
        case COPY:
          가상상대주소 생성;
          if (현재델타명령.플래그 == SAME_COPY) {
            총복사바이트수 계산;
            if (가상상대주소나 복사할 문자열이 1바이트 초과) {
                복사플래그 생성;
                SAME_XPCOPY2 또는 SAME_XNCOPY2로 인코딩;
            } else {
              if (현재델타명령.바이트 == 4) {
                SAME_PCOPY 또는 SAME_NCOPY으로 인코딩;
              } else {
                SAME_PCOPY1 또는 SAME_NCOPY1으로 인코딩;
              }
            }
          } else {
            총복사바이트수 계산;
            if (가상상대주소나 복사할 문자열이 1바이트 초과) {
                복사플래그 생성;
                XPCOPY2 또는 XNCOPY2 인코딩;
            } else {
              if (현재델타명령.바이트 == 4) {
                PCOPY 또는 NCOPY으로 인코딩;
              } else {
                PCOPY1 또는 NCOPY1으로 인코딩;
              }
            }
          }
          break;
        }
      }
      이전명령그룹 = 현재델타명령.명령그룹;
      다음명령으로이동;
    }
  }
}
```

(그림 2) 인코딩 함수

이용하여 1바이트 명령을 사용한다. 따라서 (그림 2)와 같이 판정을 통해 초벌 델타

명령들은 명령 코드 테이블에 대응하여 명령, 인수, 데이터가 델타파일로 저장된다. 그러나 vcdiff와 달리 연계 명령을 사용하지 않고 명령 코드 테이블을 더 세분화하고 확장하였다. COPY 판정 중에서 대상 파일의 바이너리와 원본 파일의 바이너리가 1:1 매치되는 경우 MOV 명령군으로 분리하고, 같은 참조 주소를 가지는 COPY 명령에 대하여 SAME_COPY 명령군을 분리한다. 이와 같이 세분화와 확장을 통해 인수의 사용을 최소화함으로 압축률을 향상시켰다. 그러므로 본 논문의 인코딩 및 델타 추출 방식은 복사 기반 방식에서 Move의 개념을 확장한 CAM(Copy-Add-Move)방식이라고 할 수 있겠다. 본 장에서는 판정에 의해서 초벌 생성된 델타 명령인 MATCH, ADD, COPY, RUN 명령이 인코딩 되는 과정과 이를 통해 산출된 델타 패키지를 디코딩되는 과정을 설명한다.

## 5.1 인코딩

(그림 3)은 wdiff가 실제로 인코딩 되는 과정을 나타내며 다음과 같은 순서로 진행된다.

① 대상 문자열의 0번지에서 읽어오는 패턴 “1234”는 인덱스에 존재하고 그 체인으로 0과 10을 가지고 있다. 연계 매치가 없으므로 현재 주소와 가장 가까운 체인 0이 선택되고 체인 0은 현재 주소와 같기 때문에 MATCH로 판정하고 “Match 4”로 인코딩 된다.

② 대상 문자열 4번지에서 읽어오는 패턴 “9012”는 인덱스에 존재하고 그 인덱스의 체인은 8이다. 그리고 패턴 이후에 연계매치가 존재하므로 총 복사 가능한 문자열은 9가 되므로 COPY로 판정되고 “Copy 8, 9”로 인코딩 된다.

