KIPS Trans. Softw. and Data Eng.
Vol.12, No.5 pp.207~216
ISSN: 2287-5905 (Print), ISSN: 2734-0503 (Online)

https://doi.org/10.3745/KTSDE.2023.12.5.207

Proposal for Decoding-Compatible Parallel Deflate Algorithm by Inserting Control Header Composed of Non-Compressed Blocks

Kim Jung Hoon†

ABSTRACT

For decoding-compatible parallel Deflate algorithm, this study proposed a new method of the control header being made in such a way that essential information for parallel compression and decompression are stored in the Disposed Bit Area (DBA) of the non-compression block and being inserted into the compressed blocks. Through this, parallel compression and decompression are possible while maintaining perfect compatibility with the existing decoder. After applying this method, the compression time was reduced by up to 71.2% compared to the sequential processing method, and the parallel decompression time was reduced by up to 65.7%. In particular, it is well known that parallel decompression is impossible due to the structural limitations of the Deflate algorithm. However, the decoder equipped with the proposed method enables high-speed parallel decompression at the algorithm level and maintains compatibility, so that parallelly compressed data can be decoded normally by existing decoder programs.

Keywords : Parallel Deflate Compression, Parallel Deflate Decompression, Deflate Algorithm, NPNCB block, Legacy Compatible Decompression

비 압축 블록으로 구성된 제어 헤더 삽입을 통한 압축 해제 호환성 있는 병렬 처리 Deflate 알고리즘 제안

김 정 훈†

요 약

본 연구에서는 압축 해제 호환성을 갖춘 병렬 처리 Deflate 압축 알고리즘을 구현하기 위하여 병렬 압축 및 압축 해제에 필수적인 정보를 복수의 비 압축 블록(Non-Compression Block)내의 버려지는 영역(Disposed Bit Area)에 저장하는 방식으로 구성한 컨트롤 헤더를 삽입하는 새로운 방식을 제안하였다. 이를 통해 기존 압축 해제 프로그램과 완벽한 호환성을 유지하면서도 병렬 압축 및 병렬 압축 해제가 가능하도록 하였다. 또한 순차 처리방식 대비 압축 시간을 최대 71.2% 절감하였고 병렬 압축해제 시간을 65.7%까지 절감하였다. 특히 Deflate 알고리즘의 구조적 제약으로 인해 병렬 압축 해제는 불가능하다고 알려져 있으나, 제안하는 방식을 탑재한 디코더로 알고리즘 수준에서 고속의 병렬 압축 해제가 가능하고, 호환성을 유지하여 동일한 압축 데이터를 기존의 압축 해제 프로그램으로도 정상적 압축 해제가 가능함을 확인하였다.

키워드 : 병렬 Deflate 압축, 병렬 Deflate 압축 해제, Deflate 알고리즘, NPNCB 블록, 압축 해제 호환

1. 서 론

Deflate 압축 알고리즘은 전 세계적으로 활용되고 있는 ZIP 포맷의 핵심 압축 알고리즘이며[1], .apk, .pdf, .xlsb 등 다양한 파일 형식에서 내부 데이터 압축 용도로 활용되고 있다. Deflate 압축 알고리즘의 특성상 압축 과정의 경우, 연산 부담이 큰 과정을 병렬 처리하여 고속 압축을 지원하는 방법 또는 이러한 방법을 통해 GPU의 연산 능력을 효율화하는 방법이 개발되어 있다[1].

하지만 압축 해제 과정에서는 알고리즘의 특성상 근본적으로 임의의 위치에서 압축 해제를 할 수 없기 때문에 대규모 데이터를 병렬 압축 해제할 수 있는 알려진 방법이 없었다[2]. 임의의 위치에서 압축해제가 불가능 한 이유로는 먼저 해제된 압축 데이터를 참조하면서 현재 압축 데이터의 압축 해제를 진행하는 Deflate 알고리즘 자체의 구조적 특징[3]이기 때문으로 여겨진다. 현재 국내 상용 프로그램인 '반디집'에 적용된 병렬 압축 및 병렬 압축 해제 방법은 Fig. 1과 같이 복수의 파일에 대해서 독립적인 스레드가 각 파일을 압축하고 압축 해제하는 방식으로 병렬 처리를 구현하고 있다고 한다[4].

그러나 이러한 방식의 경우 충분한 크기를 가진 여러 개의 파일이 있어야 병렬 압축의 효과가 나타날 수 있고, 동일 파

† 정 회 원 : 바이너리랩 주식회사 대표이사
Manuscript Received : December 2, 2022
First Revision : January 30, 2023
Second Revision : February 2, 2023
Accepted : February 3, 2023
* Corresponding Author : Kim Jung Hoon(powerzenith@naver.com)

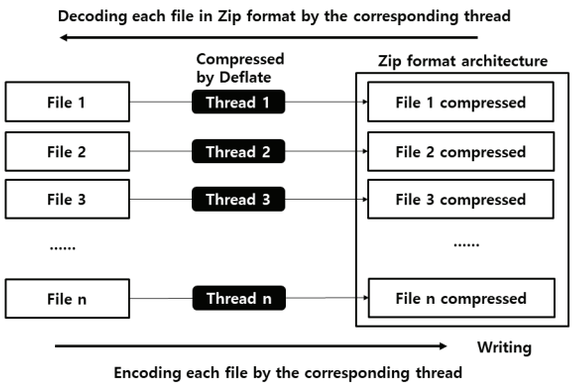

Fig. 1. General Parallelism of Deflate Compression and Decompression

일 내에서도 압축 과정에서 병렬 처리가 가능해야 더 빠른 압축 및 압축 해제가 가능하다는 한계점이 있다.

본 연구에서는 일반적인 Deflate 압축 알고리즘의 병렬처리 방법에 있어서 알고리즘의 구조적인 한계를 극복하여 병렬 처리를 구현하기 위하여 압축 블록중의 하나인 NPNCB (No Payload Non-Compressed Block)을 이용하여 알고리즘 수준에서의 병렬처리 기법을 구현하였다. 제안하는 방법은 기존 디코더와 호환성을 유지하면서 동일 파일 또는 데이터 내에서 병렬 처리 가능한 방식으로서 고도의 연산 과정이 필요한 부분을 따로 병렬 처리[1]하거나, 복수의 파일 각각을 병렬 처리하는 방식[4]과 구분 된다. 또한 Deflate 압축 알고리즘의 압축 해제 시의 병렬 처리에 관한 구조적 제약 사항[2,3]을 극복하여 압축 원본 데이터 자체를 분할하여 병렬 압축하고 병렬 압축 해제가 가능하게 하면서도, 호환성을 유지하여 압축 결과를 기존 디코더에서도 압축 해제 될 수 있게 하였다.

2. 관련 연구

2.1 특수한 유형의 비 압축 블록 - NPNCB 개요

NPNCB는 Deflate 알고리즘에서 사용하는 비 압축 블록(non-compressed block)의 한 유형이며[5], 빈 저장 블록(empty stored block)이라고도 한다[7]. 바이트 경계에서 압축 블록이 시작되는 경우 그 구조는 Fig. 2와 같다[5]. SSH 전송계층 프로토콜의 세부 개념인 이진 패킷 프로토콜(Binary packet protocol)에서는 Zlib 엔코더가 빈 저장 블록을 현재까지 생성한 압축 블록들에 이어서 추가한 다음, 압축 결과로 출력하는 기능을 수행한다[6]. 한편 압축 해제 과정에서 NPNCB를 만났을 때, NPNCB 안에는 원본 데이터의 복원과 관련한 리터럴 데이터가 존재하지 않기 때문에 디코더의 압축 해제 데이터에 영향을 주지 않는다. 따라서 기존 디코더로도 압축 해제 과정이 정상적으로 수행된다[5].

