한국정보전자통신기술학회논문지(jkiiect)'22-12, Vol.15 No.6
논문 22-15-06-492 http://dx.doi.org/10.17661/jkiiect.2022.15.6.492

IoT 장비에 있어서 실시간 데이터 압축 전송을 위한 BL-beta 유니버설 코드의 경량화, 고속화 연구

김정훈*

The study on Lightness and Performance Improvement of Universal Code (BL-beta code) for Real-time Compressed Data Transferring in IoT Device

Jung-Hoon Kim*

요 약 본 연구는 IoT 센싱 데이터의 무 손실 실시간 전송에 활용 가능한 BL-beta 코드의 엔코딩 및 디코딩 성능 개선을 통해 효과적으로 압축 데이터를 실시간으로 전송하고, 해독할 수 있도록 로직을 개선한 결과에 대한 연구이다. BL-beta 코드의 엔코딩 과정에는 비교적 연산 부담이 큰 로그 함수와 지수 함수, 나눗셈 및 제곱근 연산 등이 포함되어 있는데 이를 개선하여 비트 연산과 이진수 패턴 분석 그리고 비트 패턴을 이용한 뉴턴-랩슨 방법의 초기 값 설정을 통해 빠르게 데이터를 BL-beta 코드로 엔코딩 및 디코딩 할 수 있는 새로운 규칙성을 발견하였으며 이를 적용하여 기존 연구와 비교하여 알고리즘의 엔코딩 속도를 평균 24.8%, 디코딩 속도를 평균 5.3% 개선하였다.

Abstract This study is a study on the results of improving the logic to effectively transmit and decode compressed data in real time by improving the encoding and decoding performance of BL-beta codes that can be used for lossless real-time transmission of IoT sensing data. The encoding process of BL-beta code includes log function, exponential function, division and square root operation, etc., which have relatively high computational burden. To improve them, using bit operation, binary number pattern analysis, and initial value setting of Newton-Raphson method using bit pattern, a new regularity that can quickly encode and decode data into BL-beta code was discovered, and by applying this, the encoding speed of the algorithm was improved by an average of 24.8% and the decoding speed by an average of 5.3% compared to previous study.

Key Words : BL-beta code, BL-beta Performance Improvement, IoT data compression, Real-time compression, Universal code

1. 서론

1.1 BL-beta 코드의 개요와 특징

BL-beta 코드는 유니버설 코드(Universal Code)의 일종으로서, 임의의 정수 또는 자연수를 유일 복호성(unique decodability)을 가진 이진수 코드로 압축(encoding)하거나 또는 압축 해제(decoding)하는 코드이며 간단하면서도 비교적 높은 압축률을 가진 것이 특징이다[1]. 유니버설 코드는 심볼(symbol)의 출현 빈도(frequency)를 정확히 알기가 어렵지만 출현 빈도의 순서를 대략적으로 알고 있을 때 많이 활용된다. 특히 전송하고자 하는 데이터 특성상 상한(upper limit)이나 하한(lower limit)을 예측하기 어려운 경우이거나 실시간 수치 데이터 생성 과정에서 다양한 상황(abnormal events)으로 인해 예상 치를 벗어난 비정상 데이터(abnormal value)도 발생 할 수 있다[2].

*BinaryLab Co. Ltd.(powerzenith@naver.com)
Received December 12, 2022 Revised December 21, 2022 Accepted December 25, 2022

이러한 경우 고정 길이 비트열(fixed length bit)로 전송하면 수치 overflow 오류가 발생할 수 있으므로, 가변적 길이의 유니버설 코드를 사용하면 안정적으로 압축하여 전송할 수 있다. 또한 실시간 무 손실 전송 필요성이 급격히 늘고 있는데, IoT 장비의 하드웨어 성능은 높지 않으므로 간단한 연산으로 빠르게 압축하여 전송하는 용도로서도 활용되고 있다[3]. 최근 실제 다양한 IoT 장비에서 생산하는 데이터의 실시간 무 손실 압축을 BL-beta 코드를 이용한 결과 높은 압축률을 보여주고 있다[4].

1.2 연구의 목적

본 연구는 1.1절에서 언급한 BL-beta 코드에 대해서 실시간 압축 및 압축해제 속도를 높이기 위해 실수 및 연산 부담이 큰 기존 알고리즘을 개선하여, 속도가 빠른 비트 연산 및 비트 패턴 확인만으로 엔코딩 및 디코딩이 가능하여 처리 속도를 높일 수 있도록 알고리즘을 개선하였다. 이를 통해 IoT 장비에서의 실시간 전송 효율을 높이고 경량화를 달성하고자 하였다. 또한 엔코딩 대상 수치 값의 비트 패턴과 그 길이 정보만을 통해 주요 변수 값을 구할 수 있으므로, 매우 큰 수치 값에 대해서도 연산 오버 플로우(overflow) 없이 BL-beta 코드로 엔코딩 및 원래 값으로 디코딩을 구현할 수 있게 되었다.

2. 관련연구

2.1 BL-beta 코드 엔코딩 방법

2.1.1 BL-beta 코드의 접두코드 생성

BL-beta 코드에 대한 선행 연구에 따르면[1,3], BL-beta 코드는 두 부분으로 구분되는 데, 접두 코드(prefix code)와 접미 코드(suffix code)이다. 먼저 데이터의 압축을 위해서는 접두 코드를 다음과 같이 구한다. 먼저 압축 대상인 자연수인 Z(code-num)에 대하여, 수식 (1)에 따라 Z의 group index M 을 구한다. 참고로 Z=0인 경우를 포함하기 위해서는 Z=1로 생성한 코드를 Z=0에 대한 코드로 재 매핑(re-mapping) 하여 원하는 범위의 code-num Z를 BL-beta 코드로 엔코딩 할 수 있다.

$$M = ceiling(\log_2(\frac{Z+2^s}{2^s})) \quad (1)$$

(단, $ceiling(p)$는 p 이상인 정수 값 중 가장 작은 정수 값을 반환하는 함수이다.)

수식 (1)에서 S는 BL-beta코드의 접두 코드 다음에 나오는 접미 코드의 최초 시작 비트 크기이다. Z=1 에 대하여 S=1 인 경우, 접미 코드는 크기가 1비트에서부터 시작하여 그룹 인덱스 M이 1씩 커질 때마다 1비트씩 증가한다. S=n 이라면 n비트에서 시작하여 1비트씩 증가하는 경우를 뜻한다[1,3].

예를 들어, Z=1, S=1 일 때 BL-beta 코드는 이진 데이터 형태로 '010' 이다. Z=2 이면, '011' 이다. 최초 접두코드 '01' 에 이어서, '0' 또는 '1' 이 접미 코드가 되며 S=1이므로 1비트를 차지한다는 의미이다.

Z=1, S=2일 때 BL-beta 코드는 '0100', Z=2 일 때, '0101' 과 같이 된다. 즉 최초 접두코드 '01' 에 이어서 '00', '01', '10', '11' 과 같이 2비트의 크기로 최초의 접미 코드가 시작한다는 의미이다[3].