| 원본 데이터 | 12345678901234567890 |
|---|---|
| 대상 데이터 | 12349012345670000567811123412 34 |

| | | | | |
|---|---|---|---|---|
| 1234 | → | 0 | 10 | |
| 2345 | → | 1 | 11 | |
| 3456 | → | 2 | 12 | |
| 4567 | → | 3 | 13 | |
| 5678 | → | 4 | 14 | |
| 6789 | → | 5 | 15 | |
| 7890 | → | 6 | 16 | |
| 8901 | → | 7 | ⊠ | |
| 9012 | → | 8 | ⊠ | |
| 0123 | → | 9 | ⊠ | |

| 쓰기 주소 | 초벌 인코딩 | | 최적화된 인코딩 |
|---|---|---|---|
| 0 | Match 4 | ⇒ | MOV3 |
| 4 | Copy 8, 9 | ⇒ | XPCOPY1 4, 9 |
| 13 | Run 0, 4 | ⇒ | RUN 0x30 |
| 17 | Copy 14, 4 | ⇒ | NCOPY 3 |
| 21 | Add 2, 11 | ⇒ | ADD1 0x31 0x31 |
| 23 | Copy 10, 4 | ⇒ | SAME_NCOPY 13, 2 |
| 27 | Copy 10, 4 | | |

(그림 3) wdiff의 인코딩

③ 대상 문자열 13번지에서 읽어오는 패턴 “0000”은 인덱스에 에 없으므로 추가해야 한다. 그러나 추가문자열의 원소가 모두 같으므로 RUN으로 판정되고 “Run 0, 4”로 인코딩된다.

④ 대상 문자열 17번지에서 읽어오는 패턴 “5678”은 인덱스에 존재하고 그 체인으로 4, 14를 가지고 있다. 연계 매치는 존재하지 않으므로 쓰기 주소와 가까운 14가 선택되며 COPY로 판정하고 “Copy 14, 4”로 인코딩된다.

⑤ 대상 문자열 21번지에서 읽어오는 패턴 “1112”는 인덱스에  없으므로 추가해야 한다. 그러나 23번지 이후로는 패턴 “1234”는 인덱스에 존재하므로 “11”에 대해서만 추가한다. 따라서 ADD로 판정되고, “Add 2, 11”로 인코딩하게 된다.

⑥ 대상 문자열 23번지에서 읽어오는 패턴 “1234”는 인덱스에 존재하고 연계 매치가 없으므로 가장 근접한 체인인

10이 선택되어지고 COPY로 판정되고, “Copy 10, 4”로 인코딩 된다.

⑦ 대상 문자열 23번지에서 읽어오는 패턴 “1234”는 인덱스에 존재하고 연계 매치가 없으므로 가장 근접한 체인인 10이 선택되어지고 COPY로 판정되고, “Copy 10, 4”로 인코딩 된다.

이와 같이 초벌 델타 명령이 생성되었다면, 다음과 같이 최적화 작업을 거쳐 최종적으로 인코딩하게 된다.

① “MATCH 4”에 대해서는 매치의 길이가 16이하이므로 MOV 명령군을 사용한다. “MOV3”으로 축약하여 인코딩되며 1바이트가 소요된다.

② “Copy 8, 9”는 매치의 길이가 9이고 패턴의 위치가 8이다. 그리고 현재 쓰기 주소는 4이므로 인코딩 명령은 XPCOPY1으로 변경되고 참조주소는 상대주소인 4로 변경되어 “XCOPY1 4, 9”로 인코딩하게 된다.

③ “Run 0, 4”는 반복 문자가 4이므로 인수를 생략한 “RUN 0x30”으로 인코딩된다.

④ “Copy 14, 4”는 매치의 길이가 4이고 패턴의 위치는 14이다. 그리고 현재 쓰기 주소는 17이므로 인코딩 명령은 NCOPY로 변경되고, 참조주소는 상대주소인 3으로 변경되어 “NCOPY 3”으로 인코딩하게 된다.

⑤ “Add 2, 7”은 추가 문자열이 1～16이내이므로 인수를 생략한 “ADD1 0x31 0x31”로 인코딩된다.

⑥ 쓰기주소 23과 27의 “Copy 10, 4”는 같은 Copy 명령의 연속이므로 SAME_COPY 명령군으로 변경된다. 쓰기주소는 23이고 참조주소는 10이므로 상대참조주소는 13이 되어 결과적으로 “SAME _NCOPY 13, 2”로 인코딩하게 된다.