2.2 비 압축 블록의 DBA(Disposed Bit Area) 영역

Deflate 알고리즘은 원본 데이터의 특성에 따라 3가지 종류의 압축 데이터 블록들을 연속적으로 형성하면서 압축 데이터를 만든다. 3가지 압축 블록은 동적 허프만 코드(dynamic Huffman code)생성 방식이 적용된 다이내믹 압축 블록(dynamic compressed block), 고정 허프만 코드(fixed Huffman code) 생성 방식이 적용된 고정 압축 블록(fixed compressed block), 비 압축 블록(non-compressed block) 으로 구분된다[8]. 각각의 압축 블록은 원본 데이터 특성에 따라 바이트내의 어떠한 비트 위치에서도 시작하거나 끝날 수 있다[8].

비 압축 블록 내에는 해당 비 압축 블록이 마지막 압축 블록 인지를 나타내는 1비트 영역(BFINAL)과 블록의 종류를 나타내는 2비트 영역(BTYPE)으로 구성된 3비트의 블록 헤더가 존재한다. 헤더를 지나면 압축 해제 시에 복원되는 원본 데이터(이하, 리터럴 데이터)의 크기를 나타내는 2바이트의 영역(LEN)이 존재하는데, LEN이 바이트의 경계에서 반드시 시작하도록 하기 위해(이하, 바이트 경계 정렬) 특별한 의미가 존재하지 않는 가변적인 영역이 필요하다. 이 영역을 DBA(Disposed Bit Area)라고 한다[5].

Fig. 3은 'Previous Byte'의 MSB(Most Significant Bit)에서 직전 압축 블록이 끝나고, 'Byte 1'의 LSB(Least Significant Bit)에서 다음 압축 블록으로서 비 압축 블록이 시작되는 모습을 나타낸다[5]. 'Byte 1'의 LSB 에서는 현재 생성중인 비 압축 블록이 마지막 블록인지 여부(BFINAL)를 나타내며, MSB(Most Significant Bit) 방향으로 2비트는 블록의 타입 값(BTYPE)으로서 '00'으로 되어 있는데, 이것은 현재 블록이 비 압축 블록임을 나타낸다. 계속하여 MSB 방향으로 5비트는 'x'로 표기했는데 실무적으로 '0'으로 채워져 있으며 어떠한 값이 들어가도 무방하다는 의미이다.

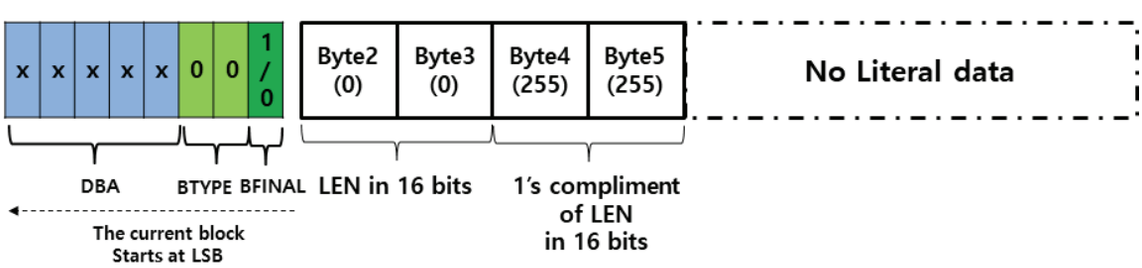

Fig. 2. The Structure of No-payload Non-compressed Block

한편, Fig. 4와 같이 직전 압축 블록의 끝이 바이트의 중간에 끝나는 경우, 비 압축 블록이 바이트내의 어느 특정 비트에서 시작하기 때문에 바이트 경계 정렬을 위한 DBA의 크기는 달라질 수 있다[5].

Fig. 5는 3비트의 블록 헤더가 바이트 경계에서 끝나는 경우로서, 이미 바이트 경계 정렬이 달성되었으므로 DBA가 필요하지 않는 경우이다. 즉 직전 압축 블록이 특정한 바이트의 5개의 비트를 사용하고 종료될 경우, 남은 3비트에 비 압축 블록인 다음 블록의 헤더 정보가 들어가므로 LEN은 바이트 경계에서 시작하게 되어 DBA가 필요 없다[5,8].

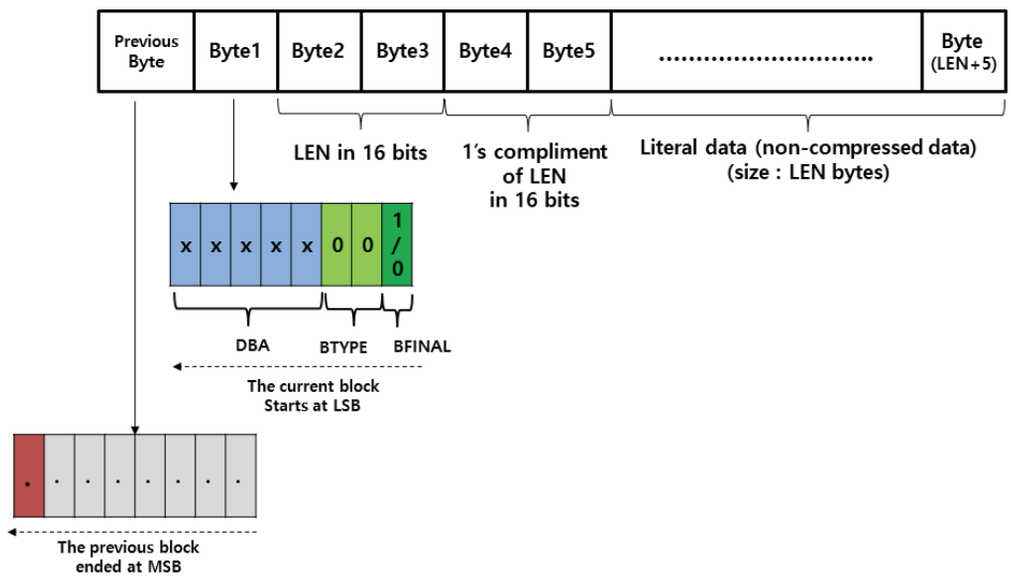

Fig. 3. Structure of Non-compressed Block when Starting at the Byte Boundary in the Deflate Algorithm

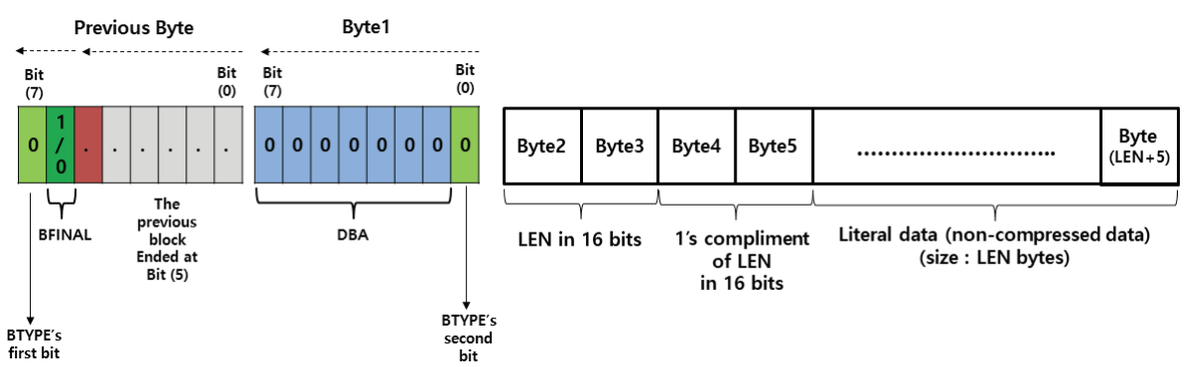

Fig. 4. The Example Structure Case of Non-compressed Block with Variable Size DBA when Starting in the Byte in the Deflate Algorithm