한편, S 및 M을 알게 되면, BL-beta 코드의 접미 코드는 M+(S-1) 비트 크기로 결정된다[1,3]. 이와 같이 M을 구하게 되면 실제 접두 코드를 생성하기 위해서는 수식 (2)에 따라 K 및 X 값을 구한다.

$$K = \left\lfloor \frac{1+\sqrt{1+8M}}{2} \right\rfloor \quad (2)$$

(단, $[p]$ 는 p보다 작은 정수 중에 최대의 정수 값을 반환하는 함수)

수식 (1), (2) 를 통해 M 및 K를 알고 있으므로, X는 수식 (3)으로 계산할 수 있다.

$$X = M - \frac{K(K-1)}{2} \quad (3)$$

수식 (1)~(3)을 통해 K 및 X를 알게 되면, (X-1)개의 '1' 이 연속되고, 이어서 K-(X-1)개의 '0' 이 이어진 다음, 최하위 비트(Least Significant Bit, 이하 LSB)가 '1'로 이루어진 이진수 코드를 생성할 수 있으

며, 이 코드가 Z 에 해당하는 접두 코드이며, 그 의미는 Z의 group index M을 나타낸다. 그림 1에서의 예시는 group index M 이 62인 경우에 사례로서, 수식 (2)에 따라 K 값을 구하면 K=11이며, X=7 이다. 따라서 X-1개(=6개)의 연속된 '1'에 이어 K-(X-1)개(=5개)의 연속된 '0' 그리고 LSB가 '1'로 세팅된 BL-beta 코드의 접두 코드가 생성되었다.

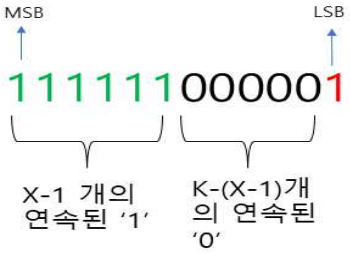

그림 1. BL-beta코드의 접두코드 생성방법 (M=62일 경우 접두코드 예시)
Fig. 1. How to generate prefix of BL-beta code(example of prefix when M=62)

물론 경우에 따라, X=1일 경우 MSB에서 LSB 방향으로 연속된 '1'이 존재하지 않을 수도 있다.

2.1.2 접미 코드 엔코딩

BL-beta 코드의 선행 연구[1,3]에서는 S+(M-1) 비트 길이를 할당 받은 접미 코드(suffix)를 아래 수식 (4)를 통해 구한 뒤,

$$suffix = Z - 2^s \times (2^{(M-1)} - 1) - 1 \qquad (4)$$

$bin(suffix)$에 '0'을 left padding하여 S+(M-1) 비트로 하였다. S=1일 경우, 접미 코드가 차지하는 비트 수는 $\leq n(bin(suffix)) = 1 + (M-1) = M$ bits 이다. 또한 접미 코드 값은 수식 (4)로부터 아래 수식 (5)와 같이 전개 된다.

$$\begin{aligned} suffix &= Z - 2 \times (2^{(M-1)} - 1) - 1 \\ &= Z - 2^M + 2 - 1 \\ &= Z - 2^M + 1 \end{aligned} \qquad (5)$$

2.2 BL-beta 코드 디코딩 방법

BL-beta의 디코딩은 먼저 접두 코드를 확인하여 그 패턴을 통해 M을 구하여야 한다[1,3]. 아래 그림과 같이 BL-beta코드의 MSB에서 LSB방향으로 이동하면서 '01'을 처음 만날 때 MSB에서부터 '01'의 '1'까지를 포함하여 분리한다. 이 값이 BL-beta코드의 접두 코드이다. 예를 들어 BL-beta 코드 '1110001111010000010010000001' 에 있어서 위와 같은 규칙으로 분리된 접두 코드가'1110001' 이므로, MSB에서 LSB방향으로 이동하면서 '0'을 처음 만날 때, 그때까지의 '1'의 개수를 T라고 하면, T=3이며, K는 분리된 비트('1110001')의 길이 - 1 이므로 6으로 계산된다[1,3]. K, T로부터 M을 구하는 아래 수식 (6)에 따라서, 접두 코드의 패턴으로부터 M=19 임을 구할 수 있다.

$$M = \frac{K(K-1)}{2} + T + 1 \qquad (6)$$

또한 위에서 예시로 든 BL-beta 코드의 MSB방향에서 LSB방향으로 이동 중 최초로 만나는 '01' 이후로부터('1' 다음 비트로부터) S+(M-1)비트의 이진수가 BL-beta 코드의 접미 코드이다. 즉 '1110001111010000010010000001' 에 있어서 '01'에 이어서 S+(M-1)=19 비트의 '1110100001001000001' 가 접미코드이다. S+(M-1) 비트가 접미 코드의 길이란 사실은 연속된 BL-beta 코드들 간의 경계를 구분할 수 있으므로 특히 중요하다[1,3]. 다시 상술한 수식 (4)로 부터, Z를 수식 (7)과 같이 계산할 수 있다.

$$Z = suffix + 2^s \times (2^{(M-1)} - 1) + 1 \qquad (7)$$

S=1,M=19 이므로 수식 (7)에 대입하면 $Z = suffix + 2^1 \times (2^{19-1} - 1) + 1$ 이므로, $Z = suffix + 2^{19} - 1$ 이다. 위 연산 과정을 그림 2에서 상세히 보였다.

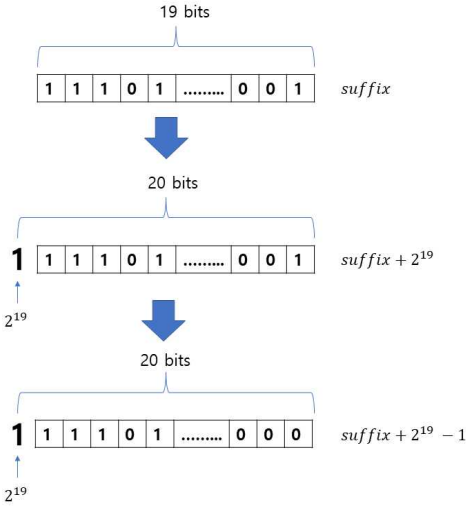

그림 2. 접미 코드로부터 Z를 구하는 계산과정
Fig. 2. Calculation process to find Z from suffix code

3. 제안하는 경량화 알고리즘

3.1 BL-beta 코드 엔코딩 효율화

3.1.1 접두 코드 엔코딩 효율화(M 값 구하기)