위와 같은 절차를 통해 인코딩된 델타 패키지는 총 14바이트를 소요하게 된다.

(그림 4)는 wdiff와 vcdiff의 인코딩 과정을 비교한 것으로 vcdiff에서는 13바이트가, wdiff는 11바이트가 소요되는 것을 알 수 있다. wdiff는 MATCH 판정을 이용하여 참조 주소 인수를 생략하였고, 적은 바이트를 차지하는 ADD에 대해서 ADD0～ADDF 명령으로 세분화하여 바이트 수를 나타내는 인수를 생략하였다. 그리고 4바이트 RUN 명령에 대하여 반복 횟수 인수를 생략하였다. 이와 같이 델타 명령의 확장과 세분화를 통해 압축률을 향상시킬 수 있었다.

| 데 이 터 | S: abcdefghijklmnop<br>T: abcdwxyzefghefghefghefghzzzz |
|---|---|

1. wdiff

| 초벌 델타 명령 | 최적화된 델타 명령 | 최적화된 인코딩 |
|---|---|---|
| MATCH 4 | MOV3 | 0x13 |
| ADD 4 ,wxyz | ADD3 wxyz | 0x33 0x77 0x78 0x79 0x7A |
| COPY 4 4 | SAME_NCOPY 4, 4 | 0x57 0x04 0x04 |
| COPY 4 4 | | |
| COPY 4 4 | | |
| COPY 4 4 | | |
| RUN 4 , z | RUN z | 0x05 0x7A |

2. vcdiff

| Delta Inst. | Plain | Optimized |
|---|---|---|
| COPY 4, 0 | 19 4 0 | 20 0 |
| ADD 4, wxyz | 1 4 wxyz | 172 wxyz 4 |
| COPY 4, 4 | 19 4 4 | |
| COPY 12, 24 | 19 12 24 | 28 24 |
| RUN 4, z | 0 4 z | 0 4 z |

(그림 4) wdiff와 vcdiff[3]의 인코딩 과정 비교

## 5.2 디코딩

델타 추출 및 인코딩 과정을 통해 델타 압축된 패키지는 디코딩 과정을 통해서 대상 파일과 동일한 파일로 복원된다. 디코딩

과정에서도 델타 압축에서 사용한 명령 코드 테이블을 이용하며, (그림 6)과 같이 델타 명령과 인수를 1:1로 대응하여 수행한다.

디코딩 작업은 열악한 환경에서 동작 가능한 것을 보장해야 한다. 따라서 본 논문은 메모리를 크게 할당해야 하는 특별한 자료 구조를 사용하지 않고 파일의 바이너리를 하나씩 읽어서 디코딩할 수 있도록 델타 명령들을 세분화하고 고정된 명령 형식을 사용한다. 델타 명령은 1바이트 크기의 코드로 열거형 타입에 대응되므로 특별한 과정 없이 디코딩할 수 있다. 그러므로 현재 명령 코드와 인수에 따라서 MOV와 COPY 명령군은 원본(구 버전) 파일에서 데이터를 읽어오고 ADD 명령군의 경우 델타 패키지에 포함되어 있는 데이터를 출력 파일에 기록하는 간단한 절차로 디코딩할 수 있다.

