KIPS Trans. Comp. and Comm. Sys.
Vol.10, No.9 pp.235~242
ISSN: 2287-5891 (Print), ISSN 2734-049X (Online)

https://doi.org/10.3745/KTCCS.2021.10.9.235

Data Scrambling Scheme that Controls Code Density with Data Occurrence Frequency

Choulseung Hyun† · Gwanil Jeong†† · Soowon You†† · Donghee Lee†††

ABSTRACT

Most data scrambling schemes generate pure random codes. Unlike these schemes, we propose a variable density scrambling scheme (VDSC) that differentiates densities of generated codes. First, we describe conditions and methods to translate plain codes to cipher codes with different densities. Then we apply the VDSC to flash memory such that preferred cell states occur more than others. To restrain error rate, specifically, the VDSC controls code densities so as to increase the ratio of center state among all possible cell states in flash memory. Scrambling experiments of data in Windows and Linux systems show that the VDSC increases the ratio of cells having near-center states in flash memory.

Keywords : Cell State, Data Scrambling, Flash Memory, Variable Density

데이터 출현 빈도를 이용하여 코드 밀도를 조절하는 데이터 스크램블링 기법

현 철 승† · 정 관 일†† · 유 수 원†† · 이 동 희†††

요 약

기존 데이터 스크램블링 기법은 랜덤한 코드를 생성한다. 이와 다르게 우리는 생성하는 코드의 밀도를 다르게 만드는 가변 밀도 스크램블링 기법을 제안한다. 먼저 코드 밀도를 다르게 만드는 조건과 방법에 대해 설명한다. 다음으로 가변 밀도 스크램블링 기법을 플래시 메모리에 적용하여 특정 셀 상태가 더 많이 발생하도록 한다. 특히 플래시 메모리의 에러율을 제한하기 위하여, 가변 밀도 스크램블링 기법은 코드의 밀도를 조절하여 모든 셀 상태 중 중간 상태를 가지는 셀 비율을 높일 수 있다. 윈도우즈와 리눅스 시스템의 데이터에 가변 밀도 스크램블링 기법을 적용하였으며, 실험 결과는 가변 밀도 스크램블링 기법이 중간과 가까운 상태를 가지는 셀의 비율을 증가시킴을 보여준다.

키워드 : 셀 상태, 데이터 스크램블링, 플래시 메모리, 가변 밀도

1. 서 론

통신에서 데이터 스크램블링은 주어진 대역폭을 사용하기 위해 그리고 저장장치에서는 중요한 정보를 감추거나 에러율을 일정 수준 이내로 제한하기 위해 사용된다. 이러한 목적으로 사용되는 대부분의 데이터 스크램블링 기법은 데이터를 순수하게 랜덤화한다.

플래시 메모리에서는 고유의 특성 때문에 읽기 또는 프로그램 간섭이나 리텐션 에러가 발생하며, 이러한 에러는 플래시 메모리 셀에 저장된 데이터의 형태와 관계가 있다[1-4]. 특히 프로그래밍 된 셀의 상태에 따라 에러율이 달라지며, 이러한 에러 문제를 해결하는 방법들 중 하나는 플래시 메모리 셀에 저장된 데이터가 편향적이지 않도록 데이터를 랜덤화하는 스크램블링 기법을 적용하는 것이다[5-7].

기존의 스크램블링 기법들은 대부분 데이터를 순수하게 랜덤화한다. 이와는 다르게 본 논문에서는 스크램블링을 수행하면서 생성하는 코드의 밀도를 다르게 생성하는 가변 밀도 스크램블링 기법을 제안한다. 그러나 코드 밀도를 다르게 만들기 위해서는 몇 가지 조건이 필요하며, 논문에서는 먼저 코드 밀도를 다르게 만들기 위한 조건과 방법에 대해 설명한다. 구체적으로 데이터가 가져야 하는 조건, 적절한 자료구조, 그리고 비균일 확률 밀도 함수를 이용한 랜덤넘버 생성에 대해 설명한다.

※ This Research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education(2016009066).
† 정 회 원 : 서울시립대학교 컴퓨터과학부 연구교수
†† 비 회 원 : 서울시립대학교 컴퓨터과학부 석사과정
††† 비 회 원 : 서울시립대학교 컴퓨터과학부 교수
Manuscript Received : February 3, 2021
Accepted : March 3, 2021
* Corresponding Author : Donghee Lee(dhl_express@uos.ac.kr)

다음으로 가변 밀도 스크램블링 기법을 플래시 메모리에 적용한다. 데이터를 순수하게 랜덤화하는 기존 스크램블링 기법은 플래시 메모리 셀들이 균등하게 모든 상태를 가지도록 만든다. 그 결과 평균적으로 셀들은 중간 상태를 가지며 인접 셀과의 상태 차이를 일정 수준으로 제한한다. 모든 셀 상태가 균등하게 나타나도록 하는 기존 스크램블링 기법과 다르게 가변 밀도 스크램블링 기법은 중간 상태를 가지는 코드를 더 많이 생성하며, 더 많은 셀이 중간 상태를 가지게 만든다.

실제 사용되는 컴퓨터 시스템의 데이터를 이용하여 가변 밀도 스크램블링 기법의 효과를 측정하였다. 먼저 윈도우즈와 리눅스 환경에서 중요 폴더에 대해 데이터 코드의 출현 빈도를 조사하였다. 조사 결과 데이터 코드들간에 출현 빈도의 차이가 존재하며, 이러한 차이를 이용하여 스크램블링을 수행할 때 코드의 밀도를 조절하는 실험을 수행하였다. 특히 플래시 메모리 셀 상태를 중간 상태로 만드는 코드를 더 많이 생성하도록 하였으며, 실험 결과는 더 많은 셀의 상태가 중간 상태 또는 이와 가까운 상태를 가지고 있음을 보여주었다.

