논문 2020-15-02

SSD FTL의 캐시 메커니즘에 대한 심층 분석 및 개선

(An In-Depth Analysis and Improvement on Cache Mechanisms of SSD FTL)

이 형 봉, 정 태 윤*
(Hyung-Bong Lee, Tae-Yun Chung)

Abstract : Recently, the capacity of SSD has been increasing rapidly due to the improvement of flash memory density. To take full advantage of these SSDs, first of all, FTL's prompt adaptation is necessary. The FTL is a translation layer existing in SSDs to overcome the drawback of the SSD that cannot be modified in place, and has garbage collection and caching functions in addition to the map table management function. In this study, we focus on caching function, compare and analyze the cache implementation methodologies, and propose improved methods. Typical cache implementations divide the cache into groups, manage and retrieve the caches in the group as a linked list. Thus, searches are made in the order of the linked list. In contrast, we propose a method of sequential searching using the search area group of a cache registered in the map table regardless of the linked list and cache group. Experimental results show that the proposed method has a 2.5 times improvement over the conventional method.

Keywords: SSD, FTL(Flash Translation Layer), Cache group, Linked list, B tree

Ⅰ. 서 론

플래시 메모리에 기반한 SSD (Solid State Disk)는 널리 알려진 바와 같이 빠른 접근속도, 낮은 전력소모, 높은 내충격성, 가벼운 무게 등의 장점으로 기존의 하드 디스크를 빠르게 대체하고 있다. 이 중에서도 특히 빠른 접근속도는 일반 사무용 PC의 부팅과정에서 쉽게 실감할 수 있다. 기존의 하드 디스크가 데이터의 흩어짐 정도에 따라 기계적 탐색시간이 크게 늘어나는 반면, SSD는 저장 위치가 전자적으로 탐색되므로 시간적 성능 저하가 전혀 없다. 이에 따라 최근 출시되는 대부분의 PC는 시스템 디스크용으로 SSD를 탑재하고 있다 [1].

이러한 장점에도 불구하고 초기화 없이 제자리 쓰기가 불가능하고, 초기화 회수가 한정되어 있다는 단점은 SSD의 효율적 이용에 대한 장애요인이다. 여기에 더해 읽기·쓰기 단위인 페이지 크기와 초기화 단위인 블록의 크기가 다르고, 각 연산에 소모되는 전력이 각각 다르다는 점은 SSD의 보편적 최적화는 거의 불가능함을 의미한다 [2]. 따라서 이와 같은 문제들에 대한 해결방안은 SSD 사용 환경에 따라 최적화되어야 한다 [3]. 이런 문제점들의 해결을 지원하기 위해 모든 SSD는 내부에 64~256MB 정도의 RAM과 함께 컴퓨팅 환경을 제공하고 있는데 여기서 이루어지는 다양한 최적화 알고리즘들이 FTL (Flash Translation Layer)이다.

이 연구에서는 FTL의 핵심 기능 중의 하나인 디스크 캐시 메커니즘들을 심층 분석하고, 성능을 향상하기 위한 방안을 제시한다. 이를 위해 2장에서 FTL 관련 연구를 살펴보고, 3장에서 기존 캐시 메커니즘들을 비교·분석한다. 4장에서는 캐시 메커니즘의 성능 향상 방안을 제시하며 마지막으로 5장에서 결론으로 이 논문을 맺는다.

*Corresponding Author (tychung@gwnu.ac.kr)
Received: Nov. 5, 2019, Revised: Jan. 13, 2020, Accepted: Jan. 20, 2020.
H.B. Lee, T.Y. Chung: Gangneung-Wonju National University(Prof.)
※ 이 논문은 2020년도 정부(산업통상자원부)의 재원으로 한국산업기술진흥원 지원을 받아 수행된 연구임(R0006229, 차세대 생명・건강산업생태계 조성사업).