2.1.1절에서 살펴보았듯이, 수식 (1)~(3) 과정은 log 함수 연산과 제곱근 연산, 지수의 나눗셈 연산 등 저사양의 IoT 장비가 처리하기에 연산 부담이 큰 과정을 요구한다. 또한 높은 연산 부담은 IoT 의 전력 소모량 등을 증대시켜 활용성을 저하시킬 수도 있다[5]. 본 연구는 이와 같은 실수 함수 연산을 연산 속도가 매우 빠른 비트 이동 연산과 비트 패턴의 특성을 이용하여 간단히 함으로써 BL-beta 코드의 생성 속도와 효율을 높였다. 본 연구에서는 BL-beta 코드 중 가장 간단하여 활용성이 높은 S=1인 경우에 대해서만 분석하였다. 구체적으로는, 수식 (1) 로부터 M을 구할 때, S=1 일 경우 $Z+2^s$는 $Z+2$가 되며, $\frac{Z+2}{2}$ 값을 밑(base)가 2인 로그를 취한 값의 ceiling 함수를 구하는 것과 같다. 이를 간단한 비트 연산으로 구하기 위해, 먼저 자연수 Z(Z>=1)를 이진수 형태로 표현한다. 이를 본 연구에서는 $bin(Z)$ 와 같이 표현하기로 하였다. 즉 $bin(x)$는 x의 이진수 비트 표현으로 정의하였다. 한편 매우 큰 십진수에 대해서 이진수로 빠르게 변환하는 알고리즘은 이미 많이 알려져 있으므로[6], 매우 큰 Z 의 $bin(Z)$ 를 구하는 것에 대해서는 공개된 알고리즘을 활용할 수 있을 것이다.

한편, 수식 (1)에서 S=1인 경우는 수식 (8)과 같다.

$$M = ceiling(\log_2(\frac{Z+2^1}{2^1})) \quad (8)$$

위 수식에서 분자의 $Z+2$을 살펴보면, 2는 이진수로 '10' 이다. 따라서 Z 에 이진수 '10'을 더하는 것이며, $bin(Z)$의 LSB로부터 MSB방향으로 1비트 이동한 위치에 '1'을 더하는 것과 같다. 분모 2로 분자를 나누는 것은, Z+2 의 연산 결과를 우측으로 1비트 shift 처리하는 과정을 통해 더욱 간략화 할 수 있다. 후술하겠으나, Z+2가 홀수인 경우가 있기 때문에 이 경우엔 1 비트 우측 shift 연산 값과 실제 나누어진 값과 차이가 있기 때문에 보정하여야 하며 이는 후술할 예정이다. 한편, 수식 (1)의 연산을 비트 연산으로 일반화한다면 S=2,3,4,…와 같이 증가할 경우 $Z+2^s$에서 Z 에 더하는 이진수는 '100', '1000', '10000', … 과 같다. 이를 다시 표현하면, Z 의 LSB 에서 MSB 방향으로 S 비트 이동한 위치에 1을 더하는 것과 같을 것이다. 또한 분모 2^s 로 나누는 과정은 우측으로 S 비트 shift하는 연산 후, $Z+2^s$ 가 홀수 일 경우 특별히 고려하여 처리할 수 있다. 상술한 대로, 본 연구에서는 먼저, S=1 인 가장 간단한 형태의 BL-beta 코드의 효율화에 대해서만 고찰하였다.

(1) bin(Z) 유형 - A type

bin(Z) 가 '1'로만 이루어진 이진수인 경우로서

Z 의 이진수 변환 형이 '1', '11', '111', '1111',… 와 같이 '1'로만 이루어진 이진수일 경우, $M = ceiling(\log_2(\frac{Z+2^1}{2^1}))$ 값을 아래 표 1과 같이 구하였다. 표 1의 칼럼 제목의 '≫' 기호는 1비트 우

측 shift 연산을 의미한다. w 및 $ceiling(w)$에 대해서는 후술할 예정이다.

표 1. $bin(Z)$ A 타입의 경우 $ceiling(w)$값
Table 1. $ceiling(w)$ of bin(Z) A type

bin(Z)	bin((Z+2))	bin(((Z+2)>>1))	$\log_2$(((z+2)>>1))	w	ceiling(w)
1	11	1	0	0.xx	1
11	101	10	1	1.xx	2
111	1001	100	2	2.xx	3
1111	10001	1000	3	3.xx	4
11111	100001	10000	4	4.xx	5
111111	1000001	100000	5	5.xx	6
1111111	10000001	1000000	6	6.xx	7
…	…	…	…	…	…

표 1에서와 같이, bin(Z)가 '1', '11', '111', '1111', '11111', '111111',… 과 같이 '1'로만 이루어진 이진수의 경우 $Z+2$ 는 홀수이므로, $\frac{Z+2}{2}$의 정확한 값을 구하기 위해선 수식 (9)와 같이 $bin(Z+2)$를 1비트 우측으로 shift한 뒤 0.5를 더해주어야 정확한 값이 나온다. 즉 다음과 같다.

$$j = \frac{Z+2}{2} = bin((Z+2) \gg 1) + 0.5 \quad (9)$$

한편 표 1로부터 아래 수식 (10)과 같은 부등식을 항상 만족하므로

$$bin((Z+2) \gg 1) \geqq 1 \quad (10)$$

다음 수식 (11)의 부등식 또한 성립한다. $bin((Z+2) \gg 1)$ 이 1 이상 일 때, $bin((Z+2) \gg 1)$ 에 0.5를 더한 값 보다는 $bin((Z+2) \gg 1)$을 2배 한 값이 더 크다는 것은 자명하기 때문이다.

$$bin((Z+2) \gg 1) < bin((Z+2) \gg 1) + 0.5 \quad (11)$$
$$bin((Z+2) \gg 1) + 0.5 < bin((Z+2) \gg 1) \times 2$$

수식 (9)을 이용해서 위 수식 (11)을 다시 표현하면 수식(12)과 같다.

$$bin((Z+2) \gg 1) < j \text{ 이고 } j < bin((Z+2) \gg 1) \times 2 \quad (12)$$

이때, $w = \log_2(j)$이라면, 수식 (12)로부터 수식 (13)의 부등식이 성립함을 알 수 있다.

$$\log_2(bin(Z+2) \gg 1) < w \text{ 이고 } w < \log_2(bin(Z+2) \gg 1) + 1 \quad (13)$$

표 1과 수식 (1) 및 (13)로부터 $bin(Z)$가 A type인 경우에는 $ceiling(w)$가 아래 수식 (14)같은 관계가 있음을 알 수 있다.

$$M = ceiling(w) = ceiling(\log_2(bin(Z+2) \gg 1)) + 1 = \leq n(bin(Z)) \quad (14)$$

이로부터 $bin(Z)$가 '1', '11', '111', '1111',… 와 같이 '1'로만 이루어진 A type 인 경우 $M = \leq n(bin(Z))$임을 알 수 있다.