논문의 구성은 다음과 같다. 2장에서 관련 연구를 설명하며, 3장에서 코드 밀도를 조절하기 위한 회전판 기법을 설명한다. 4장에서 밀도를 조절하는 또 다른 구현 방법인 클래스 분할 기법을 설명하고 5장에서 플래시 메모리에 적용한 실험 결과를 제시한다. 마지막으로 6장에서 논문을 끝맺는다.

2. 관련 연구

통신이나 저장장치에서 데이터 변환은 널리 사용된다. 예를 들면, 통신을 위해 8B/10B 방식으로 데이터를 1:1로 변환하기도 하며, 대역폭을 모두 사용하도록 데이터를 랜덤화하는 스크램블링 기법을 사용하기도 한다. 일부 저장장치의 경우에도 스크램블링 기법을 사용하여 저장하는 데이터를 랜덤화한다.

플래시 메모리는 비트 변환마다 에너지를 사용하고 수명이 단축된다. 이러한 비트 변화를 줄이기 위해 제안된 Bit-flipping 기법은 저장될 데이터를 일정 크기의 청크로 분할하고 셀에 저장되어야 할 비트에 대해 미리 정의된 조건에 따라 비트를 변경할 것인지를 결정한다. 비트가 정의된 조건에 따라 변경되어 저장되고 이 청크가 플리핑되었는지 확인하기 위한 플리핑 비트가 플래시 메모리 페이지의 OOB영역에 저장된다. 이에 더해 플리핑 비트는 데이터를 읽기 위해 반드시 보호해야 한다. 이를 위해 추가적인 ECC 정보가 저장될 공간이 있어야 한다[8-10].

플래시 메모리는 읽기, 프로그램, 그리고 소거 연산을 지원한다. 읽기와 프로그램 연산은 페이지 단위로 이루어지며, 소거 연산은 페이지 집합인 블록 단위로 수행된다. 플래시 메모리의 초기 상태는 소거 상태로 프로그램 이전에 반드시 소거 연산이 수행되어야 한다. Fig. 1은 플래시 메모리 셀 상태를 보여준다. 셀 당 2비트를 저장하는 MLC 플래시 메모리 셀은 소거 후 S0 상태를 가지며, 프로그램 동작은 셀의 임계전압을 바꾸어 셀 상태를 S1-S3으로 바꾼다.

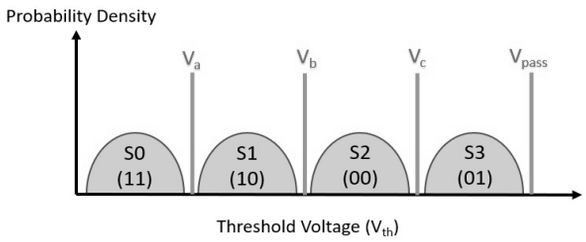


Fig. 1. Cell State Distribution of MLC Flash Memory

프로그래밍할 때 바꾸는 상태 차이를 Programming Distance(PD)로 정의할 수 있다. 이러한 PD가 작을수록 프로그램 속도가 빨라지고 셀의 손상을 낮추어 수명이 늘어난다[1,11]. 이러한 목적으로 다양한 데이터 변환기법들이 연구되었다[8-10,12-15]. ACS(Adaptive code selection) 기법은 페이지를 프로그래밍하기 전에 0의 개수를 카운트하고 그 개수가 페이지 크기의 절반보다 크면 셀의 변화가 작아지도록 코딩 방법을 변경한다[12]. 이와 유사하게 NRC(Nibble remapping coding) 기법은 기록할 데이터를 세그먼트로 나누고 세그먼트에 출현하는 코드의 빈도에 따라 변환 데이터를 결정한다. 그리고 변환 정보는 플래시 메모리 페이지의 OOB영역에 기록된다[13]. ELSE(Endurance enhancing lower state encoding) 기법은 페이지 간의 관계 및 데이터 의존성을 활용하여 셀이 낮은 전압으로 프로그램이 되도록 데이터를 변환하여 기록한다. 특히 MLC나 TLC에서 페이지 간의 관계를 고려하여 변환 동작을 수행한다[15].

플래시 메모리의 경우 PD를 작게 만드는 것이 목표가 아닐 수 있다. 예를 들면 플래시 메모리에서 읽기 동작을 수행할 때 목표 셀과 같은 컬럼에 있는 셀에는 Fig. 1의 바이패스 전압이 가해진다. 그리고 S0와 같이 셀이 가지는 임계전압과 바이패스 전압 간의 차이가 클수록 읽기 간섭 에러 확률이 높아진다[16]. 반대로 셀이 S3 상태를 가지는 경우 플로팅 게이트의 전자가 감소하면서 발생하는 리텐션 에러 확률이 높다[16]. 이처럼 셀이 양 극단 상태를 가지는 경우 에러 확률이 높다고 할 수 있다. 물론 셀이 S1이나 S2와 같은 중간 상태를 가질 때도 프로그래밍 방식에 따라 에러 확률이 높게 나타나기도 하지만, 이는 프로그래밍할 때 증가시키는 전압을 변경하여 조절할 수 있다. 이러한 점을 고려하여 다음 절에서 설명하는 가변 밀도 스크램블링 기법은 셀 상태를 중간에 가깝게 만들려고 한다. 그렇지만 중간이 아닌 다른 상태가 에러율 감소에 도움이 된다면 가변 밀도 스크램블링 기법은 셀을 원하는 상태로 만들 수 있다.

플래시 메모리에서 사용하는 스크램블링 기법은 에러율이 높은 데이터 패턴의 생성을 제한하기 위해 0과 1의 개수를 랜덤화한다. 이와 같은 데이터 랜덤화는 데이터와 함께 저장되는 에러 교정 코드의 교정 능력 이내로 에러율을 제한한다. 대다수의 스크램블링 기법들은 성능과 비용상의 이유로 XOR 연산을 사용한다[5-7]. 이들은 먼저 데이터의 논리주소

를 시드(seed) 값으로 제공하여 키를 생성하고, 저장될 데이터와 키를 대상으로 XOR 연산을 수행한다. 이렇게 랜덤화되어 저장된 데이터에 대해 동일한 키로 XOR연산을 적용하면 원래 데이터 코드를 얻을 수 있다. 보안을 강화하기 위해 키를 생성할 때 전역 카운터 변수를 사용할 수도 있으며, 스크램블링 성능을 높이기 위해 플래시 컨트롤러의 ECC 로직과 연계하거나 플래시 메모리 칩내의 버퍼를 활용할 수도 있다[6,7]. 본 논문에서 제안하는 가변 밀도 스크램블링 기법의 경우에도 일부 기능을 플래시 컨트롤러에서 하드웨어 로직 형태로 구현할 수 있다.