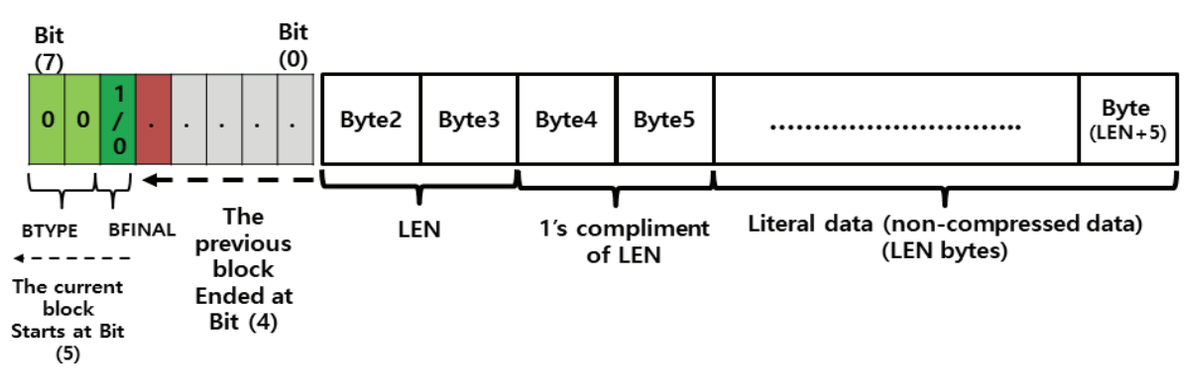

Fig. 5. There is no DBA Structure Case of Non-compressed Block for Block Starting at Bit (5)

DBA에는 현재 표준 및 실무상 어떠한 값이 들어가도 Deflate 알고리즘을 구현한 디코더들은 이를 무시(ignored)한다고 RFC 1951 표준에 명시되어 있으며[8], 실제로 DBA에 임의의 값을 삽입하여도 디코딩에서 오류가 없음도 알려져 있다[5]. DBA에 임의의 값이 들어갈 수 있다는 점에서 Deflate 알고리즘에 대한 NPNCB 주입 취약점이 있음도 알려져 있다[5]. 그러나 NPNCB 주입 취약점은 적절한 정보 보호 도구로 방어가 가능하고 디코더로 하여금 표준 압축 알고리즘의 작동을 방해하지 않으면서 부가적인 작동을 할 수 있도록 한다는 점에서 NPNCB 활용을 통한 이익이 비용보다 크다고 여겨진다.

3. 제안 방법

3.1 병렬 처리 엔코더 개요

제안하는 병렬처리 엔코더는 Fig. 6와 같이 원본 데이터를 적절한 크기로 분할하는 과정을 가장 먼저 수행한다. 원본 데이터를 모두 동일한 크기로 분할한 각각의 데이터를 세그먼트(segment)라고 명명하였다. 원본 데이터의 크기에 따라 마지막 세그먼트의 크기는 동일하지 않을 수 있다.

다음으로 Fig. 7과 같이 기존 디코더로도 압축을 풀 수 있도록 연속하는 NPNCB 블록들로 구성된 병렬 압축 해제용

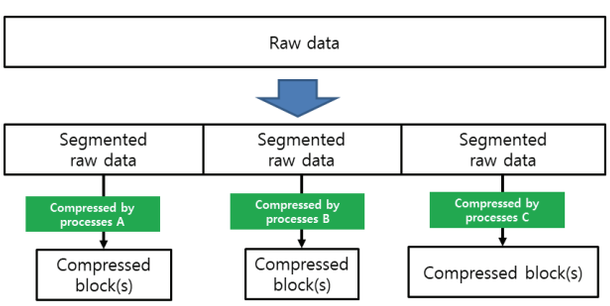

Fig. 6. The Summary of the Initial Step of Raw Data Segmented and Compressed by Parallel Processes

제어 헤더(control header)를 삽입하고, 각 세그먼트에 대한 압축 결과로 생성한 압축 블록들에 이어서 NPNCB를 연결한다. 이렇게 연결 용도로 사용한 NPNCB를 본 연구에서는 연결 꼬리 NPNCB(Link tail NPNCB)라고 하였다. 이 과정을 거치면 세그먼트 별로 압축한 결과는 반드시 바이트 경계에서 끝나게 된다. 바이트 단위로 처리가 되면 데이터 처리가 매우 용이해지고, 압축 데이터 크기를 바이트 단위로 쉽게 표현할 수 있게된다. 2.2절에서 상술한대로 기존 디코더들은 제안된 방법과 같이 제어 헤더 및 연결 꼬리로서 추가된 NPNCB를 무시하므로 압축 해제 과정에 영향을 받지 않기 때문에 호환성을 유지할 수 있다.

3.2 병렬 처리를 위한 제어 헤더(Control Header)

제안하는 병렬처리 압축 및 압축 해제 기법의 가장 큰 특징은 데이터를 분할하여 병렬 처리를 하면서 기존의 디코더에서 압축 해제가 가능하도록 호환성을 유지할 수 있다는 점이다. 일반적으로 원본 데이터를 분할하는 형태의 병렬 처리는 쉽게 고려가 가능하나, 이 경우 압축 해제를 위해서 부가정보를 반드시 별도로 구성하여 압축 결과에 담아야 한다. 이를 처리하기 위해서 기존 디코더는 프로그램 변경이 필요하다. 대부분의 디코더로서는 표준을 준수하면 되기 때문에 호환성을 해치면서까지 새로운 처리 방식을 고려하기는 쉽지 않을 것이다. 따라서 기존 디코더에 별도의 추가 작업 없이도 호환성을 유지하는 것은 매우 중요하다. 이를 구현하기 위한 가장 중요한 부분은 Fig. 7과 같이 연속된 NPNCB 블록의 형태로 구성된 제어 헤더의 삽입 과정이다. 제어 헤더는 압축 결과의 가장 앞부분에 위치하며, 원본 데이터를 세그먼트로 분할한 개수와 각 세그먼트별 압축데이터의 크기에 관한 정보를 5비트 단위로 분리하여 여러 개의 NPNCB 내의 DBA 영역에 나누어 저장한다. 기존 디코더들은 NPNCB를 Fig. 8과 같이 DBA 영역을 무시(ignored)하기 때문에 디코딩에 영향을 주지 않는다.

한편 제안하는 알고리즘을 적용한 병렬 처리 전용 디코더로 처리하면 Fig. 9와 같이 제어 헤더 부를 특별히 인식하여

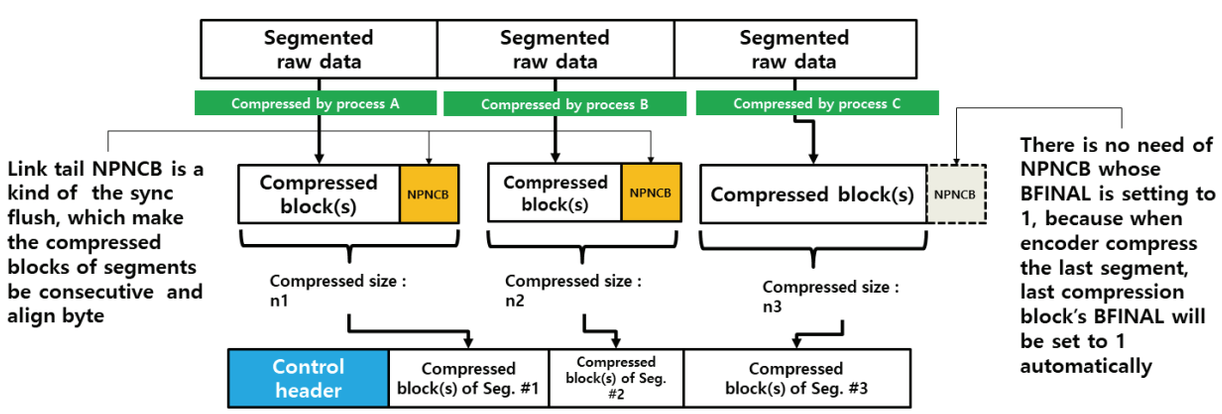

Fig. 7. The Summary of Main Step of Segmented Data Compressed Individually and Concatenated by Parallel Processes with Link Tail NPNCB and Control Header