델타 패키지에 대한 디코딩 과정은 델타

| 원본 데이터 | 12345678901234567890 |
|---|---|

| 델타 패키지의 내용 | | | | | | |
|---|---|---|---|---|---|---|
| 0x13 | 0x52 | 0x04 | 0x09 | 0x05 | 0x30 | 0x51 |
| 0x03 | 0x31 | 0x31 | 0x31 | 0x57 | 0x0D | 0x02 |

| 바이너리 | 디코딩 내역 | 상세 내역 |
|---|---|---|
| 0x13 | MOV3 | 원본으로부터 “1234”를 읽어서 출력파일에 기록한다. |
| 0x52 | XPCOPY1 | 쓰기주소가 8이므로 실 참조주소는 8 − 4 = 4이고 매치길이가 9이므로, 원본파일의 4번지의 “567890123”을 읽어서 출력파일에 기록한다. |
| 0x04 | 상대참조주소 : 4 | |
| 0x09 | 매치 길이 : 9 | |
| 0x05 | RUN | “0”을 출력파일에 4회 기록한다. |
| 0x30 | 데이터 : 0x30 | |
| 0x51 | NCOPY | 쓰기주소가 17이므로 실 참조주소는 10이다. 따라서 원본파일의 14번지의 “5678” 읽어 출력파일에 기록한다. |
| 0x03 | 상대참조주소 : 3 | |
| 0x31 | ADD1 | 출력파일에 “11”을 기록한다. |
| 0x31 | 데이터 : 1 | |
| 0x31 | 데이터 : 1 | |
| 0x57 | SAME_XNCOPY | 쓰기주소가 23이므로 실참조주소는 10이다. 따라서 원본파일의 10번지에 있는 “1234”를 2회 출력파일에 기록한다. |
| 0x0D | 상대참조주소 : 13 | |
| 0x02 | 반복횟수 : 2 | |

(그림 5) wdiff의 디코딩 과정

```
while (1) {
  델타명령 = 델파 파일에서 1바이트 바이너리 읽기;
  if (델타명령 == EOF) then break;
  switch (델타명령) {
  case XMOVEXX:
    XMOVEXX로 디코딩;
    break;
  case XMOVEX:
    XMOVEX으로 디코딩;
    break;
  case MOV0 ~ MOVF:
    MOV명령군으로 디코딩;
    break;
  case XMOV0 ~ XMOVF:
    XMOV명령군으로 디코딩;
    break;
  case ADD0 ~ ADDF:
    ADD명령군으로 디코딩;
    break;
  case XADD0 ~ XADDF:
    XADD명령군으로 디코딩;
    break;
  case PCOPY, NCOPY, SAME_PCOPY, SAME_NCOPY:
    가상참조주소를 실참조주소로 변환;
    if (델타명령 == SAME_COPY명령군) {
      반복횟수적용하여 각 SAME_P/NCOPY으로 디코딩
    } else {
      P/NCOPY로 디코딩
    }
    break;
  case XPCOPY1, XNCOPY1, SAME_XPCOPY1, SAME_XNCOPY1:
    가상참조주소를 실참조주소로 변환;
    복사할 바이트 읽기;
    if (델타명령 == SAME_COPY명령군) {
      반복횟수적용하여 각 SAME_P/NCOPY1으로 디코딩
    } else {
      P/NCOPY1으로 디코딩
    }
    break;
  case XPCOPY2, XNCOPY2, SAME_XPCOPY2, SAME_XNCOPY2:
    플래그 해독;
    가상참조주소를 실참조주소로 변환;
    복사할바이트 읽기;
    if (델타명령 == SAME_COPY명령군) {
      반복횟수적용하여 각 SAME_P/NCOPY2로 디코딩
    } else {
      P/NCOPY2로 디코딩
    }
    break;
  case RUN:
    RUN으로 디코딩;
    break;
  case XRUN0 ~ XRUNF:
    XRUN으로 디코딩;
  default:
    오류 메시지 출력;
}
```

(그림 6) 디코딩 과정

패키지를 읽어서 그 바이너리를 명령 코드 테이블에 대응하는 방식으로 수행한다. (그림 5)는 (그림 3)의 예제에서 wdiff에 의해 인코딩된 델타 패키지를 실제로 디코딩하는 과정을 나타내고 있으며 그 절차는 다음과 같다.

① 델타 패키지의 처음 바이너리 0x13은 MOV3 명령이고 추가적인 인수는 없다. 그리고 MOV3 명령은 원본과 대상이 1:1로 매치되는 부분이므로 원본 파일에서 4바이트를 읽어서 출력파일에 기록하게 된다.