다음 절에서 설명하는 가변 밀도 스크램블링 기법은 바이트 단위로 데이터를 변환하지만, 4비트로 구성된 니블(nibble) 단위로 데이터를 변환할 수도 있다. 또는 바이트를 2개의 니블로 나누고 각 니블마다 변환 테이블을 따로 만들어 변환할 수 있다. 셀 당 2비트 또는 4비트를 저장하는 MLC 또는 QLC 플래시 메모리의 경우 스크램블링 기법은 바이트 또는 니블 단위로 데이터를 변환하여 셀 상태를 조절할 수 있다. 그러나 셀당 3비트를 저장하는 QLC의 경우 셀 상태를 조절하기 위해서는 24비트 단위로 데이터를 변환하거나, 24비트를 3바이트 또는 6개의 니블로 나누고 각 바이트 또는 니블 단위로 변환 테이블을 만들어 변환해야 한다.

3. 가변 밀도 스크램블링: 회전판 기법

데이터를 순수하게 랜덤화하는 기존의 스크램블링 기법과 다르게 가변 밀도 스크램블링 기법은 데이터 스크램블링을 수행하면서 생성하는 코드의 밀도를 다르게 만든다. 이 장에서는 코드 밀도를 조절하기 위해 필요한 조건과 방법에 대해 설명한다.

설명을 위해 스크램블 이전 데이터를 플레인 코드(plain code), 스크램블링이 생성한 데이터를 사이퍼 코드(cipher code)라 부른다. 스크램블링 기법은 플레인 코드를 사이퍼 코드로 변환하며, 논문에서는 8비트 플레인 코드를 8비트 사이퍼 코드로 변환한다고 가정한다. 플래시 메모리에 데이터를 기록하는 경우 메모리에서 위치를 지정하는 주소 addr과 기록하고자 하는 코드 d가 주어진다. 이를 이용하여 데이터 쓰기 동작은 write(addr, d)로 표현할 수 있다. 그리고 데이터 읽기 동작은 addr에서 코드를 읽어 이를 반환하는 read(addr)로 표현할 수 있다. 플래시 메모리의 주소 addr은 로우 번호, 컬럼 번호, 그리고 기타 플레인이나 다이 번호 등의 조합으로 표현될 수 있다. 또는 펌웨어나 소프트웨어 수준에서 블록 번호, 페이지 번호, 페이지 내 오프셋 등의 조합으로 표현할 수도 있다. 스크램블링하는 함수를 encode()라고 하면, 이 함수는 주소 addr과 데이터 d를 인자로 가지는 encode(addr, d)로 표현될 수 있다. 스크램블링 후 데이터를 기록하는 동작은 write(addr, encode(addr, d))로 표현된다. 디스크램블링을 수행하는 함수를 decode(addr, e)라고 하면, 디스크램블링을 수행하면서 데이터를 읽어오는 동작은 decode(addr, read(addr))로 표현할 수 있다. 위에서 read(addr)은 주소 addr로부터 사이퍼 코드를 읽어 decode() 함수에 전달하며, decode() 함수는 사이퍼 코드를 플레인 코드로 변경한 후 이를 반환한다.

코드의 밀도를 다르게 만들기 위해서는 다음 세 가지 조건과 방식이 충족되어야 한다. 먼저, 플레인 코드에 비균일성이 존재해야 한다. 플레인 코드가 랜덤한 형태를 가지는 경우 밀도가 다른 사이퍼 코드를 만들어내기 어렵다. 다음으로 플레인 코드의 비균일성을 목적하는 코드 형태로 바꾸기 위한 자료구조가 필요하다. 마지막으로 비균일 확률 밀도 함수를 이용한 랜덤넘버 생성이 필요하다.

본 논문에서는 플레인 코드에서 비균일한 형태를 얻기 위해 플레인 코드들의 출현 빈도를 이용한다. 이러한 플레인 코드의 출현 빈도의 차이를 이용하면 밀도가 다른 사이퍼 코드를 생성할 수 있다. 특히 출현 빈도의 차이에 비례하여 밀도 차이를 만들 수 있다.

다음으로 플레인 코드의 출현 빈도 순서에 맞게 원하는 사이퍼 코드로 변환하는 자료구조가 필요하다. 가변 밀도 스크램블링 기법은 1:1 변환 기법은 아니지만 출현 빈도를 기반으로 밀도를 조절하기 위해 출현 빈도 순서에 맞춰 원하는 코드를 대응시키는 테이블이 필요하다. Fig. 2에서 FT는 8비트 플레인 코드들을 출현 빈도 순으로 정렬한 테이블이다. 8비트 코드 ‘0’, ‘255’ ‘32’, ‘1’ 순으로 출현 빈도가 높으며, 이들은 FT 테이블에서 순위 0, 1, 2, 3을 차지한다. Fig. 2에서 PDT 테이블은 셀당 4비트를 저장하는 QLC에서 셀 상태를 가운데 상태로 만드는 코드순으로 정렬한 것이다. QLC는 S0-S15까지 16개 상태가 존재하며, 이들 상태에 비트 ‘1111’, ‘1110’, ‘1100’, ..., ‘0111’을 할당한다고 가정하였다. 그리고 S0-S15 상태마다 숫자 0-15를 대응시키면, 가운데 상태는 7.5가 된다. 그리고 PDT 테이블은 셀 상태를 가운데 상태인 7.5에 가깝게 만드는 순서로 코드를 정렬한 것이다.