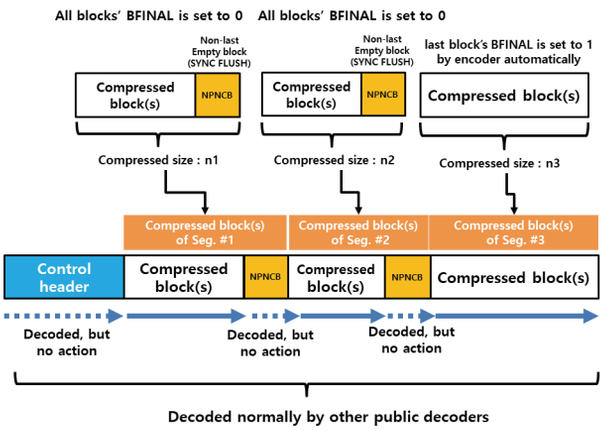

Fig. 8. Decoding Compressed Data Encoded by Proposal Method with other Public Decoders

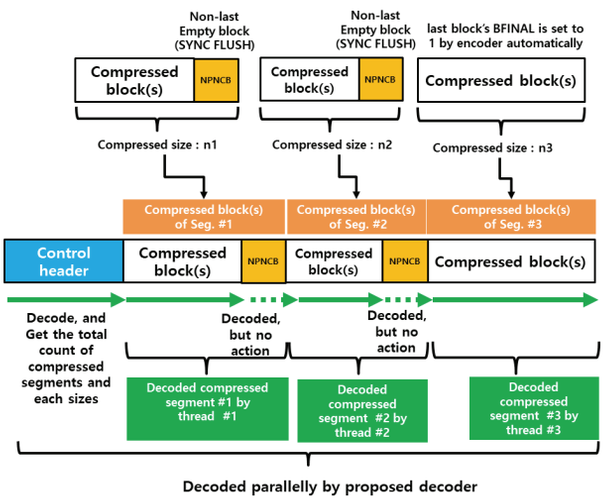

Fig. 9. Decoding Compressed Data Encoded by Proposal Method with Proposed Decoder

압축한 전체 세그먼트의 개수 및 각 세그먼트의 압축 크기를 알게 되어, 병렬 압축 해제 처리 가능하다.

3.3 제어 헤더의 상세 구조

제어 헤더는 3.2절에서 언급한 바와 같이 전체 세그먼트 개수 및 각 세그먼트를 압축한 데이터 크기에 관한 정보를 여러 개의 NPNCB 블록의 DBA 영역에 나누어 담은 뒤 이 NPNCB 블록들을 연속적으로 연결하여 구성하였다. Fig. 10에서와 같이, 첫 번째 NPNCB 의 DBA영역에는 원본데이터를 몇 개의 세그먼트로 분할했는지를 저장한다. 세그먼트의 개수가 많을 경우 이를 표현하기 위해서는 더 많은 비트가 필요하다. 예를 들어, 20비트로 세그먼트의 전체 개수를 표현해야 한다면 DBA는 5비트이므로 NPNCB는 4개 필요하다. 이와 같은 프로토콜의 변경은 엔코더 및 디코더가 미리 알고 있어야 한다. 한편, Fig. 10은 DBA를 1개 활용하여 최대 32개까지의 세그먼트 개수 정보를 저장한 사례이다. 제안하는 디코더는 첫 번째 NPNCB의 DBA 5비트는 전체 세그먼트의 개수로 인식한다. 다음으로 각 세그먼트를 압축한 데이터 크기를 각각 20비트로 표현할 경우 최대 2^{20} 바이트 크기까지 표현 가능하며, 이 정보는 5비트씩 나뉘어 4개의 NPNCB내의 DBA에 저장된다. Fig. 10의 예시는 전체 3개의 세그먼트에 대해서 처리하므로 세그먼트 각각의 압축 데이터 크기 정보를 저장하기 위해서 NPNCB가 12개가 필요하다. 요약하면, 제어 헤더에는 전체 세그먼트 개수를 표현하기 위한 1개의 NPNCB 블록과 세그먼트를 압축한 데이터 크기를 저장하기 위해 12개의 NPNCB 블록이 필요하다. 1개의 NPNCB 블록은 바이트 경계에서 시작할 경우 Fig. 2와 같이 항상 5바이트를 차지하므로, 제어 헤더는 65바이트를 차지하게 된다. 각 세그먼트에 대한 압축 데이터 크기 정보는 세그먼트 개수만큼 가변적으로 존재하므로, 제어 헤더 영역의 크기는 가변적이다.

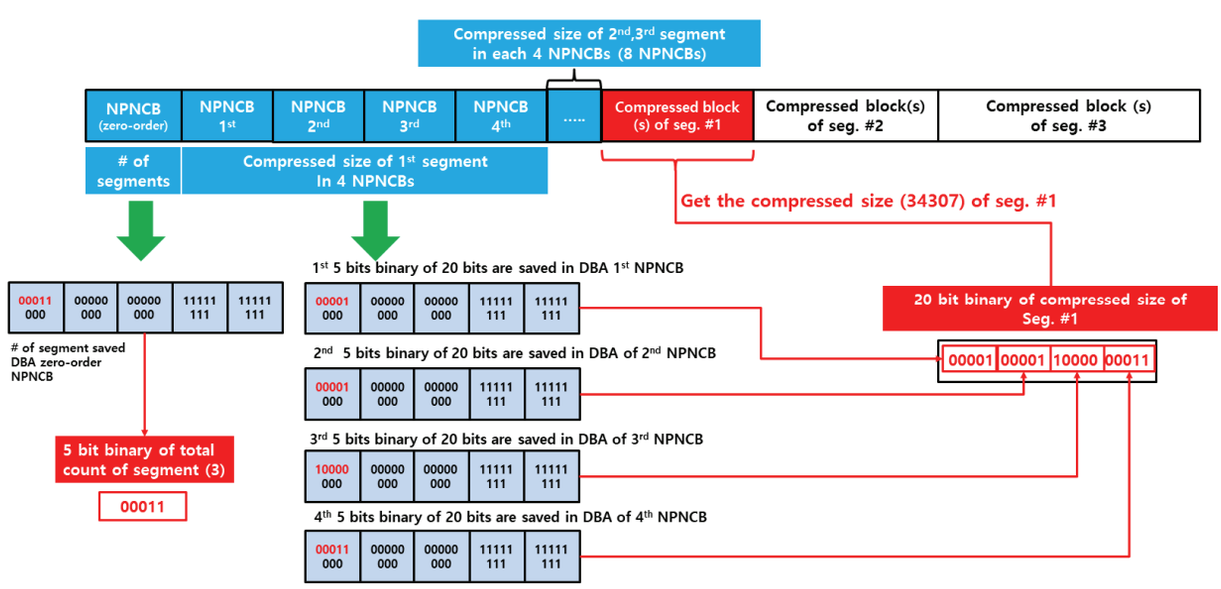

Fig. 10. Getting the Compressed Size of Each Segment by Reading Control Header

3.4 세그먼트별 압축 데이터 생성

3.3절에서 언급하였듯이 제어 헤더에는 각 세그먼트에 대한 압축 데이터 크기가 저장되어야 하는데 압축 데이터 크기는 바이트 단위이다. 그런데 각 세그먼트의 압축 결과는 RFC 1951 표준에 따르면 바이트내의 어떤 위치에서도 끝날 수도 있다[8].