② 바이너리 0x52는 XPCOPY1 명령이고 2개의 인수를 사용하므로 델타파일에서 2바이트를 더 읽어온다. 0x04는 상대참조주소이고, 0x09은 매치의 길이를 나타낸다. 상대참조주소는 쓰기주소가 8이므로 8 − 4 = 4가 실 참조주소가 되고, 따라서 원본파일의 4번지에서 9바이트(“567890123”)를 읽어서 출력파일에 기록하게 된다.

③ 바이너리 0x05는 RUN 명령이다. RUN 명령은 해당 문자 하나를 4번 반복하는 명령이므로 반복할 문자를 나타내는 인수가 필요하다. 따라서 델타 패키지에서 데이터를 가져오기 위해 한 바이트만 더 읽게 된다. 그리고 읽어 온 데이터 z를 출력 파일에 4번 반복한다.

④ 바이너리 0x51은 NCOPY이고 1개의 인수를 사용하므로 델타파일에서 1바이트를 더 읽어온다. 0x03는 상대참조주소를 나타낸다. 상대참조주소는 쓰기주소가 17이므로 17 − 3 = 14이 실 참조주소가 되고, 따라서 원본파일의 14번지에서 4바이트(“5678”)를 읽어서 출력파일에 기록하게 된다.

⑤ 바이너리 0x31은 ADD1을 나타내고 2바이트의 추가문자열을 인수로 사용하므로 델타파일에서 2바이트를 더 읽어온다. 출력파일에는 읽어온 2바이트 추가문자열 “11”이 기록된다.

⑥ 바이너리 0x57은 SAME_NCOPY이고 2개의 인수를 사용하므로 델타파일에서 2바이트를 더 읽어온다. 0x13는 상대참조주소이고, 0x02는 반복횟수를 나타낸다. 상대참조주소는 쓰기주소가 23이므로 23 − 13 = 10이 실 참조주소가 되고, 따라서 원본파일의 10번지에서 4바이트(“5678”)를 읽어서 출력파일에 기록하게 된다.

이러한 과정을 통해 출력 파일은 대상 문자 열(그림 3)과 같은 “12349012345670000 56781112341234”로 복원된다.

## 6. 실험

본 장에서는 wdiff와 다른 유사 도구들과의 압축률을 비교하고 분석한다. 실험을 위해 사용된 장비의 CPU는 intel Pentium 4 프레스캇 − 3.0GHz이며, 메인 메모리는 1GB이다. 측정을 위해 사용한 프로그램은 (표 4)와 같다. MiBench[4]의 몇 가지 프로그램과 간단한 응용 프로그램 그리고 본 논문의 방식으로 구현한 wdiff를 사용하였다.

(표 4) 실험에 사용된 프로그램 목록

| 번호 | 프로그램명 | 옵션 | 비고 |
|---|---|---|---|
| 1 | math | small 버전 → large 버전 | MiBench |
| 2 | bitcnts | 최적화 옵션 O0 → O3 | MiBench |
| 3 | qsort | small 버전 → large 버전 | MiBench |
| 4 | dijkstra | 최적화 옵션 O0 → O3 | MiBench |
| 5 | susan | 최적화 옵션 O0 → O3 | MiBench |
| 6 | susan | 최적화 옵션 O0 → O4 | MiBench |
| 7 | qsort | large 버전 → small 버전 | MiBench |
| 8 | adpcm | rawcaudio.arm → rawdaudio.arm | MiBench |
| 9 | patricia | 최적화 옵션 O0 → O3 | MiBench |
| 10 | FFT | 최적화 옵션 O0 → O3 | MiBench |
| 11 | CRC | 최적화 옵션 O0 → O3 | MiBench |
| 12 | rijndael | 최적화 옵션 O0 → O3 | MiBench |
| 13 | sha | 최적화 옵션 O0 → O3 | MiBench |
| 14 | Line Editor | 최적화 옵션 O0 → O3 | 일반응용 |
| 15 | wdiff | v.0.9a → v.0.9b (테스트 버전) | 일반응용 |

비교 대상으로는 xdelta[5]와 jojo's diff(jdiff)[6]를 사용하였다. xdelta는 리눅스에서 널리 쓰이고 있으며, rsync 알고리즘을 사용하고 표준화된 vcdiff 형식의 복사 기반 인코딩을 사용하는 바이너리 델타 압축 도구이다. 그리고 jdiff도 복사기반방식의 또 다른 도구이다. 최악 경우에 델타 압축보다 데이터 압축이 유리한 경우를 알아보기 위해 데이터 압축 도구인 RAR32를 사용하였다.