(2) $bin(Z)$ 유형- B type

다음으로 $bin(Z)$가 '10', '110', '1110', '11110',… 과 같이 LSB가 '0' 이고 나머지가 '1'로만 이루어진 이진수인 경우는 A type 일 때 보다 간단하게 $M = ceiling(\log_2(\frac{Z+2^1}{2^1}))$이 구해진다. 먼저 $bin(Z+2)$는 '100', '1000', '10000',… 과 같고, 모두 짝수이므로, $\frac{Z+2}{2}$ 을 구하기 위해 우측으로 1비트 shift 한 것으로도 오차 없이 구하여 지고, 그 값은 이진수로 '10', '100', '1000',…과 같이 된다. 즉 A type과 달리, $j = \frac{Z+2}{2} = bin((Z+2) \gg 1)$ 만

으로 정확하게 구할 수 있다. 이때, $w=\log_2(j)$이라면, 아래 표 2와 같이 $bin(Z)$='10', '110', '1110', '11110',…일 때 $w(=\log_2(j))$ 값은 1,2,3,4,….와 같고, $ceiling(w)$ 값 역시 1,2,3,4,… 이다. 따라서 $bin(Z)$가 B type일 때 $M=\leq n(bin(Z))-1$이 된다.

표 2. $bin(Z)$ 가 B 타입의 경우 $ceiling(w)$값
Table 2. $ceiling(w)$ of $bin(Z)$ B type

$bin(Z)$	$bin(Z+2)$	$bin((Z+2)\gg 1)$	$Log_2(bin((Z+2)\gg 1))$	w	$ceil(w)$
10	100	10	1	1	1
110	1000	100	2	2	2
1110	10000	1000	3	3	3
11110	100000	10000	4	4	4
111110	1000000	100000	5	5	5
1111110	10000000	1000000	6	6	6
11111110	100000000	10000000	7	7	7
…	…	…	…	…	…

(3) $bin(Z)$ 유형 - C type

마지막 유형으로서 $bin(Z)$가 '1'로만 이루어져 있지 않고, LSB가 '0'이고 나머지는 '1'로 이루어져 있지 아니한 경우로서, 이와 같은 형태는 이진수 '1110111' 등과 같이 '0' 이 최소 한 번 이상 MSB 및 LSB를 제외한 위치에서 나오는 경우이다. 이러한 형태의 Z에 있어서 $bin(Z+2)$연산은 그림 3와 같이 Bit(1) 값이 '0' 인 경우에는 $bin(Z)$와 자리 수 변화가 없고, 그림 3과 같이 Bit(1) 값이 '1' 인 경우에는, $bin(Z+2)$ 의 연산 과정에서 그림 4과 같이 자리 올림이 발생하지만, 최소 한번 이상 MSB, LSB를 제외하고 '0' 이 존재하므로, 해당 위치에서 자리 올림이 종료된다. 따라서 두 경우 모두 $\leq n(bin(Z+2))=\leq n(bin(Z))$ 임을 알 수 있다.

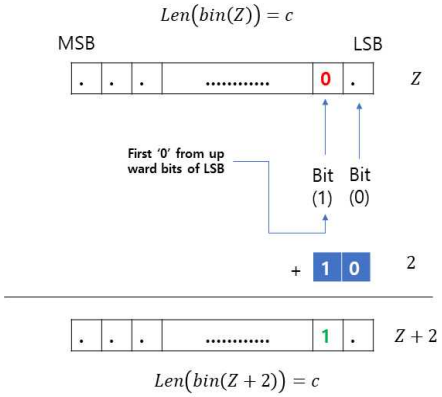

그림 3. 유형1: Z 와 $Z+2$ 의 자릿수 변화 없음
Fig. 3. Type 1: No change in the number of digits in Z and $Z+2$

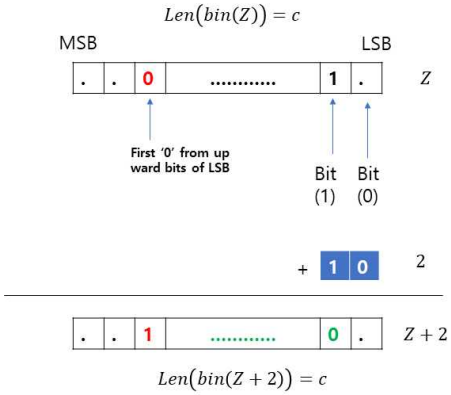

그림 4. 유형 2:Z 와 $Z+2$ 의 자릿수 변화 없음
Fig. 4. Type 2: No change in the number of digits in Z and $Z+2$

부연하자면, $\leq n(bin(Z+2))$와 $\leq n(bin(Z))$가 차이가 있는 경우는(즉 자리수가 변경이 있는 경우는) Bit(1)부터 MSB까지 모든 자리수가 '1'로 세팅되어 있는 경우만 해당한다. 한편 상술한 $bin(Z)$ C type 유형은 아래 표와 같이 $bin(Z)$ A type 및 B type 이 아닌 모든 형태이며, $bin(Z)$ A type과 $bin(Z)$ B type의 $Ceiling(w)$ 값이 동일하다면, 로

그 함수는 증가함수 이므로, 그 사이에 위치한 $bin(Z)$ C type 의 $Ceiling(w)$ 값은 같은 값이어야 하는 것임을 표 3의 음영 표시된 부분을 통해 알 수 있다.

표 3. $bin(Z)$ 가 C 타입의 경우 $Ceiling(w)$ 값
Table 3. $Ceiling(w)$ of C type $bin(Z)$

$bin(Z)$	$bin($ (Z+2))	$bin($ (Z+2)> >1)	$\log_2($ (z+2)> >1)	$w=\log_2\frac{Z+2}{2}$	$ceil(w)$	$bin(Z)$ type
1	11	1	0	0.xxx	1	A
10	100	10	1	1	1	B
11	101	10	1	1.xxx	2	A
100	110	…	…	…	2	C
101	111	…	…	…	2	C
110	1000	100	2	2	2	B
111	1001	100	2	2.xxx	3	A
1000	1010	…	…	…	3	C
…	…	…	…	…	…	C
1101	1111	…	…	…	3	C
1110	10000	1000	3	3	3	B
1111	10001	1000	3	3.xxx	4	A
10000	10010	…	…	…	4	C
…..	…	…	…	…	4	…
11101	11111	…	…	…	4	C
11110	100000	10000	4	4	4	B
11111	100001	10000	4	4.xxx	5	A
100000	…	…	…	…	5	C
…...	…	…	…	…	…	C
111101	…	…	…	…	5	C
111110	1000000	100000	5	5	5	B
111111	1000001	100000	5	5.xxx	6	A
1000000	…	…	…	…	…	C
…...	…	…	…	…	…	C
1111101	…	…	…	…	…	C
1111110	10000000	1000000	6	6	6	B
1111111	10000001	1000000	6	6.xxx	7	A
…	…	…	…	…	…	…

이때 표 3으로부터 $ceiling(w)$ 값은 $\leq n(bin(Z))-1$임을 알 수 있다. $bin(Z)$='1101' 이면 $ceiling(w)$=3 임을 알 수 있고, 따라서 $M=3$ 임을 알 수 있다. 상술한 바와 같이 $bin(Z)$ 의 타입이 어떤 종류인지와 $\leq n(bin(Z))$는 매우 쉽게 알 수 있으므로, S=1인 경우 $M=ceiling(\log_2(\frac{Z+2^s}{2^s}))$을 IoT 장비의 경우 연산 부담이 큰 실수 함수 연산 등의 과정 없이 즉시 구할 수 있다.