따라서 압축 결과를 항상 바이트의 경계에서 끝나게 하면서 기존 디코더로 압축 해제 시에 각 세그먼트의 압축 데이터의 경계 지점에서 압축 해제가 멈추지 않고 계속하여 압축 해제 과정을 이어갈 수 있도록 하여야 한다.

이를 위해 하나의 세그먼트를 압축하여 생성된 압축 블록들은 모두 계속되는 압축 블록임을 나타내도록 압축 블록 헤더내의 BFINAL정보를 '0'으로 세팅한 다음, 추가로 계속 되는 압축 블록(NPNCB의 BFINAL 정보를 '0'으로 세팅)으로서 NPNCB 블록을 세그먼트 압축 데이터의 마지막에 추가한다. 이것이 3.1절에서 상술한 연결 꼬리 NPNCB(Link tail NPNCB)이다. 이 작업은 Zlib 압축에서 SYNC_FLUSH 모드를 사용한 압축과 동등한 작업이다.

후술할 실험 과정에서도 위의 과정을 간편하게 구현하기 위해 SYNC_FLUSH 모드를 활용하였다. 한편 마지막 세그먼트의 압축 결과로 생성된 압축 블록들에 이어 마지막 블록인(즉 BFINAL이 1로 세팅된 압축 블록)인 NPNCB를 추가한다. 기존 디코더들은 마지막 압축 블록으로 처리된 NPNCB를 읽고 압축 해제 과정을 정상적으로 종료할 수 있다.

다만 Fig. 8과 Fig. 9에서와 같이 마지막 세그먼트를 압축할 경우 Deflate 알고리즘은 자동적으로 마지막 압축 블록의 헤더내의 BFINAL 정보를 '1'로 세팅하여 바이트 단위로 결과를 반환 하여 준다. 따라서 본 연구에서는 별도로 마지막 세그먼트의 압축 결과에 마지막 압축 블록(BFINAL 정보를 1로 세팅)인 NPNCB를 추가하지 않았다.

3.5 제어 헤더 영역 크기 예측 계산

원칙적으로 병렬 압축 데이터에 포함되는 제어 헤더 영역내의 세그먼트별 압축 데이터 크기는 모든 세그먼트에 대한 병렬 압축 과정이 끝나야 알 수 있기 때문에 시간적으로 가장 마지막에 생성한다. 그러나 제어 헤더를 마지막에 생성하여 출력 스트림의 맨 앞에 첨부하기 위해서는 이미 생성된 압축 데이터에 대한 스트림 처리가 필요하다. 이 과정으로 인해 병렬 압축을 통해 얻은 압축 시간 절감의 효과는 반감될 수 있다. 이를 해결하기 위해 미리 계산된 크기의 빈 제어 헤더(empty control header)를 먼저 생성한 다음, 각 세그먼트별로 압축 데이터를 병렬 처리하여 모두 생성한 뒤 빈 제어 헤더에 각 세그먼트의 압축 크기를 제어 헤더에 업데이트 하였다.

예를 들어, 원본 데이터가 10 메가바이트(이하 MB)라 하고, 병렬 처리를 담당하는 스레드의 개수를 10개로 설정하면, 1개의 스레드가 처리해야 하는 세그먼트 데이터 크기는 1 MB이다. 전체 세그먼트 개수는 10개가 되므로, 제어 헤더의 첫 번째 NPNCB의 DBA에 십진수 10을 나타내는 이진수 "01001"를 저장하고, 각 세그먼트에 대한 압축 데이터 크기를 20비트로 표현할 경우, 4개의 NPNCB의 DBA에 나누어서 데이터 크기를 저장해야 하므로, $4 \times 10 = 40$개의 NPNCB가 필요하다. 따라서 제어 헤더는 전체 41개의 NPNCB로 구성된다. 1개의 NPNCB 당 바이트 경계에서 시작할 경우 항상 5바이트가 필요하므로 그 크기는 $41 \times 5 = 245$바이트임을 알 수 있다. 따라서 245 바이트의 빈 제어 헤더를 먼저 출력하고, 각 세그먼트 별로 압축 데이터를 병렬 처리하여 생성한다. 병렬 압축이 끝나면 다시 빈 헤더 제어부로 돌아와서 각 세그먼트의 실제 압축 데이터의 크기를 빈 제어 헤더부의 각 NPNCB의 DBA에 업데이트 하여 압축 데이터 생성을 보다 효과적으로 할 수 있다.

3.6 병렬처리 디코더 개요 및 병렬 압축 해제 과정

제안하는 Deflate 병렬 처리 알고리즘의 가장 큰 장점은, 병렬 처리된 압축 데이터를 기존 디코더로 압축 해제 할 수 있다는 점이다. 물론, 이 경우에 병렬처리 방식으로는 압축해제 되지는 않는다. 그러나 본 연구에서 제안하는 디코더를 사용할 경우 동일 압축 데이터로 병렬 압축 해제 가능하다. 이런 방식이 가능한 이유는 기존 디코더들이 NPNCB를 처리하는 방식에 있다. Fig. 8에서와 같이 NPNCB의 특성상 압축 해제 시 생성되는 원본 데이터가 없기 때문에 압축 해제 데이터의 변조 여부를 검증하는 CRC체크 테스트도 정상적으로 통과한다[5].

병렬 처리용 디코더는 Fig. 10과 같이 제어 헤더 영역의 첫 번째 NPNCB의 DBA로부터 전체 세그먼트 개수를 얻는다. 다음으로 첫 번째 세그먼트의 압축 데이터의 크기를 얻기 위해, 4개의 NPNCB를 읽어서 각 DBA 영역을 순차적으로 조합하여 20비트로 구성된 첫 번째 세그먼트의 압축 데이터 크기 정보를 얻는다. 두 번째 이후 세그먼트의 압축 데이터 크기도 같은 방식으로 구한다. 모든 세그먼트의 압축 데이터 크기를 구하면, 각 세그먼트의 압축데이터가 시작하는 위치와 끝나는 위치를 알 수 있다. 다음 각 스레드를 이용하여 병렬적으로 각 세그먼트 압축 데이터의 시작 위치에서 압축 크기만큼 읽어온 뒤 압축 해제한다. Fig. 10을 다시 보면, 첫 번째 NPNCB로부터 DBA영역에서 "00011"을 확인한다. "00011"은 십진수로 3이므로, 3개의 세그먼트가 압축되어 있음을 의미한다. 각 세그먼트의 압축 데이터 크기는 20비트로 표현하므로 5비트의 DBA가 4개가 필요하다. 따라서 1개의 세그먼트의 압축 데이터 크기를 표현하기 위해서 4개의 NPNCB가 필요하고 전체 세그먼트가 3개이므로 12개의 NPNCB가 필요하다. 결과적으로 제어 헤더는 전체 13개의 NPNCB로 구성되어 있음을 알 수 있다. 제어 헤더의 두 번째 NPNCB에서부터 얻은 4개의 NPNCB로부터 확인한 DBA는 다음과 같이 "00001", "00001", "10000", "00011"이며 이를 연결하여 20비트로 만들면, "00001000011000000011"이다. 이 값은 십진수로 34307 이며, 따라서 첫 번째 세그먼

트의 압축 데이터의 크기는 34307 바이트임을 의미한다. 이 크기는 Fig. 7에서 상술한 바와 같이 연결 꼬리 NPNCB를 포함한 크기이다. 두 번째 세그먼트의 압축 데이터 크기 또한 4개의 NPNCB를 제어 헤더로부터 계속 읽어와 동일한 방식으로 구한다. 모든 세그먼트의 압축 데이터 크기를 파악하면, 제어 헤더에 이어진 압축 데이터에서 어떤 영역이 특정 세그먼트의 압축 결과인지를 모두 식별할 수 있고, 식별한 압축데이터는 병렬 처리하여 독립적으로 압축 해제한 다음 압축 해제 데이터를 순서대로 결합하여 최종 압축 해제 원본 데이터를 얻게 된다.