ⓒ IEMEK J. Embed. Sys. Appl. 2020 Feb. 15(1) 9-16
ISSN : 1975-5066
http://dx.doi.org/10.14372/IEMEK.2020.15.1.9

Ⅱ. FTL 관련 연구

1. FTL 주요 연구 분야

SSD는 이전에 할당된 페이지에 대한 수정이 필요하면 반드시 해당 페이지를 초기화한 후 쓰기가 가능하다. 그런데 초기화는 페이지 단위가 아닌 여러 개의 페이지로 구성된 블록 단위로만 가능하여 주변의 다른 페이지들까지 지우게 되므로 즉시 초기화가 사실상 불가능하다. 따라서 수정된 페이지 내용은 다른 빈 페이지에 저장하고, 이에 따른 매핑 정보를 수정해주어야 하는데 이것이 FTL의 가장 기본적인 기능에 해당된다. 이를 위한 매핑 테이블의 크기는 SSD 용량의 크기에 비례한다. 페이지 크기가 8KB이고 하나의 블록이 256페이지로 구성되며, 총 128x1024개의 블록으로 이루어져 전체 용량이 256GB인 SSD [4]의 경우 총 32M개의 페이지가 존재한다. 4바이트 정수로 단순 매핑할 경우 128MB가 필요한데, 이는 SSD 자체에서 제공하는 RAM 크기의 제한을 고려하면 적지 않은 양이다. 이 범주의 문제는 맵 테이블의 압축이나 캐싱 등의 방법으로 연구가 이루어진다 [3, 5].

페이지를 재할당하고 맵 테이블을 수정하는 순간 기존에 할당되었던 페이지는 가비지가 된다. 따라서 FTL은 가비지를 수집하고 초기화하여 빈 페이지 풀에 등록해주어야 한다. 이 때 모든 페이지가 가비지인 블록은 사실상 존재하지 않기 때문에 가비지가 가장 많이 포함된 희생 블록을 찾되 마모 빈도를 고려해야 하는 등의 전략이 필요하다. 연구 [6]은 각 블록에 대한 초기화 작업이 이루어지는 기간을 합산하여 그 누적 기간이 긴 블록을 먼저 선택함으로써 전체적인 마모 평준화를 추구하고, [7]은 운영체제 수준에서 FTL의 가비지 수집 부하 정도를 반응 시간으로부터 학습하고, 부하가 심할 경우 파일 시스템의 불필요한 부분을 먼저 제거(TRIM) 요청함으로써 부하를 완화시킨다. 연구 [8]은 선택된 희생 블록에 포함된 유효 페이지들을 즉시 복사하지 않고 일정 기간 동안 캐시로 보관함으로써 전체적인 성능향상을 추구한다.

2. FTL 캐시 연구

매핑과 가비지 수집 기능 외에 FTL의 또 다른 주요 기능으로 페이지 캐싱을 들 수 있다. 이 분야는 대부분 널리 알려진 운영체제 캐시 전략을 도입 적용하기 때문에 이와 관련된 연구는 의외로 많지 않다. 일반적인 운영체제 디스크 버퍼 캐시는 어플리케이션의 입·출력 크기 및 위치가 디스크 페이지 크기와 경계를 만족하지 못할 때 데이터를 임시 보관하는 기능과 디스크의 저장 내용을 RAM에 복사하여 성능을 개선하는 등 두 가지 기능을 동시에 가진다 [9]. 반면에 FTL 캐시는 SSD 물리 페이지의 복사본으로서 성능 향상이 주목적이다. 그 외에 양쪽 캐시 사이의 아주 중요한 차이점은 아래와 같다.

■ 지역성 활용 관점

운영체제 디스크 캐시는 어플리케이션의 디스크 접근 지역성을 충분히 반영해야 하지만, 운영체제 캐시 아래에서 활동하는 FTL 캐시는 지역성에 의한 재접근율이 높지 않다. 즉, FTL 캐시는 최초 읽을 때 한 번 참조되고 마지막 쓰기가 이루어질 때 한 번 참조되는 경우가 많다. 따라서 운영체제의 캐시 교체는 참조 카운트와 청결여부 (dirty)를 고려하는 NUR (Not Used Recently)이 일반적이지만, FTL 캐시는 LRU (Least Recently Used) 알고리즘이 적합하다.