3.1.2 접두 코드 엔코딩 효율화 (K 값 구하기)

3.1.1 절에서 $bin(Z)$ 의 세 가지 유형에 따라 비트 패턴만으로 M을 구하였다면, K 는 상술한 수식 (2)을 이용해 구할 수 있다. 수식 (2)을 살펴보면 정확한 값을 구할 필요가 없이 근사치인 소수점 첫째 자리까지만 구하여도 정확한 K를 구할 수 있으므로 $\sqrt{1+8M}$ 을 구하는 작업의 연산 부담을 줄이면서 근사치를 빠르게 구할 수 있는 뉴턴-랩슨 방법(Newton-Raphson method)을 이용하여 근사 값을 구하면 M 이 매우 커지더라도 연산 부담을 덜 수 있다. 한편 뉴턴-랩슨 방법의 초기 값을 적절히 선택함에 따라 더욱 빠르게 근사 값을 찾을 수 있는 방법이 이미 알려져 있는데[7], 본 연구에서는 그 초기 값을 $bin(1+8M)$ 을 이용하여 적절히 산출하는 방법을 아래와 같이 제안하였다. 먼저 뉴턴-랩슨 방법에 따르면, 수식 (2)의 일부인 $\sqrt{1+8M}=q$ 라고 했을 때, q는 방정식 $x^2-q^2=0$ 의 한 근이다. $f(x)=x^2-q^2$ 로 하면, 점화식은 수식 (15) 과 다음과 같이 계산된다.

$$x_n=\frac{1}{2}x_{n-1}+\frac{q^2}{2x_{n-1}} \quad (15)$$

이때 뉴턴-랩슨 방법에서 빠르게 근사 값을 찾기 위하여 x_0을 어떤 값으로 할지가 매우 중요하며[7], 본 연구에서는, $f(x)=x^2-q^2$ 함수에서, x절편이 q이고 구하고자 하는 최종 해이므로, $q(=\sqrt{1+8M})$와 절대적 거리가 가장 가까운 수를 x_0 로 선정하는 것이 유리함은 자명하다. 그러나 본 연구에서는 실용적으로 경량화 하여 비트 패턴만으로 적절한 수준의 x_0을 구하는 방법을 아래와 같이 제안하였다.

먼저, $q(=\sqrt{1+8M})$에서, q^2=$1+8M$ 임을 알 수 있고, $8M$ 은 $bin(M)$을 좌측으로 3비트 shift 한 값

과 같고, 1을 여기에서 더하여 쉽게 계산할 수 있으므로 $bin(1+8M)$의 패턴에 대해서 분석하였다. 물론 $q \leq q^2 (\because q \geq 1)$이고 q와q^2 사이에 임의의 a 값에 대하여 $|q-a|$ 을 최소화 할 수 있는 a을 구하여 x_0을 선정하여야 하는 것은 최적일 것이나, 엄밀한 수학적 최적 값을 찾는 것은 본 연구의 주제를 벗어나므로 실용적으로 비트 패턴 분석을 통해 빠르게 x_0을 선정할 수 있는 방안으로서, q^2을 활용하였다.

아래 표 4와 같이 $q^2(=1+8M)$ 과 $\log_2(q^2)$ 와의 관계를 살펴보면, q^2 은 홀수 이므로 표 4에서 같이 어두운 색으로 표시된 경우 중에만 존재할 것이다. 그러나 표 4는 아래 수식 (16)의 관계가 일반화 될 수 있음을 보이기 위해 1씩 증가하는 일반적인 상황으로 표현하였다. 표 4를 통해, $p = truncate(\log_2(q^2)))$ (단,$truncate(x)$는 x의 소수점 이하 값을 버리는 함수로 정의) 라고 할 때, 다음 수식 (16)와 같은 부등식이 성립한다.

$$2^p \leq q^2 \leq 2^{(p+1)} - 1 \quad (16)$$

이때 수식 (16)와 표 4로부터 $p = \leq n(bin(q^2)) - 1$ 임을 알 수 있다.

표 4. q^2 의 이진수 형태에 있어서 $\log_2(q^2)$값의 관계
Table 4. The relation of the value of $\log_2(q^2)$ in the binary form of q^2

q^2	bin(q^2)	$\log_2(q^2)$	$p = trun(\log_2(q^2$
1	1	0	0
2	10	1	1
3	11	1.5849625	1
4	100	2	2
5	101	2.32192809	2
6	110	2.5849625	2
7	111	2.80735492	2
8	1000	3	3
9	1001	3.169925	3
10	1010	3.32192809	3
11	1011	3.45943162	3
12	1100	3.5849625	3
13	1101	3.70043972	3
14	1110	3.80735492	3
15	1111	3.9068906	3
16	10000	4	4
17	10001	4.08746284	4
…	…	…	…

또한 $2^p \geq 1$ 이므로, 수식 (16)으로부터 수식(17)도 성립한다.

$$2^{\frac{p}{2}} \leq q \quad (17)$$

따라서 원하는 해 $q = \sqrt{1+8M}$ 이하의 수이지만, $bin(q^2) = bin(1+8M)$을 통해 $p = \leq n(bin(q^2)) - 1$ 를 구하여 얻어지는 $2^{\frac{p}{2}}$ 값을 x_0 로 선정하여 수행하면 보다 더 빠르게 수렴할 것으로 기대된다. 실제로 $M = 10000000000$ 일 때, 수식 (2)에 포함된 $\sqrt{1+8M} = \sqrt{1+8\times 10000000000}$ 을 제안된 형태의 뉴턴-랩슨 방법으로 구할 경우 $bin(1+8M)$은 '1001010100000010111110010000000000001' 이므로 $p = \leq n(bin(1+8M)) - 1$ 으로 부터 $p = 36$ 이 계산되고, $2^{\frac{36}{2}} = 262144$ 이므로, $x_0 = 262144$로 하여 수식 (15)의 점화식을 계산하면, 표 5와 같이 실제 값 282842.7125…에 3회의 계산으로 빠르게 근접함을 알 수 있다.만약, 위에서 특별한 기준 없이

표 5. 뉴턴-랩슨 방법을 이용한 x_0 선택에 따른 $\sqrt{1+8M}(=q)$ 연산 비교
Table 5. Comparison with calculations of $\sqrt{1+8M}(=q)$ using the Newton-Raphson method by selecting x_0

$x_n = \frac{1}{2}x_{n-1} + \frac{q^2}{2x_{n-1}}$	x_n	x_n
x_0	262144	80000000001
x_1	283659.8906…	40000000001
x_2	282843.8896…	20000000002
x_3	282842.7125…	10000000003
x_4	282842.7125…	5000000005
…	…	…
x_{22}	…	282842.7125 …

$x_0 = 1 + 8M$을 대입하여, $x_0 = 8 \times 10000000000 + 1$ 로 진행할 경우, 22회의 계산을 거쳐야 282842.7125… 에 접근하였다.