FT와 PDT를 이용하여 플레인 코드를 사이퍼 코드로 변환하는 인코딩 테이블과 역으로 변환하는 디코딩 테이블을 만들 수 있다. Fig. 3에서 ET와 ECC는 인코딩 테이블로서 플레인 코드 ‘1’은 순위 3을 가지고 이에 1:1로 대응되는 사이퍼 코드는 ‘93’이다. 또한 DT와 DCC는 디코딩 테이블로서 사이퍼 코드 ‘93’은 순위 3을 가지며, 이에 1:1로 대응되는 플레인 코드는 ‘1’이다. Fig. 3의 ECC는 Fig. 2의 PDT와 같고 Fig. 3의 DCC는 Fig. 2의 FT와 같으며, 이름만 다를 뿐 같은 테이블이다.

뒤에서 설명하겠지만 스크램블링 기법은 1:1로 대응되는 코드로 변환하지 않고 랜덤넘버로 순위를 조정하여 해당 순위의 코드로 변환하며, 앞으로 이를 쉬프트(shift)라 부른다. 쉬프트 할 때 논리적으로 ECC와 DCC를 환형 테이블로 간주하며, 결국 엔트리 ‘0’과 엔트리 ‘255’는 논리적으로 인접하다. 그런데 이엔트리 ‘0’와 엔트리 ‘255’ 사이에 쉬프트되는 경우 순위가 붕괴된다. 예를 들어 엔트리 ‘0’과 ‘1’, ‘1’과 ‘2’는 순위가 점진적으로 차이가 나지만, 엔트리 ‘0’과 ‘255’는 점진적이지 않고 양 극단의 순위를 가진다. 그 결과 엔트리

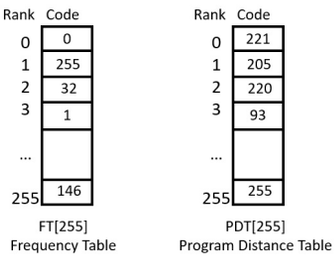
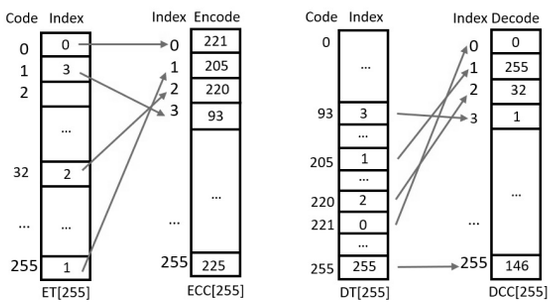


Fig. 2. Frequency and Program Distance Table



(a) Encoding Table (b) Decoding Table

Fig. 3. Basic Tables for Code Translation

'0'과 '255' 사이를 쉬프트할 때 순위가 붕괴하여 원하는 데로 코드 밀도를 조절할 수 없게 된다. 이 문제를 해결하기 위하여 ECC와 DCC를 회전판(Wheel) 형태로 재구성한다. Fig. 4는 ECC와 DCC를 회전판 형태의 EW와 DW로 재구성한 모습과 그 사용 방법을 보여준다. 특히 EW와 DW에서 코드들은 순위가 점진적으로 감소하다가 전진적으로 증가하는 형태를 가지고 있어 어떠한 방향으로 쉬프트 하더라도 순위가 붕괴하지 않는다. ECC와 DCC를 회전판 형태의 EW와 DW로 변환하고 ET와 DT가 이들을 가리키도록 만든다.

Fig. 5는 회전판 기법을 사용하는 인코딩과 디코딩 알고리즘이다. 이 알고리즘과 Fig. 4의 플레인 코드 '255'를 변환 과정을 함께 살펴보자. 먼저 플레인 코드 '255'로부터 ET를 거쳐 회전판 EW에서의 인덱스인 wh_index를 구한다. 그리고 저장주소 addr를 인자로 '-128'에서 '127' 사이의 랜덤넘버를 생성하는 shift() 함수를 호출하여 랜덤넘버 s를 얻는다. 그리고 wh_index에서 s를 더하여 EW 엔트리를 새로 계산하고 이곳에서 사이퍼 코드 '220'을 가져온다.

사이퍼 코드 '220'의 디코딩 과정은 다음과 같다. 사이퍼 코드 '220'으로 부터 DT를 거쳐 회전판 DW에서의 인덱스 wh_index를 구한다. 그리고 저장주소 addr를 인자로 shift() 함수를 호출하여 랜덤넘버 s를 얻는다. 다음으로 wh_index에서 s를 빼서 EW 엔트리를 새로 계산하고 이곳에서 플레인 코드 '255'을 가져온다. shift() 함수는 동일한 주소 addr에 대하여 동일한 값을 반환한다.

가변 밀도 스크램블링 기법이 충족해야 하는 마지막 조건으로 shift() 함수가 가져야 하는 속성에 대해 설명한다. 위에서 설명한 예에서 shift() 함수는 '-128'에서 '127' 사이의 값을 반환한다. 만약 shift()함수가 Fig. 6(a)와 같이 100%의