(표 5)와 (표 6)은 실험 결과를 나타내고 있다. (표 5)의 Source 컬럼은 압축되지 않은 원본 파일의 용량을, Target 컬럼은 압축되지 않은 대상 파일의 용량을, wdiff, jdiff, xdelta -0 컬럼은 각 델타 압축 도구에 의해 생성된 델타 패키지의 용량을, RAR32 컬럼은 RAR32를 사용하여 데이터 압축된 대상 파일의 용량을 나타낸다. (표 5)의 wdiff, jdiff, xdelta -0 컬럼은 각 델타 압축 도구에 의해 생성된 델타 패키지의 압축률을, RAR32 컬럼은 RAR32를 사용하여 데이터 압축된 대상 파일의 압축률, 그리고 wdiff vs. 컬럼은 각 도구 압축률과 wdiff의 압축률의 비교한 것이다. 이 중 굵은 글씨로 표시된 것은 가장 좋은 압축률을 보인 것을 나타낸다.

(표 5), (표 6)의 실험 결과와 같이, wdiff가 최악 경우를 제외하고는 0.01~17.39% 우세한 것을 알 수 있다. 그러나 최악 경우에서는 다른 유사 도구들에 비하여 낮은 압축률을 보이고 있다. 그 원인은 델타 추출 및 인코딩 정책에서 찾을 수 있다. wdiff는 최대한 참조 데이터를 이용하여 데이터를 표현하므로 원본과 대상 차이가 커서 바이너리의 유사도가 떨어질 때 멀리 떨어진 작은 크기의 바이트를 복사해오는 경우가 자주 발생될 수 있다. 이 때, COPY 명령은 큰 주소를 나타내기 위해 인수가 커지게 되고 따라서 압축률을 향상시키기 어렵다. 다른 유사 도구들은 그러한 경우에는 멀리 떨어지거나 작은 매치에 대하여 ADD 판정을 하는 경우가 많다. 그러나 이러한 정책이 반드시 좋지는 않다. 이와 같은 상황을 ADD로 판정하는 것이 최악 경우에서는 좋을 수 있으나 두 버전 간의 파일의 유사도가 클 경우에는 COPY 보다 ADD가 많아지는 경향을 보이므로 압축률

(표 5) wdiff와 다른 유사 도구들과의 압축 용량 비교

단위 : bytes

| | Source | Target | **wdiff** | jdiff | xdelta3 -0 | rar32 |
|---|---|---|---|---|---|---|
| 1 | 1,023,131 | 1,022,555 | **34,853** | 35,863 | 62,136 | 262,449 |
| 2 | 15,536 | 15,536 | **1,600** | 2,120 | 3,335 | 4,367 |
| 3 | 11,322 | 11,915 | **2,548** | 3,737 | 4,620 | 3,214 |
| 4 | 12,653 | 12,653 | **24** | 38 | 224 | 3,850 |
| 5 | 43,761 | 43,761 | **13** | 19 | 188 | 10,837 |
| 6 | 43,761 | 35,707 | 32,782 | **25,008** | 27,687 | **15,368** |
| 7 | 11,915 | 11,322 | **2,548** | 3,002 | 4,203 | 10,837 |
| 8 | 952,984 | 952,968 | **23,212** | 26,989 | 52,496 | 243,365 |
| 9 | 14,973 | 14,273 | **4,736** | 5,922 | 7,191 | **4,615** |
| 10 | 30,345 | 30,963 | **6,251** | 8,424 | 10,241 | 10,031 |
| 11 | 11,361 | 11,361 | **874** | 1,042 | 1,885 | 4,080 |
| 12 | 46,854 | 48,016 | 33,701 | 19,841 | **18,932** | **18,092** |
| 13 | 12,429 | 12,429 | **11** | 17 | 177 | 3,470 |