3.1.3 접미 코드 엔코딩 효율화

BL-beta 코드의 선행 연구[1,3]에서는 S+(M-1) 비트 길이를 할당 받은 접미 코드를 아래와 같은 수식 (18)를 통해 구한 뒤,

$$suffix = Z - 2^s \times (2^{(M-1)} - 1) - 1 \quad (18)$$

$bin(suffix)$에 '0'을 left padding하여 S+(M-1) 비트로 하였다. S=1일 경우, 접미 코드가 차지하는 비트 수는 $\leq n(bin(suffix) = 1 + (M-1) = M$ bits 이다. 또한 접미 코드 값은 수식 (18)로부터 아래 수식 (19)와 같이 전개 된다.

$$\begin{aligned} suffix &= Z - 2 \times (2^{(M-1)} - 1) - 1 \\ &= Z - 2^M + 2 - 1 \\ &= Z - 2^M + 1 \end{aligned} \quad (19)$$

접미 코드 또한 접두 코드와 같이 $bin(Z)$ 의 type에 따라 달리 생성되는데, 다른 점은 크게 2가지로 나뉜다는 점이다.

(1) $bin(Z)$ 가 A type 인 경우의 접미코드

만약 $bin(Z)$ 가 A type이면, $bin(Z)$ 및 $bin(2^M)$는 그림 5와 같은 패턴이며, $Z - 2^M = -1$ 임을 알 수 있고, $suffix = Z - 2^M + 1 = 0$ 이다. 또한 $\leq n(bin(suffix)) = M$ 이므로, M 비트의 '0'으로만 이루어진 이진수가 접미 코드가 된다.

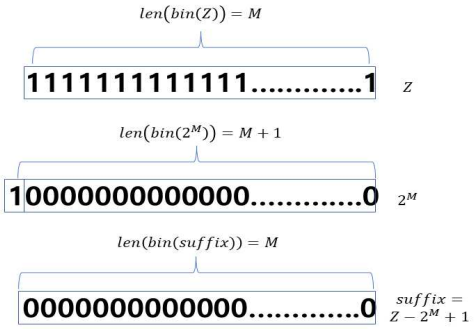

그림 5. Z 및 2^M 으로부터 suffix 계산

Fig. 5. calculate suffix from Z and 2^M

(2) $bin(Z)$가 B type 또는 C type인 경우 접미 코드

한편 $bin(Z)$ 가 B 또는 C type 이면, 3.1.1절에서 보였듯이 $M = \leq n(bin(Z)) - 1$ 이다. 이로부터 $\leq n(bin(Z)) = M + 1$이다. 따라서 $bin(Z)$는 그림 6과 같이 '1'로 시작하는 M+1자리의 이진수로 일반화 된다.

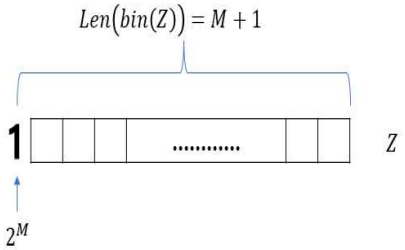

그림 6. bin(Z)가 B 또는 C type일 경우 형태

Fig. 6. Bit pattern when bin(Z) is B or C type

단, $bin(Z)$가 A type이 아니므로 M+1개의 비트가 모두 '1'이 되는 경우는 존재하지 않는다. 따라서 $Z - 2^M$ 을 수행하면 그림 7과 같이 $bin(Z)$의 MSB 1비트가 '0'으로 세팅된다.

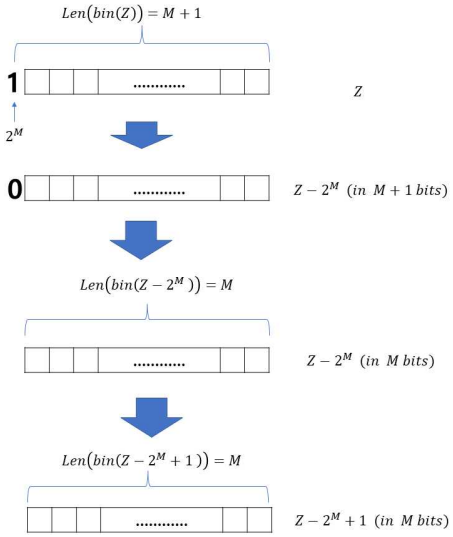

그림 7. bin(Z)가 B 또는 C type일 때, 접미 코드를 구하는 과정

Fig. 7. Process of obtaining suffix code when bin(Z) is of type B or C

다음으로 $Z-2^M+1$ 을 수행하기 위해 1을 더하여도 $\leq n(bin(Z-2^M+1)) = M$ 가 된다. 즉, $Z-2^M+1$ 의 비트 길이는 M 비트로서 변하지 않는다. 왜냐하면, 그림 8과 같이, M 비트의 bin $(Z-2^M)$ 에는 $bin(Z)$가 B type 또는 C type일 경우에 반드시 M 비트 안에 1개 이상의 '0' 이 존재하기 때문에, LSB에 1을 더하여 자리 올림이 연쇄적으로 발생하더라도 M 비트 내에서 MSB방향으로 처음으로 '0'이 있었던 비트 위치가 '1'로 세팅되면서 자리 올림이 중단 되게 된다. 이와 같은 규칙으로부터 $bin(Z)$가 B type 또는 C type일 경우, 접미 코드는 $bin(Z)$의 MSB의 1비트를 제거한 다음, 1을 더한 수를 M 비트의 이진수로 나타낸 값이란 것을 알 수 있다.

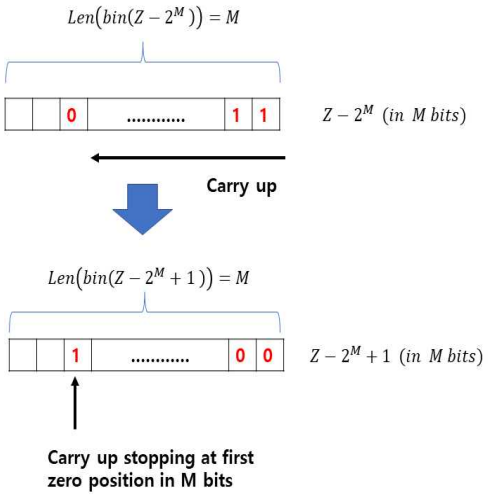

그림 8. $Z-2^M$ 과 $Z-2^M+1$ 의 자리수가 동일한 이유

Fig. 8. The reason why $Z-2^M$ and $Z-2^M+1$ have the same number of binary digits