4. 구현 및 실험결과

4.1 실험 개요

제안한 알고리즘을 탑재한 엔코더, 디코더는 실제의 대규모 압축 데이터를 병렬 처리하기 적합하도록 Fig. 10에서 언급한 제어 헤더의 구조를 확장하여 Fig. 11과 같이 전체 세그먼트의 개수를 3개의 NPNCB의 DBA에 나누어 저장하여 전체 15비트로 표현하도록 함으로써 32,768개의 세그먼트를 처리할 수 있도록 하였다. 또한 각 세그먼트의 압축 데이터 크기는 5개의 NPNCB의 DBA에 나누어 저장하여 전체 25비트로 표현함으로써 33,554,432 바이트까지의 크기를 표현할 수 있도록 하였다. 따라서 최대 압축 데이터는 32,768 × 33,554,432 바이트인 1,099,511,627,776 바이트 크기의 압축 데이터를 병렬 처리할 수 있도록 하였다. 한편, 실무상 매우 큰 데이터를 병렬 압축 처리하기 위해서는 메모리 제약 등으로 인해 한 번에 매우 큰 데이터를 몇 개의 세그먼트로 직접 나누어서 처리하기에는 어렵다. 따라서 Fig. 12와 같이 엔코더의 메모리 버퍼 크기를 기준으로 원본 데이터를 몇 번에 나누어 읽어 와서 처리하였다. 엔코더 메모리 버퍼로 읽어오는 과정은 순차적으로 처리하되, 일단 엔코더의 메모리 버퍼로 읽어온 데이터는 미리 정해진 개수의 세그먼트로 분할하여 병렬 압축하였다. Fig. 12에서는 3번에 걸쳐서 엔코더 메모리 버퍼에 원본 데이터를 나누어 담아서 병렬 처리하는 모습을 도식화 하였다. 3회로 나누어 읽어온 원본 데이터로부터 생성된 각각의 세그먼트들은 모두 고유하게 구분되어 압축 데이터 크기를 제어 헤더에 저장된다. 실험의 간편함을 위해 압축을 하기 위한 원본 데이터를 10MB 크기로 설정된 엔코더 메모리 버퍼로 10MB 씩 읽어 와서 1 MB 단위로 데이터를 분할하여 세그먼트로 만든 다음, 각 스레드가 1 MB의 원본 데이터를 압축 처리하였다.

또한 Fig. 7과 같이 각 스레드가 처리한 압축 데이터의 마지막에 연결 꼬리 NPNCB를 추가한 다음 결합하여 최종 압축 데이터를 생성하였다. 3.4절에서 언급한 바와 같이 마지막 세그먼트의 압축데이터에는 별도의 연결 꼬리 NPNCB를 추가하지 않았다. 이어서 다음 원본 데이터를 엔코더 메모리 버퍼로 읽어 와서 동일한 작업을 반복한다. 모든 원본 데이터에 대한 처리 작업이 끝나면, 세그먼트 각각의 압축 데이터 크기 정보를 확인하여 제어 헤더 영역에 업데이트 하였다. 제어 헤더는 3.5절에서 상술한 바와 같이 미리 크기를 계산할 수 있는데,엔코더 메모리 버퍼의 크기는 READ_BUFFER_ SIZE라고 하였으며 그 크기는 1024(byte) × 1024 × THREAD_COPY_ NUMBER 로 설정하였다. 즉 10 MB라고 설정하였다. 가용한 병렬 처리 스레드의 개수를 THREAD_ COPY_NUMBER라고 하였고 본 연구에서는 이를 10이라고 설정하였다. 하나의 스레드가 처리할 세그먼트 크기를 THREAD_ HANDLE_SIZE라면 READ_BUFFER_SIZE / THREAD_ COPY_ NUMBER로 계산되므로 1MB로 계산하였다. 제어 헤더를 구성하기 위해 필요한 전체 NPNCB의 개수는 TOTAL_ NPNCB_NEEDS라고 하였으며, 3 + (원본데이터 전체 크기 / THREAD_HANDLE_ SIZE + 1) × 5로 계산이 되는데, 3 은 3개의 NPNCB를 나타내며 세그먼트 전체 개수를 15비트로 표현하기 위하여 필요한

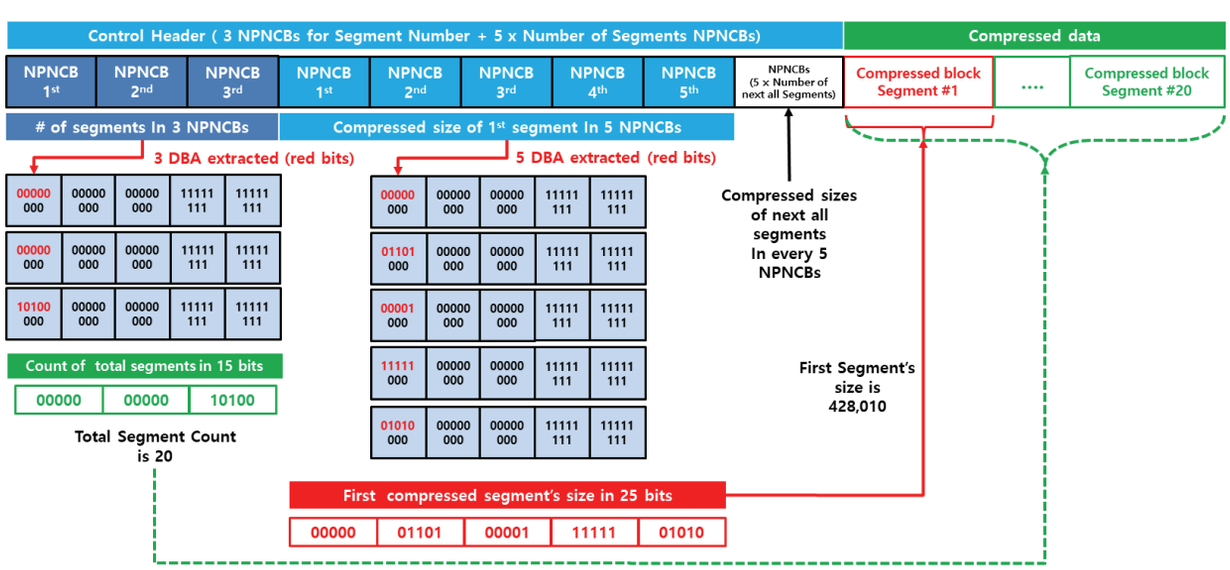

Fig. 11. Getting the Compressed Size of Each Segment using Extended Control Header in the Experiment Program