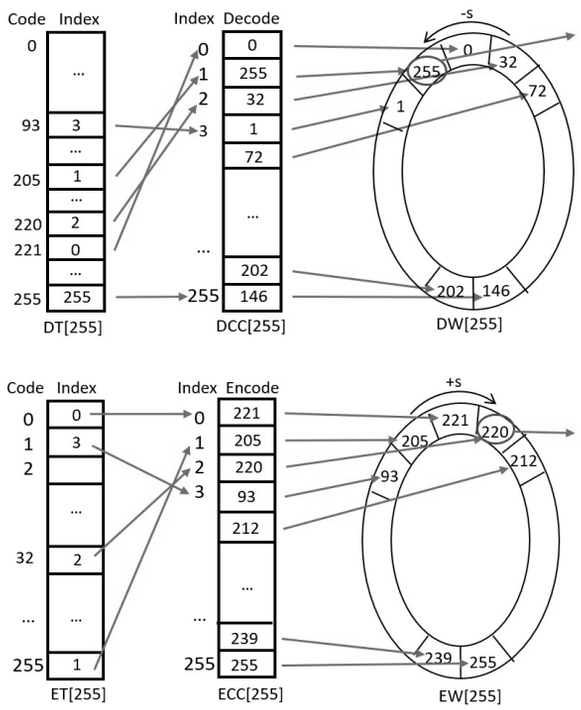


Fig. 4. Tables and Usage Examples of Wheel based Translation

```
int encode(addr, '255') {
        wh_index = ET[255];
        s = shift(addr);
        wh_index += s;
        /* return 220 in this example*/
        return EW[rotate(256, wh_index)];
}
int decode(addr, '220') {
        wh_index = DT[220];
        s = shift(addr);
        wh_index -= s;
        /* return 255 in this example*/
        return DW[rotate(256,  wh_index)];
}
int rotate(int r, int n) {
        while (n >= r) n -= r;
        while (n <0) n += r;

        return n;
}
```

Fig. 5. Implementation Example of Encode/decode Functions for Wheel Based Translation

확률로 '0'을 반환한다면 제안하는 기법은 결국 1:1로 변환하는 기법이 된다. 그렇지 않고 Fig. 6(b)와 같이 '-128'에서 '127' 사이의 값을 동일한 확률로 반환하면 제안하는 기법은 순수한 랜덤화 기법과 동일하다. 만약 shift() 함수가 Fig. 6의 (a), (c) 또는 (d)와 같이 균일하지 않은 확률 밀도에 따라 값을 반환한다면, 특히 0에서 멀어질수록 그 값의 리턴 확률

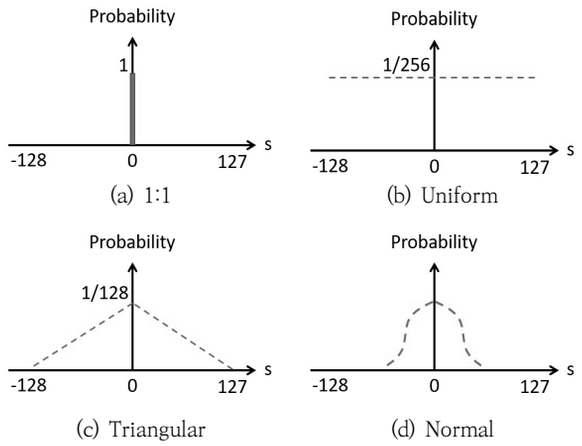


Fig. 6. Various Probability Distributions that can be used in Shift() Function

이 떨어진다면 순위가 높은 코드의 밀도가 높아진다. 즉 1:1로 대응되는 코드에서 가까운 곳으로 쉬프트될 확률이 먼 곳으로 쉬프트될 확률보다 높아 1:1로 변활될 때 가지는 순위 관계가 생성되는 사이퍼 코드들 사이에도 존재한다.

코드들이 가지는 순위 관계를 정형화된 형태로 표현하면 다음과 같다. 두 플레인 코드 d_i와 d_j사이에 순위 관계가 존재하고, d_i의 순위가 d_j보다 높으면 이를 $d_i > d_j$로 표현한다. d_i가 n번 인코드되어 만들어진 사이퍼 코드 집합과 d_j가 n번 인코드되어 만들어진 사이퍼 코드 집합 E_i와 E_j가 다음과 같다.

$$E_i = e_0^i, e_1^i, e_2^i, ..., e_{n-1}^i \quad (1)$$
$$E_i = e_0^j, e_1^j, e_2^j, ..., e_{n-1}^j$$

$AvgRANK(E_i)$와 $AvgRANK(E_j)$를 코드 집합 E_i와 E_j의 평균 순위라고 하자. 이 때 $d_i > d_j$이면 $AvgRANK(E_i) > AvgRANK(E_j)$ 관계가 성립한다.

4. 가변 밀도 스크램블링: 클래스 분할 기법

이 절에서는 비균일 확률 밀도 함수를 사용하지 않고 가변 밀도 스크램블링 기법을 구현하는 클래스 분할 기법에 대해 설명한다. 클래스 분할 기법은 플레인 코드와 사이퍼 코드를 순위에 따라 몇 개의 클래스로 나누고 플레인 코드에 대응되는 클래스에 있는 사이퍼 코드 중 하나로 변환하는 것이다.

Fig. 7을 보면 ECC와 DCC에 256개의 플레인 코드와 사이퍼 코드가 존재한다. 이를 16개의 클래스로 분할하면, 각 클래스에는 16개의 코드가 존재한다. 플레인 코드 '255'의 인코딩은 다음과 같이 이루어진다. ET로 부터 플레인 코드 '255'의 순위가 1이며, Fig. 7의 ECC에서 보듯이 이는 클래스 0에 속함을 알 수 있다. 그러면 클래스 분할 기법은 플레인 코드 '255'를 클래스 '0'에 있는 16개의 사이퍼 코드 중 하나로 변환한다. 구체적으로 0-15 값을 반환하는 shift16A(addr)이 제공하는 랜덤넘버를 이용하며 순위를 조정한다. Fig. 7의 예