3.2 BL-beta 코드 디코딩 성능 개선

3.1 절에서와 같이 $bin(Z)$의 패턴 및 간단한 비트 연산을 통해 BL-beta 코드의 접두 코드(prefix)와 접미 코드(suffix)를 구할 수 있었다. 한편, 그림 8을 상세히 관찰하면, 그림 9와 같이 새로운 규칙성을 발견할 수 있는데, 이 규칙성으로부터 $bin(Z)$를 BL-beta 코드 자체의 패턴으로부터 쉽게 확인이 가능하였다. 예를 들어 BL-beta 코드가 '11100011110100001001000001' 라면, MSB방향에서 LSB방향으로 스캔하면서 '01'을 처음 만났을 때, '1'부터 LSB 까지를 취하여 그 값에서 1을 빼주면 Z가 된다. 혼동하지 말아야 할 것은 '01'을 처음 만났을 때, '1'을 포함하여 LSB까지의 비트 영역은 접미 코드의 최상위 비트에 1비트를 추가한 비트열이며 엄밀히는 접미 코드가 아니다. 그러나 BL-beta 코드로부터 접두 코드를 구하고 이로부터 M을 구하면, 접미 코드 영역 및 크기를 쉽게 확인 가능하므로, 그 앞 1비트를 포함한 비트 영역의 패턴으로부터 즉시 $bin(Z)$를 구할 수 있다는 점이 본 연구를 통해 새롭게 확인한 점이다. 매우 큰 크기의 $bin(Z)$를 최종적인 code-num

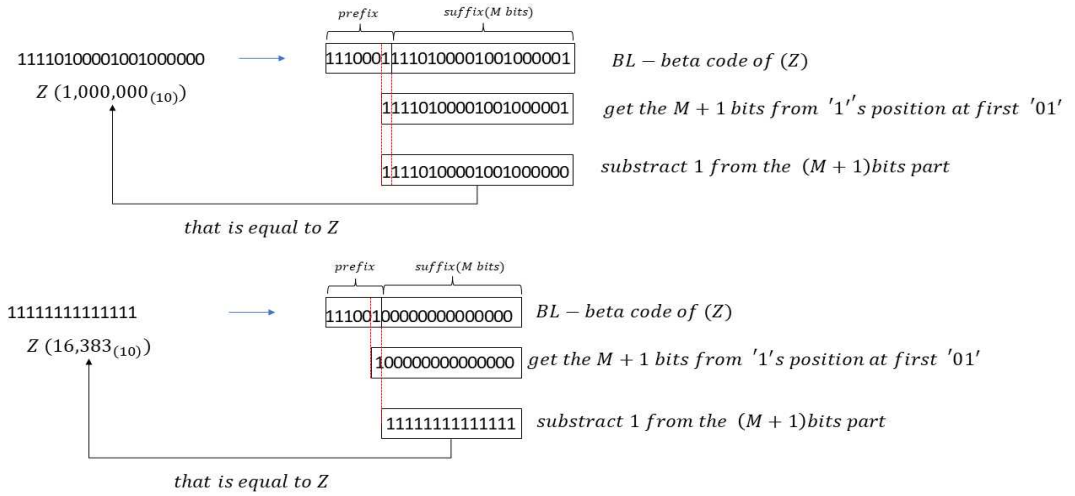

그림 9. 접미코드에서 MSB방향으로 1비트를 추가로 연결한 비트패턴으로 부터 Z를 구하는 계산과정 일반화
Fig. 9. Generalization of the calculation process for obtaining Z from a bit pattern in which 1 bit is additiona lly connected in the MSB direction from the suffix code

인 십진수로 변환하는 알고리즘은 이미 알려져 있으므로[8], 본 연구에서는 언급을 생략하도록 하겠다.

4. 성능평가

4.1 테스트 환경

본 연구에서 개발한 BL-beta 코드 엔코더 및 디코더는 Microsoft 사의 Visual Studio Community 2022 (64-bit) 버전 17.3.2의 C# 으로 작성하였고 release모드에서 테스트하였다. 테스트 하드웨어 사양은 Intel(R) Core(TM) i5-8400 CPU@ 2.80GHz, 2.81GHz 사양의 16GB 램, 윈도우즈 10(64비트 운영체제)로 수행하였다. 경량화와 기존 로직과의 공정한 비교를 위해 함수 도입부와 값 리턴 부를 공통 로직을 사용하였으며, 경량화 부분외의 로직은 가급적동일한 알고리즘에 따르도록 하여 테스트하였다.

4.2 BL-beta 코드 엔코딩 성능비교

엔코딩 성능 테스트를 위해 Code-num Z를 1에서부터 각 100000, 1000000, 10000000, 100000000까지 연속적으로 엔코딩 하였을 때 소요되는 시간을 계산하였으며 모두 2회씩 테스트하여 그 평균값을 구하였다. 테스트 결과는 Fig. 10 및 표 6과 같다.

성능 테스트 결과, 평균 약 24.8% 엔코딩 시간이 절감됨을 확인할 수 있었다. 한편 Code-num 연산이 많아질수록 점진적으로 엔코딩 시간 절감 율이 낮아짐을 알 수 있었는데 이는 경량화 구현 방식이 테스트 환경에서는 비트 패턴을 문자열 형식(String 형식)으로 처리하였기 때문으로 여겨진다. 문자열 형식의 경우 처리해야할 대상 문자열 길이가 길어짐에 따라 성능이 저하되는 데[9], 이러한 특성 때문으로 예상된다. 최적의 경량화를 위하여 비트 패턴을 문자열 형식으로 구현하기보다 바이트내의 실제 비트로부터 비트 연산자 등을 통해 비트 패턴을 직접 확인하는 방식으로 구현하면 성능저하 없이 작동할 것으로 예상 된다.

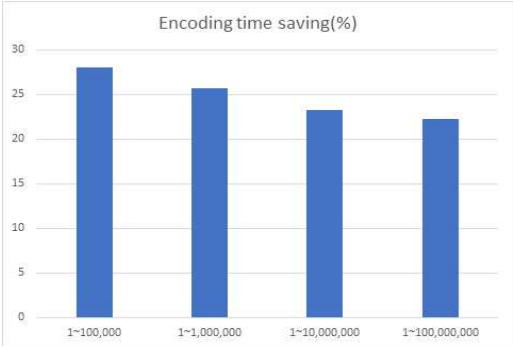

그림 10. 경량화 및 기존 방식에 따른 엔코딩 시간 절감율 비교 그래프

Fig. 10. Comparison graph of encoding time reduction rate according to lightweight and existing methods

표 6. 경량화 및 기존 방식에 따른 엔코딩 시간 절감 율 비교

Table 6. Comparison of encoding time reduction rate according to lightweight and existing methods

연속엔코딩범위 (Z)	1~100000	1~1000000	1~10000000	1~100000000
경량화 알고리즘 소요시간 (miliseconds)	12.89	93.53	1127.02	154373.88
기존 알고리즘 소요시간 (miliseconds)	12.22	129.90	1518.54	198524.33
평균 엔코딩 시간 절감율 (%))	28	25.75	23.25	22.25

4.3 BL-beta 코드 디코딩 성능비교

디코딩 성능 테스트를 위해 엔코딩 성능 테스트와 동일한 방식으로 Code-num Z를 1에서부터 각 100000,1000000,10000000,100000000 까지 연속적으로 디코딩 하였을 때 소요되는 시간을 계산하였으며 모두 2회씩 테스트하여 그 평균값을 구하였다. 테스트결과는 Fig. 11 및 표 7과 같다.