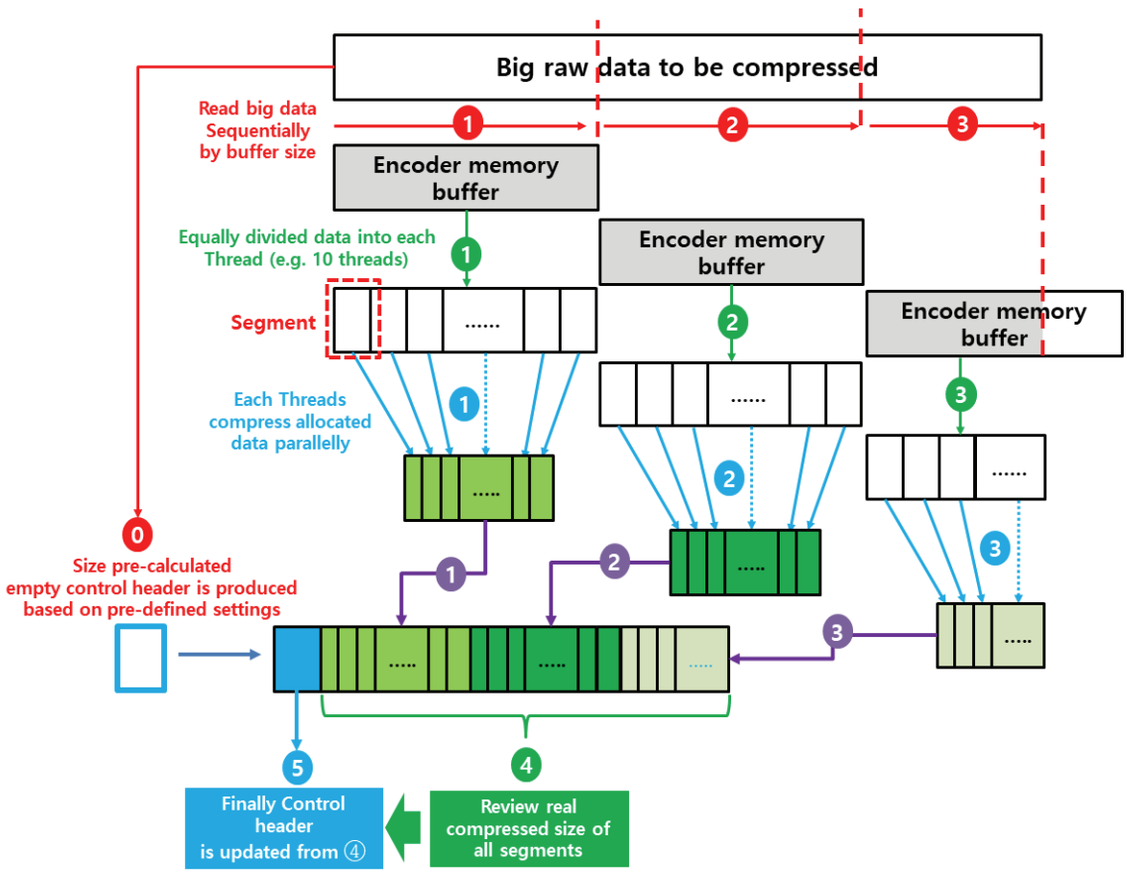

Fig. 12. Parallel Encoding the Big Raw Data using Encoder Memory Buffer

개수이며, 세그먼트 각각의 압축 데이터 크기를 25비트로 표현한다면 5개의 NPNCB 가 필요하므로 5를 곱하였다. THREAD HANDLE SIZE 보다 원본 데이터가 작을 경우에도 하나의 세그먼트는 반드시 할당되어야 하므로 수식에 1을 더하였다. 따라서 최종 제어 헤더 크기는 TOTAL_NPNCB_NEEDS × 5 바이트가 된다.

4.2 실험 환경

본 연구에서는 Microsoft Visual Studio Community 2022 (64-bit) 버전 17.3.2 에서 C#을 이용하여, 릴리즈 모드(Release mode)로 테스트하였다. 개발 하드웨어 장비는 Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz 2.81 GHz, RAM 16GB 이며, OS는 Windows 10(64비트, x64기반 프로세서)을 사용하였다.

4.3 구현 라이브러리 및 개발 방법

본 연구에서는 Deflate 압축 및 압축 해제 라이브러리로서, C#으로 구현한 Ionic.Zlib.DeflateStream 함수를 사용하였다. 원본 데이터를 각 세그먼트로 분할하여 병렬 압축한 다음 연결 꼬리 NPNCB를 자동적으로 부가하기 위해 Ionic.lib.DeflateStream에서 지원하는 FlushMode 속성 값을 Ionic.Zlib.FlushType.Sync 로 세팅하여 Sync Flush 기능을 활용하여 구현하였다. 한편 제어 헤더는 직접 그 크기를 미리 계산하여 빈 제어 헤더를 먼저 생성한 다음, 각 세그먼트에 대한 압축이 끝나고 다시 빈 제어 헤더의 각 NPNCB내의 DBA영역에 압축 데이터 크기 정보를 업데이트 하였다.

4.4 병렬 처리 엔코더 성능 실험 결과

Table 1은 다양한 원본 대상 데이터에 대하여 본 연구에서 제안한 병렬처리 방법을 이용하여 압축한 결과와 압축에 소요된 시간을 나타내었고, 비교를 위해 병렬처리 과정을 제외한 동일한 압축 알고리즘으로 같은 데이터를 압축 하였을 때의 크기와 소요된 시간을 나타내었다.

세그먼트 단위로 원본 데이터를 나누고, 각 스레드가 담당한 원본 데이터의 세그먼트를 표준 Deflate 압축 알고리즘으로 압축한 다음, 제어 헤더와 연결 꼬리 NPNCB를 부가하는 과정까지의 전체 시간 및 전체 압축 사이즈를 비교하였다. 정확한 성능 비교를 위해 각각의 세그먼트를 병렬 압축할 때와 병렬 압축 방법을 사용하지 않고 압축하는데 사용된 압축 알고리즘은 Ionic.Zlib패키지의 DeflateStream 라이브러리를 동일하게 사용하였다. 3.4절에서 설명한 바와 같이 Ionic.Zlib 패키지를 사용하면 Sync flush 기능을 간단하게 구현할 수 있는 장점이 있다.

Table 1을 살펴보면 엑셀 저장 파일, MS-SQL db backup

Table 1. Parallel Compression Performance Test Results

Tested File Type	Raw data size (bytes)	Compressed Size by Parallel encoder (bytes)	Compression Time by Parallel encoder (milliseconds)	Compressed Size by Single thread encoder (bytes)	Compression Time by single thread encoder (milliseconds)	Compression time saving (%)
xlsb	20,937,767	5,121,593	322.2752	5,050,039	690.499	53.3
bak	122,290,176	14,903,438	1109.272	14,857,160	3845.6244	71.2
dll	14,455,296	1,092,705	298.7284	1,090,814	674.2262	55.7
docx	23,882,832	22,239,800	431.7175	22,236,771	1122.625	61.5
pdf	4,481,132	4,176,930	158.6545	4,175,360	227.0535	30.1

Table 2. Parallel Decompression Performance Test Results

Tested File Type	Decompression Time by Parallel decoder (milliseconds)[a]	Decompression Time by Single thread decoder (milliseconds)[b]	Decompression Time of Single thread compressed data by Single thread decoder (milliseconds)[c]	Decompression Time Saving (%)[a/b*100]
xlsb	60.2388	118.1633	98.2571	38.7
bak	285.703	545.6874	552.9623	48.3
dll	26.6868	40.5154	45.4582	41.3
docx	81.5013	225.4499	237.5012	65.7
pdf	20.9886	46.3583	45.0396	53.4

파일, dll 파일, 워드 문서화일, pdf 문서 파일 등을 테스트 하였으며, Compressed Size by Parallel encoder 값과 Compressed Size by Single thread encoder를 비교해보면 병렬 처리로 압축한 데이터의 크기와 싱글 처리로 압축한 전체 데이터의 크기가 거의 비슷한 수준으로 나온 것을 확인할 수 있었다. 이를 통해 병렬 처리 압축 시의 제어 헤더 및 연결 꼬리 NPNCB의 삽입으로 인한 압축률의 손해는 크지 않음을 알 수 있었다. 또한 전체적으로 병렬 처리 시 압축시간 절감 효과가 53.3%에서 71.2%까지 있음을 확인하였다.

4.5 병렬 처리 디코더 성능 및 압축 해제 호환성 증명

Table 2는 디코딩 성능 테스트 결과를 보여주고 있다.