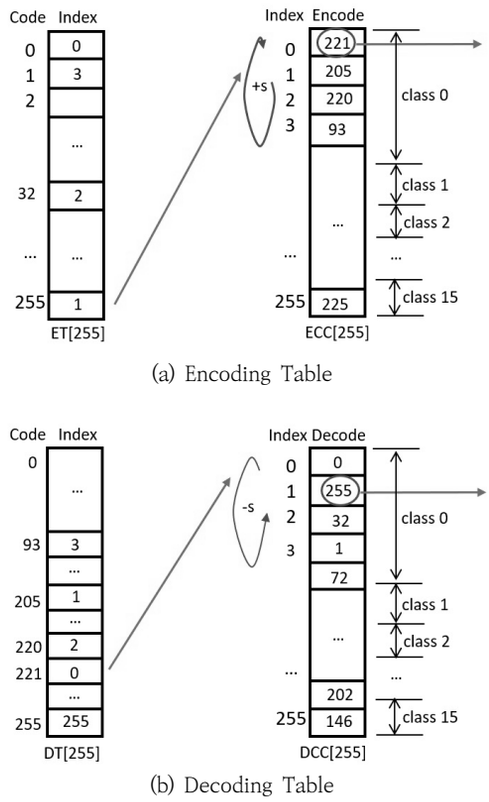


Fig. 7. Tables and Usage Examples of Class Based Translation

에서는 shift16A()가 15를 리턴하며, 순위 1에 리턴 값 15를 더한 후 클래스 내부에서 환형으로 순회하여 새로운 순위 0에 있는 사이퍼 코드 '221'를 선택한다.

디코딩 과정은 다음과 같다. 먼저 DT로부터 사이퍼 코드 '221'의 순위 0을 알아낸다. 순위 0은 클래스 '0'에 속하며, 이제 shift16A(addr)이 제공하는 랜덤넘버를 이 순위에서 빼서 새로운 순위를 결정한다. Fig. 7의 예에서는 shift16A(addr)가 15를 리턴하며, 이를 순위 0에서 빼고 환형으로 순회하여 새로운 순위 1에 있는 플레인 코드 '255'를 선택한다.

위에서 설명한 것과 같이 클래스 분할 기법은 플레인 코드를 특정 클래스에 속한 사이퍼 코드 중 하나로 변환하는데, 이 경우 스크램블링 정도가 약하게 된다. 스크램블링 정도를 높이기 위해서 특정 확률로 클래스 자체를 변경할 수 있다. 클래스 변경을 위해 Fig. 8에서와 같이 prob(addr) 함수와 shift16B(addr) 함수가 사용된다. prob(addr)은 특정 확률로 TRUE를, 나머지 확률로 FALSE를 반환하며, shift16B(addr)은 0에서 15 사이의 값을 반환한다. 만약 prob(addr)이 TRUE를 리턴하면 shift16B(addr)이 반환하는 값을 클래스 번호에 더한 후 환형으로 순회하여 새로운 클래스를 선택한다. 이와 같이 일정한 확률로 클래스번호를 조정하여 스크램블링 정도를 조절할 수 있으며, prob(addr)이 100% TRUE를 반환하는 경우 클래스 분할 기법은 순수한 랜덤화 기법이 된다. 위 예에서 shift16A(addr), shift16B(addr), 그리고 prob(addr)은 동일한 주소 addr에 대해 항상 동일한 값을 반환한다.

```
int encode(addr, '255') {
    index = ET[255];
    cl = index /16;
    offset = index % 16;
    s = shift16A(addr);
    if ((prob(addr))
        cl = rotate(16, cl + shift16B(addr));
    /* return 221 in this example*/
    return ECC[cl *16+ rotate(16, offset+s)];
}
int decode(addr, '221') {
    index = DT[221];
    cl = index /16;
    offset = index % 16;
    s = shift16A(addr);
    if ((prob(addr))
        cl = rotate(16, cl - shift16B(addr));
    /* return 255 in this example*/
    return DCC[cl *16+ rotate(16, offset-s)];
}
```

Fig. 8. Implementation Example of Encode/decode Functions for Class Based Translation

5. 실험 결과

5.1 코드 출현 빈도 분석

3장에서 설명한 대로 가변 밀도 스크램블링이 가능하려면 플레인 코드가 비균일 분포를 가지고 있어야 한다. 이를 확인하기 위해 Table 1과 같이 실제 사용되는 컴퓨터 시스템에서 데이터 코드의 출현 빈도를 조사하였다. 윈도우 시스템의 시스템 폴더인 C:\Windows, C:\Program Files, 그리고 C:\Program File (x86)의 경우 데이터 코드의 출현 빈도 차이가 매우 크다. 그리고 대부분의 윈도우 시스템에서 거의 동일한 출현 빈도를 보여준다. 이러한 현상은 Linux 시스템의 바이너리나 라이브러리 폴더에서도 비슷하게 나타난다.

사용자 데이터까지 포함한 결과는 컴퓨터마다 차이를 보여주지만 여전히 특정 코드는 빈번히 출현하고 어떤 코드는 그 발생 빈도가 매우 낮다. Fig. 9는 사용자 데이터까지 포함하도록 윈도우 시스템이 설치된 노트북 컴퓨터의 C:\ 드라이브와 리눅스 시스템이 설치된 서버 컴퓨터의 루트 디렉터리에서 측정한 데이터 코드 출현 빈도를 보여준다. 두 시스템에서 공통적으로 0xFF와 0x00 값을 가지는 데이터 코드가 상당히 많이 발생한다. 출현 빈도가 가장 높은 0x00 코드를 제외하고 확대한 코드 분포를 보면 각 데이터 코드마다 출현 빈도에 상당한 편차가 있음을 확인할 수 있다. 이러한 비균일 코드 분포는 본 논문에서 제안한 가변 밀도 스크램블링 기법이 데이터 코드 출현 빈도의 차이에 비례하여 코드 밀도 차이를 만들 수 있다는 것을 의미한다.

5.2 가변 밀도 스크램블링 기법의 실험 결과

본 논문에서 제안한 가변 밀도 스크램블링 기법을 평가하기 위해 1:1 변환 기법, 회전판 기법, 그리고 클래스 분할 기법을 구현하였으며, Table 2는 각 기법의 이름과 특징을 보여준다. Raw는 데이터 코드들을 변환하지 않고 플레인 코드 그대로 플래시 메모리에 기록한다. M0는 1:1 변환 기법을 구현한 것이다. M1은 회전판 기법을 구현한 것으로 shift()함수에서 정규분포를 사용한다. M2는 클래스 분할 기법으로 10%의 확률로 클래스를 변환한다. M0-M2의 세 가지 기법 모두 데이터 코드 출현 빈도에 따라 정렬된 FT 테이블과 셀 상태가 중앙에 가까운 순으로 정렬된 PDT 테이블을 이용하여 생성된 인코딩과 디코딩 테이블을 사용하였다. MLC의 경우 PDT 테이블에서 제일 높은 순위의 코드는 상태 S2를 가지며, QLC에서는 상태 S7을 가진다.