| 14 | 16,482 | 17,133 | **6,669** | 7,748 | 9,340 | 5,600 |
| 15 | 581,685 | 581,685 | **42,367** | 60,484 | 93,839 | 116,885 |

(표 6) wdiff와 다른 유사 도구들과의 압축률 비교

| | **wdiff** | jdiff | xdelta3 -0 | rar32 | wdiff vs. | | |
|---|---|---|---|---|---|---|---|
| | | | | | jdiff | xdelta | rar32 |
| 1 | **96.59%** | 96.49% | 93.92% | 74.33% | 0.10% | 2.67% | 22.26% |
| 2 | **89.70%** | 86.35% | 78.53% | 71.89% | 3.35% | 11.17% | 17.81% |
| 3 | **78.62%** | 68.64% | 61.23% | 73.03% | 9.98% | 17.39% | 5.59% |
| 4 | **99.81%** | 99.70% | 98.23% | 69.57% | 0.11% | 1.58% | 30.24% |
| 5 | **99.97%** | 99.96% | 99.57% | 75.24% | 0.01% | 0.40% | 24.73% |
| 6 | 8.19% | **29.96%** | 22.46% | **56.96%** | -21.77% | -14.27% | -48.77% |
| 7 | **77.50%** | 73.49% | 62.88% | 4.28% | 4.01% | 14.62% | 73.21% |
| 8 | **97.56%** | 97.17% | 94.49% | 74.46% | 0.40% | 3.07% | 23.10% |
| 9 | **66.82%** | 58.51% | 49.62% | **67.67%** | 8.31% | 17.20% | 0.85% |
| 10 | **79.81%** | 72.79% | 66.93% | 67.60% | 7.02% | 12.89% | 12.21% |
| 11 | **92.31%** | 90.83% | 83.41% | 64.09% | 1.48% | 8.90% | 28.22% |
| 12 | 29.81% | 58.68% | **60.57%** | **62.32%** | -28.87% | -30.76% | -32.51% |
| 13 | **99.91%** | 99.86% | 98.58% | 72.08% | 0.05% | 1.34% | 27.83% |
| 14 | **61.08%** | 54.78% | 45.49% | **67.31%** | 6.30% | 15.59% | -6.24% |
| 15 | **92.72%** | 89.60% | 83.87% | 79.91% | 3.11% | 8.85% | 12.81% |

향상에 도움이 되지 않을 수도 있다. 그리고 최악 경우에는 두 버전 간의 차이가 크므로 델타 압축을 하는 것보다 대상 파일을 데이터 압축하는 것이 더 유리할 수도 있다. 따라서 본 논문은 최악 경우의 시점을 55%로 선정하고 압축률이 낮은 경우에는 RAR32를 이용하여 데이터 압축을 하도록 하였다.

## 7. 결론 및 향후과제

본 논문은 기존의 복사 기반 방식의 바이너리 패치를 세분화하고 확장한 CAM(Copy-Add-Move) 방식을 새로 도입하여 인코딩에 사용되는 인수를 최소화함으로써 압축률을 향상시켰으며, 델타 압축을 하는 두 버전 간의 차이가 클 경우에는 델타 압축이 비효율적이라는 것을 밝히고 그 대안으로써 대상 파일을 데이터 압축하도록 하였다. 그리고 명령 코드 테이블에 1:1로 대응할 수 있는 간단한 디코딩 과정을 통해 열악한 임베디드 환경에서도 원활히 동작할 수 있을 것으로 기대한다.