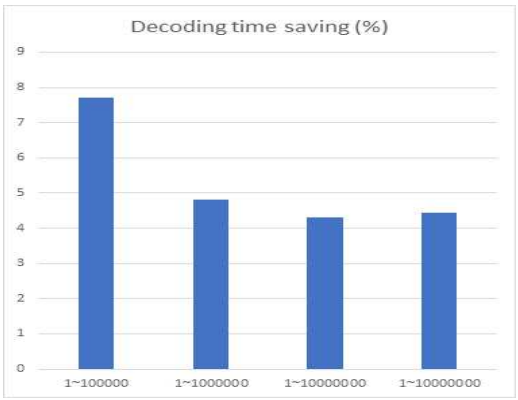

그림 11. 경량화 및 기존 방식에 따른 디코딩 시간 절감율 비교

Fig. 11. Comparison of decoding time reduction rate according to lightweight and existing methods

성능 테스트 결과, 평균 5.3% 가량 디코딩 시간이 절감됨을 확인할 수 있었다. 엔코딩과 비교하여 시간 절감 율이 상대적으로 크지 않는 것은 접두 코드를 확인하는 부분은 기존 방식과 동일하며, 디코딩 단계에서는 연산 부담이 큰 로그 함수, 실수연산, 나눗셈 연산이 없기 때문에 최적화 여지가 크지 않기 때문이다. 그럼에도 불구하고, 기존 연구에서는 수식 (7)와 같이 접미 코드(suffix)를 더하고 추가적인 지수 연산 과정을 거쳐야 code-num Z를 구할 수 있었는데 개선된 경량화 알고리즘에서는 접미 코드부와 그 앞 1비트 영역으로부터 Z를 직접 구할 수 있었기 때문에 속도가 빨라졌음을 알 수 있다. 한편 엔코딩 테스트와 시간 절감에 있어 동일한 양상이 나타났는데, code-num Z가 커지고 연산 량이 많아질수록 점진적으로 디코딩 시간 절감 율이 낮아짐을 알 수 있었는데 이 또한 경량화 구현 방식이 테스트 개발 환경에서는 비트 패턴 분석과 처리를 문자열 형식(String 형식)으로 처리하였기 때문으로 여겨진다.

표 7. 경량화 및 기존 방식에 따른 디코딩 시간 절감 율 비교
Table 7. Comparison of decoding time reduction rate according to lightweight and existing methods

연속디코딩범위 (Z)	1~100000	1~1000000	1~10000000	1~100000000
경량화 알고리즘 소요시간 (miliseconds)	156.45	1775.67	20537.39	229722.24
기존 알고리즘 소요시간 (miliseconds)	169.48	1865.83	21461.04	240402.34
평균 디코딩 시간 절감율 (%))	7.7	4.8	4.3	4.45

5. 결 론

본 연구를 통해 IoT 장비로부터 생성 되는 감지(sensing) 데이터의 무 손실 실시간 압축 전송에서 활용 가능한 BL-beta코드에서 연산 부담이 큰 지수 연산, 로그 연산을 비트 패턴, 비트 shift연산만으로 경량화 하여 빠르게 BL-beta 코드로 엔코딩 및 디코딩이 가능한 방법을 새롭게 제안하였으며, 제곱근 연산의 경우에는 기존의 뉴턴-랩슨 방법을 활용하되, 비트 패턴을 이용하여 실용적으로 빠르게 초기 값을 설정할 수 있는 방법을 새롭게 제안하였다. 제곱근 연산의 경우에는 비트 연산만으로도 가능한 개평법(digit-by-digit calculation for computing square roots)도 추가 적용이 가능할 것으로 여겨진다. 한편 본 연구는 BL-beta 코드에서 가장 간단한 형태인 S=1인 코드에 대해서 연산 부담을 줄이고 성능을 향상시켰으며, S가 1보다 큰 경우에도 유사한 방법의 비트 패턴 분석으로 경량화가 가능할 것으로 예상되나, 이 부분에 대해서는 추가적인 구체적인 연구를 통해 S가 1보다 큰 경우에도 비트 패턴 분석을 통해 본 연구를 통해 확인된 규칙을 일반화 할 수 있는지에 대해 연구가 필요하다.

REFERENCES

[1] J. H. Kim, S. Yeo, J. W. Kim, K. S. Kim, T. K. Song, C. H, Yoon, J. H. Sung, "Real-Time Lossless Compression Algorithm for Ultrasound Data Using BL Universal Code. ",Sensors, 18, 3314. https://doi.org/10.3390/s18103314, 2018.

[2] Guneet Bedi, Ganesh Kumar Venayagamoorthy, Rajendra Singh, Richard Brooks, Kuang-Ching Wang, "Review of Internet of Things (IoT) in Electric Power and Energy Systems", IEEE Xplore Digital Library, pp. 9-12, accessed Dec.8, 2022, https://ieeexplore.ieee.org/ielaam/6488907/8334665/8281479-aam.pdf.

[3] J. H. Kim, "New high-efficient universal code(BL-beta) proposal for compressed data transferring of real-time IoT sensing or financial transaction data", Journal of Korea Institute of Information, Electronics, and Communication Technology (KIIECT), Vol 11, no.4,pp.421-429, 2018.

[4] M. J. Lee, S. B. Oh, J. H Kim,"Comparison and analysis of compression algorithms to improve transmission efficiency of manufacturing data",Journal of the Korea Institute of Information and Communication Engineering, Vol. 26, No. 1,pp.94-103, Jan. 2022.

[5] J. Lee, "Analysis of the Hardware Structures of the IoT Device Platforms for the Minimal Power Consumption", Journal of Internet of Things and Convergence Vol. 6, No. 2, pp. 11-18, 2020.

[6] "Convert big decimal number to binary." accessed Dec.8, 2022, http://www.dec2bin.com.

[7] C. S. Choi, J. Y. Lee, Y. L. Kim,"Initial Point Optimization for Square Root Approximation based on Newton-Raphson Method", Journal of the Institute of Electronics Engineers of Korea. SD, v.43 no.3=no.345 , pp.15-20,2006.

[8] "Convert big binary to decimal number." accessed Dec. 8,2022, http://www.bin2dec.com.

[9] "Use Visual C# to improve string concatenation performance",accessed Dec. 22,2022,https://learn.microsoft.com/en-us/troubleshoot/developer/visualstudio/csharp/language-compilers/string-concatenation.

저자약력

김 정 훈 (Jung-Hoon Kim) **[정회원]**

- 2002년 2월 : 서울대학교 약학과 (약학사)
- 2010년 2월 : 서울대학교 보건학과 (보건학 석사)
- 2013년 2월 : 서울대학교 보건학과 (보건학 박사 수료)
- 2019년 2월 : 경희대학교 컴퓨터공학과(SW융합학 석사)
- 2016년8월~현재 : 바이너리랩(주) 대표이사

〈관심분야〉 정보이론, 정보보호, 의료정보