Table 1. Computer Systems for Experiments

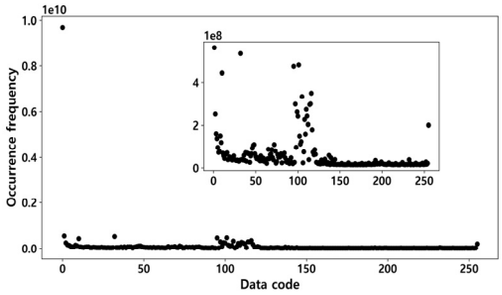
Type	Operating System	Storage Device Usage
Notebook I	Win10 pro(v.2009)	140/256 GB
Notebook II	Win10 pro(v.2004)	55/128 GB
Notebook III	Win10 e(v.20H2)	228/512 GB
Desktop	Win10 e(v.1909)	58/128 GB
Server I	Ubuntu 18.04 (ker 5.3.7)	35/512 GB
Server II	Ubuntu 20.04 (ker 5.4)	78/256 GB
Server III	Ubuntu 18.04 (ker 4.19.154)	168/256 GB

Table 2. Schemes Implemented in Simulator

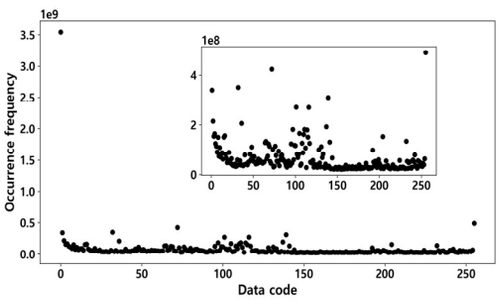
Label	Scrambling Method	Note
Raw	N/A	
M0	1:1	
M1	Wheel	Normal Distribution (SD: 5)
M2	Class division	10% probability

실험은 Table 1과 같이 노트북, 테스크톱, 그리고 서버에서 수행되었으며, 이들은 거의 비슷한 실험 결과를 보였다. Fig. 10은 실험 결과 중 Notebook I에서 수행된 실험 결과이다. MLC와 QLC 두 가지의 플래시 메모리 타입에서 Raw의 결과는 특정 셀의 상태가 두드러지게 높게 나오는 형태를 보여주고 있다. 특히 MLC의 경우 S2, QLC의 경우 S10의 빈도가 높은데, 그 이유는 가장 빈번히 출현하는 플레인 코드 0x00의 셀 상태가 S2와 S10이기 때문이다. 1:1 변환기법을 사용하는 경우 빈번히 출현하는 플레인 코드에 대응되는 사이퍼 코드들은 MLC의 경우 셀 상태 S2와 S3, 그리고 QLC의 경우 S7과 S8 상태를 가진다. 그 결과 Fig. 10에서 보듯이 이 두 상태 비율이 높게 나온다.

회전판 기법(M1)과 클래스 분할 기법(M2)은 확률적으로 스크램블링 하면서 가운데 셀 상태를 가지는 코드를 많이 생성하여 비슷한 결과를 보인다. MLC의 경우 4개의 셀 상태 중 두 개가 중앙에 가까운 상태이기 때문에 Fig. 10(a)에서 보듯이 확률적 스크램블링 기법이나 1:1 변환기법의 차이가 크지 않다. 그러나 QLC 결과인 Fig. 10(b)에서 이들은 차이를 보이는데, 확률적 스크램블링 기법의 경우 셀 상태는 1:1 변환

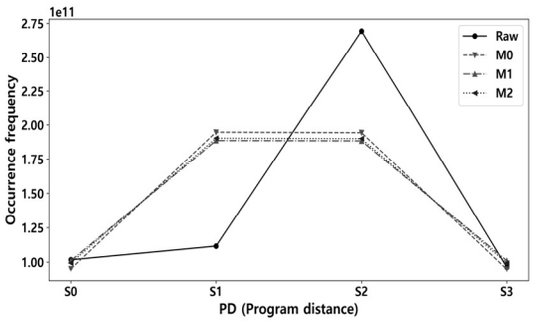


(a) Notebook I

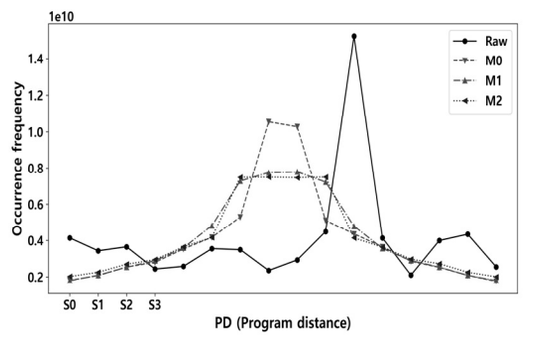


(b) Server I

Fig. 9. Occurrence Frequency of Data Code



(a) MLC



(b) QLC

Fig. 10. Experimental Results of Scrambling Schemes

기법보다는 더 넓게 퍼져 있으면서 중앙에 가까운 셀 상태들의 빈도가 높다. 이러한 실험 결과는 가변 밀도 스크램블링 기법이 우리가 목표한 대로 스크램블링을 수행하면서도 더 많은 셀들의 상태를 중앙에 가깝게 만들 수 있음을 보여준다.