향후 연구 방향으로 차이점 추출을 위해 패턴을 찾는 과정에서 그 개선점을 찾을 수 있을 것이다. 많은 바이너리 패치 도구들이 윈도우를 교체할 때마다 인덱스를 새로 만들고 있다. 따라서 윈도우 밖에 있는 패턴을 복사해올 수 없다. 그러나 파일 전체에 대한 인덱스를 메모리상에서 운용한다는 것은 사실상 불가능하다. 따라서 일정 크기의 고정된 인덱스를 사용하고 그 인덱스는 윈도우가 바뀌어도 다시 생성하지 않는다. 그리고 LRU(Least Recently Used) 알고리즘[7]을 사용하여 가장 오랫동안 사용되지 않은 인덱스 노드를 선택하여 교체하는 방식으로 운용한다면 파일 내의 모든 패턴을 찾을 수 있을 것이다. 따라서 보다 많은 패턴을 찾을 수 있게 되므로 압축률도 향상시킬 수 있을 것이다.

또한 인코딩 시 주소 표현을 개선하여 압축률을 향상 시킬 수 있다. wdiff는 현재 명령에 의해 쓰일 주소에 대해서 상대적인 가상 참조 주소를 사용하고 있다. 이보다 작은 크기로 표현될 수 있는 주소 표현 방식을 찾을 수 있다면 더 큰 효과를 기대할 수 있을 것이다. 그러나 너무 많은 주소 체계를 쓰면 델타 명령이 세분화 되어야 하므로 오히려 해가 될 수 있다. 따라서 그 타협점을 찾는 것이 관건이라고 생각한다.

## 참 고 문 헌

[1] MOUNT10 DATA CONTINUITY ARCHITECTS, White Paper “FastBIT™ Binary Patching vs. Block Technology”, Release : 1.0, October 2004.

[2] D. Trendafilov, N. Memon, and T. Suel. “zdelta: An Efficient Delta Compression Tool”, Technical Report TR-CIS-2002-02, Polytechnic University, CIS Department, June 2002.

[3] David G. Korn and Kiem-Phong Vo, “Engineering a Differencing and Compression Data Format”, USENIX, 2002.

[4] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “Mibench : A free, commercially representative embedded benchmark suite”. In IEEE 4th Annual Workshop on Workland Characterization, Austin, TX, Dec. 2001.

[5] J. P. MacDonald, “File System Support for Delta Compression, Master's Thesis”, University of California at Berkeley, 2000.

[6] Joris Heirbaut, JojoDiff - diff utility for binary files, http://jojodiff.sourceforge.net/
[7] A. Silberschatz, P. B. Galvin and G. Gagne, OPERATING SYSTEM CONCEPTS, 6th Edition, JOHN WILEY & SONS, INC., 2003.

이 민 재

2005년 한국산업기술대학교 컴퓨터 공학과졸업(학사).
2005년~현재 한국산업기술대학교 석사과정.
관심분야 : 프로그래밍 언어, 소프트웨어공학

한 경 숙

홍익대학교 컴퓨터공학과 졸업 (학사, 석사, 박사).
1995년~1997년 삼성전자㈜ 반도체사업부 주임연구원.
2000년~2003년 ㈜참좋은인터넷 부사장.
주요추진실적은 정보가전용 RTOS 커널 및 도구 기술 개발 과제 중 내부 모듈 개발,
관심분야 : 컴파일러 최적화, 프로그래밍 언어.

우 덕 균

홍익대학교 컴퓨터공학과 졸업 (학사, 석사, 박사).
2001년~2006년 한국전자통신연구원 선임연구원.
2006년~현재 한국전자통신연구원 S/W개발도구연구팀 팀장.
관심분야 : 임베디드 시스템, 개발 도구, 컴파일러