정확한 압축 해제 성능 비교 및 제어 헤더 및 연결 꼬리 NPNCB가 삽입된 제안된 병렬 압축 데이터에 대하여도 기존 디코더들이 압축 해제를 정상적으로 진행함을 확인하기 위해, 기존 디코더중의 하나인 C# 의 기본 압축 패키지인 System.IO.Compression의 DeflateStream 라이브러리로 압축 해제하였다. 그 결과 성공적으로 압축 해제 되었다. 또한 제안하는 병렬처리 전용 디코더로 압축해제 시, 병렬압축 해제 시간이 기존 싱글스레드 압축 해제 처리보다 38.7%에서 65.7%까지 시간을 절감 시킬 수 있음을 확인하였다.

Table 2의 c 항목에 일반적 방식의 기존 엔코더로 압축한 데이터를 싱글 스레드(single thread)방식의 디코더로 압축 해제 한 결과도 표현하였다. 싱글 스레드 방식의 디코더로는 병렬 압축된 데이터 또는 기존 엔코더로 압축된 데이터를 압축 해제 하는 시간에 큰 차이가 없었다. 따라서 병렬 압축된 데이터를 기존 디코더로 압축해제 할 경우에도 압축해제 시간에 특별한 차이는 없었다.

5. 결 론

5.1 제안하는 알고리즘의 성능과 호환성

Deflate 알고리즘은 압축 효율을 높이기 위해 입력 원본 데이터의 최대 32KB 앞의 내용을 참조하여 LZ77 압축을 진행하는데 제안 방법에서는 인위적으로 원본 데이터를 세그먼트로 나누기 때문에 각 세그먼트 압축 시 해당 세그먼트를 새로운 압축 원본으로 인식하여 압축을 진행하여 압축률이 다소 떨어질 수 있다. 그러나 실험에서 확인하였듯이 1MB 정도 단위로 세그먼트를 나눌 경우 전체적인 압축률의 저하는 크지 않다는 점을 알 수 있었다. 세그먼트의 크기 또는 개수와 압축률 저하와의 관계는 추가 연구를 통해 확인할 필요가 있다. 또한 제안된 방법과 같이 NPNCB 블록을 활용할 경우 병렬 압축 및 압축 해제에 필요한 정보를 NPNCB 의 DBA 영역에 담기 때문에 기존 디코더들과의 호환성을 유지 할 수 있었다. 한편 제안하는 병렬처리 전용 엔코더 및 디코더로 알고리즘 수준에서 싱글 스레드 처리와 비교하여 향상된 속도로 압축 및 압축해제를 수행할 수 있음을 확인하였다.

5.2 기존 연구들과의 비교와 한계점

본 연구에서 제안하는 방법과 다른 연구에서 제안하는 방

법을 직접 비교하기에는 테스트 프로그램의 최적화와 성능 테스트에 사용되는 원본 데이터 및 개발 언어 선택과 구체적인 프로그래밍 방법에 따른 차이가 매우 크기 때문에 후속 연구를 통한 엄밀한 비교가 필요하다고 판단하였다. 다만 상술한 바와 같이 본 연구에서 제안한 방법은 NPNCB의 DBA에 병렬 압축 및 압축해제 정보를 담아서 구성한 제어 헤더를 압축 데이터의 앞쪽에 삽입하여 새로운 병렬 처리 방법을 제안하였고 기존 디코더와 호환성을 갖추도록 고안하였다는 점에서 매우 큰 차별점이 있다고 보여 진다. 유사한 연구로서 병렬 압축에 관련한 연구 가운데 소프트웨어 방식으로 많이 알려진 것으로서 Pigz(Parallel gzip)이 있다[3]. gzip 형식 내에서 Deflate 알고리즘을 병렬 처리한 것인데 병렬 압축은 구현되었으나 압축 해제 과정은 본 연구와 달리 싱글 스레드 방식으로 처리된다고 알려져 있다[9]. 또한 E. Sitaridi.등의 같은 연구에서는 Gompresso 라는 새로운 파일 형식을 제안하였다. 병렬 압축 과정에서 원본 데이터를 동일한 크기로 분할한 뒤 LZ77을 병렬적으로 처리하여 압축한 뒤, 허프만 압축을 병렬적으로 시행하여 여러 개의 부 블록(sub-block)으로 구성된 압축 데이터를 생성하여 파일을 구성한다는 점이 주요 특징이다[9]. 그러나 이러한 파일 형식을 해석하여 병렬 압축해제 할 수 있는 디코더를 새롭게 고안하여야 하므로 기존 Deflate 디코더로 해당 압축 데이터를 압축 해제할 수 없다. 한편 J. Gilchrist and Y. Nikolov가 개발한 병렬 처리 알고리즘인 BZIP2(Parallel BZIP2)의 경우는 본 연구와 유사하게 동일한 원본 데이터 블록에 대하여 병렬 압축 및 병렬 압축 해제가 가능하게 한 사례이다[10]. 그러나 BZIP2는 BWT(Burrows-Wheeler Transform) 알고리즘과 허프만 코딩을 사용하는 알고리즘으로서 LZ77과 허프만 코딩을 사용하는 Deflate알고리즘과 직접적인 알고리즘에 성능을 비교하기에는 다소 무리가 있었다.

5.3 보안 취약점과 관련한 한계점

NPNCB의 DBA 영역에 악의적 코드를 숨기거나 삽입 할 수 있다는 점에서 Deflate 알고리즘 자체에 취약점이 존재한다[5]. 그러나 NPNCB는 기존압축 프로그램과 호환성 유지를 하면서도 NPNCB 의 DBA 영역에 병렬 처리를 위한 부가 정보를 기록할 수 있다. 이는 표준으로 약속한 Zip 포맷의 구조나 Deflate 알고리즘을 변화시키지 않고 표준의 틀 안에서 새로운 부가 정보를 담을 수 있다는 의미이므로 매우 유용하다고 판단된다. 따라서 장점을 활용하고 보안 취약점 우려를 해소하기 위해 NPNCB내의 악성 데이터를 검사할 수 있는 보안 수단이 필요하다고 사료된다.

References

[1] D. Takafuji, K. Nakano, Y. Ito, and A. Kasagi, "Acceleration of deflate encoding and decoding with gpu implementations," *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*, pp.180-186, 2021.

[2] M. Kerbiriou and R. Chikhi, Parallel decompression of gzip-compressed files and random access to DNA sequences [Internet], https://doi.org/10.48550/arXiv.1905.07224.

[3] M. Adler, pigz: A parallel implementation of gzip for modern multi-processor multi-core machines [Internet], https://zlib.net/pigz.

[4] Bandisoft, Parallel extraction [Internet], https://kr.bandisoft.com/bandizip/help/parallel-extraction.

[5] J. H. Kim, "Malicious code injection vulnerability analysis in the deflate algorithm," *Journal of The Korea Institute of Information Security and Cryptology*, Vol.32, No.5, pp.869-879, 2022.

[6] T. Ylonen, SSH Communications Security Corp,C. Lonvick, Ed., Cisco Systems, Inc.,"*The Secure Shell (SSH) Transport Layer Protocol,*" *RFC 4253*, Jan. 2006.

[7] J. Gailly and M. Adler, Zlib 1.2.11 manual [Internet], https://www.zlib.net/manual.html.

[8] L. Peter Deutsch, "*DEFLATE compressed data format specification version 1.3*," *RFC 1951*, May 1996.

[9] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. Ross, Massively-parallel lossless data decompression [Internet], http://arxiv.org/abs/1606.00519.

[10] J. Gilchrist and Y. Nikolov, Parallel BZIP2 [Internet], http://compression.ca/pbzip2.

김 정 훈

https://orcid.org/0000-0003-3866-1165
e-mail : powerzenith@naver.com
2002년 서울대학교 약학과(학사)
2010년 서울대학교 보건대학원(석사)
2013년 서울대학교 보건대학원(박사수료)
2019년 경희대학교 SW융합학과(석사)
2016년 ~ 현 재 바이너리랩㈜ 대표
관심분야 : 의료정보, 정보이론, 정보보호