6. 결 론

랜덤한 코드를 생성하는 기존 스크램블링 기법과 달리 본 논문에서는 변환된 코드의 밀도를 다르게 만드는 가변 밀도 스크램블링 기법을 제안하였으며, 이 기법을 플래시 메모리에 적용하여 셀 상태를 중앙에 위치시키는 코드의 밀도를 높게 만들었다. 실제 시스템에 저장된 데이터로 제안한 기법의 효과를 측정하였으며, 실험 결과는 가변 밀도 스크램블링 기법이 목표 상태에 근접한 셀의 밀도를 높일 수 있음을 보여준다. 이와 같이 더 많은 셀 상태를 중앙에 가깝게 만들면 셀 간 간섭을 줄여 에러율을 개선할 수 있을 것으로 기대되며, 향후 연구에서 셀 상태에 따른 에러율을 측정할 예정이다. 아울러 데이터 패턴에 따라 동적으로 인코딩과 디코딩 테이블을 재구성하는 동적 기법도 연구할 계획이다. 또한 효율적인 하드웨어 구현도 향후 연구 주제이다.

References

[1] K. Lee, M. Kang, S. Seo, D. H. Li, J. Kim, and H. Shin, “Analysis of failure mechanisms and extraction of Activation Energies (Ea) in 21-nm nand flash cells,” *IEEE Electron Device Letters*, Vol.34, No.1, pp.48-50, 2013.

[2] K. Lee, D. Kang, H. Shin, S. Kwon, S. Kim, and Y. Hwang, “Analysis of failure mechanisms in erased state of sub 20-nm nand flash memory,” in *2014 44th European Solid State Device Research Conference (ESSDERC)*, pp.58-61, 2014.

[3] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, “Characterizing, exploiting, and mitigating vulnerabilities in MLC NAND flash memory programming,” 2018.

[4] Y. Cai, S. Ghose, Y. Luo, K. Mai, O. Mutlu, and E. F. Haratsch, “Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 49-60, 2017.

[5] J. Cha and S. Kang, “Data randomization scheme for endurance enhancement and interference mitigation of multilevel flash memory devices,” *ETRI Journal*, Vol.35, pp.166-169, 2013.

[6] C. Kim, et al., “A 21nm high performance 64Gb MLC NAND flash memory with 400MB/s asynchronous toggle DDR interface,” in *2011 Symposium on VLSI Circuits - Digest of Technical Papers*, pp.196-197, 2011.

[7] N. Sommer, M. Anholt, O. Golov, U. Perlmutter, S. Winter, and G. Semo, “Data scrambling schemes for memory devices,” US Patent (US 8,261,159 B1), 2012.

[8] S. Tanakamaru, C. Hung, and K. Takeuchi, "Highly reliable and low power SSD using asymmetric coding and stripe bitline-pattern elimination programming," *IEEE Journal of Solid-State Circuits*, Vol.47, No.1, pp.85-96, 2012.
[9] Q. Xu, T. M. Chen, Y. Hu, and P. Gong, "Write pattern format algorithm for reliable NAND-Based SSDs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, Vol.61, No.7, pp.516-520, 2014.
[10] W. Zhang, Q. Cao, and Z. Lu, "Bit-Flipping schemes upon MLC flash: Investigation, implementation, and evaluation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.38, No.4, pp.780-784, 2019.
[11] P. Cappelletti, R. Bez, D. Cantarelli, and L. Fratin, "Failure mechanisms of flash cell in program/erase cycling," in *Proceedings of 1994 IEEE International Electron Devices Meeting*, pp.291-294, 1994.
[12] C. Lee, et al., "A 32-Gb MLC NAND flash memory with vth endurance enhancing schemes in 32 nm CMOS," *IEEE Journal of Solid-State Circuits*, Vol.46, No.1, pp.97-106, 2011.
[13] D. Wei, L. Deng, P. Zhang, L. Qiao, and X. Peng, "NRC: A nibble remapping coding strategy for NAND flash reliability extension," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.35, No.11, pp.1942-1946, 2016.
[14] J. Guo, Z. Chen, D. Wang, Z. Shao, and Y. Chen, "DPA: A data pattern aware error prevention technique for NAND flash lifetime extension," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 592-597, 2014.
[15] W. Lee, M. Kang, S. Hong, and S. Kim, "Interpage-based endurance-enhancing lower state encoding for MLC and TLC flash memory storages," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol.27, No.9, pp.2033-2045, 2019.
[16] N. Mielke, et al., "Bit error rate in NAND flash memories," *IEEE International Reliability Physics Symposium*, pp.9-17, 2008.

현 철 승
https://orcid.org/0000-0001-5157-8652
e-mail : cshyun@uos.ac.kr
2007년 서울시립대학교 컴퓨터통계학과(석사)
2012년 서울시립대학교 컴퓨터과학부(박사)
2012년 ~ 2014년 ㈜티엘아이 책임연구원
2017년 ~ 현 재 서울시립대학교 컴퓨터과학부 연구교수
관심분야 : 고성능 스토리지 시스템, 시스템 소프트웨어

정 관 일
https://orcid.org/0000-0001-7756-6457
e-mail : gijeong@uos.ac.kr
2020년 동서울대학교 컴퓨터소프트웨어학과 (학사)
2020년 ~ 현 재 서울시립대학교 컴퓨터과학부 석사과정
관심분야 : 내장형시스템, 시스템 소프트웨어

유 수 원
https://orcid.org/0000-0003-1292-8298
e-mail : ysw1508@uos.ac.kr
2020년 연세대학교 컴퓨터정보통신공학부 (학사)
2020년 ~ 현 재 서울시립대학교 컴퓨터과학부 석사과정
관심분야 : 운영체제, 스토리지 시스템

이 동 희
https://orcid.org/0000-0003-2240-7939
e-mail : dhl_express@uos.ac.kr
1991년 서울대학교 컴퓨터공학과(석사)
1998년 서울대학교 컴퓨터공학과(박사)
1998년 ~ 1999년 삼성전자 중앙연구소 선임연구원
1999년 ~ 2001년 제주대학교 통신컴퓨터과학부 조교수
2002년 ~ 현 재 서울시립대학교 컴퓨터과학부 교수
관심분야 : 운영체제, 내장형시스템, 스토리지 시스템