■ 쓰기 팽창 관점

운영체제 디스크 캐시에서는 교체를 위한 희생 캐시 선택의 영향이 미치는 범위가 제한적이다. 그러나 FTL의 캐시에서는 희생 캐시의 선택이 페이지 저장 패턴에 영향을 미치고, 그 결과는 다시 가비지 페이지들의 분포에 영향을 미친다. 즉, 가비지 페이지들이 여기저기 흩어져 존재하는 경우 보다는 동일한 블록에 집중적으로 포함되는 정도가 높을수록 유효 페이지에 대한 복사·쓰기 빈도는 줄어든다 [10].

이러한 차이점에도 불구하고 청결 캐시 우선 (CFLRU: Clear First LRU) [11], 정적 청결 캐시 우선 (CCFLRU: Cold Clean First LRU) [12] 등 LRU에 NUR 알고리즘을 부가적으로 적용하여 제한적 환경에서 페이지 쓰기 회수를 줄이려는 연구들이 있다. 이 연구에서는 LRU 페이지 교체 알고리즘을 기반으로 몇 가지 캐시 구현 방법론을 비교·분석하고 개선안을 제시한다.

Ⅲ. 일반적인 캐시 구현 방법 비교・분석

1. 실험환경

■ FTL 시뮬레이터

DiskSim [13]을 축약하여 [10]에서 제안한 바와 같이 운영체제 인터페이스 에뮬레이터 후단에

표 1. 실험 플랫폼
Table 1. Experimental Platform

Items	Specification or model
CPU	Intel i5-4590, 3.3GHz
Memory	4GB
OS	Cygwin 2.1 in Window 7
Language	GCC and POSIX thread

FTL을 배치한 모형을 적용하고, SSD RAM 256M 중 128MB는 맵 테이블로, 나머지 128MB는 캐시로 사용한다.

■ 실험 플랫폼

실험은 표 1과 같이 윈도우 내 리눅스 개발 환경인 Cygwin [14]에서 진행한다.

■ 워크로드

FTL에 적용할 부하 워크로드로, 200MB 크기의 각 파일에 대하여 랜덤하게 설정한 8KB 안팎의 레코드를 15,000번 순차 쓰기 하는 스레드 4개, 순차 읽기 하는 스레드 4개, 랜덤 위치 (offset)에서 읽기 하는 스레드 4개, 랜덤 위치에서 쓰기 하는 스레드 4개, 랜덤 위치에서 읽고 덮어쓰는 스레드 4개 등 대표적인 입·출력 패턴 다섯 가지 유형을 포괄하는 총 20개의 스레드를 사용한다.

■ LRU 캐시 교체 알고리즘

이 연구에서 구현한 LRU 캐시 교체 알고리즘은 그림 1과 같이 최초 쓰기의 경우 일단 데이터를 캐시에만 보관할 뿐 곧바로 물리 페이지를 할당하지 않고, 이후에 교체 대상으로 선정되었을 때 비로소 물리 페이지에 플러시한다. 따라서, 최선의 경우 물리 페이지는 할당되지 않은 채 최종적으로 삭제와 함께 마무리될 수도 있다.

■ 가비지 수집 알고리즘

가비지 수집은 플래시의 빈 페이지 개수가 하한선인 256개 이하이면 시작하여 상한선인 512개에 도달하면 멈추도록 한다. 가비지 수집 대상 블록 선정에는 인위적 우선순위를 두지 않고 랜덤 하게 선택된 블록에 대하여 가비지를 수집한다.

2. 해싱 그룹에 의한 LRU 캐시 구조 및 운영

일반적인 캐시 구현은 그림 2와 같이 캐시들을 그룹으로 분할하고 논리 페이지 번호 (lpn)에 대한 해시 값으로 할당 그룹을 결정한다. 이 연구에서는 'lpn mod NGRP' 형태의 해시 함수를 적용한다. 희생 캐시는 CLRU (Cache Least Recently Used)에서 할당되고 읽기나 쓰기가 이루어진 캐시는 CMRU (Cache Most Recently Used)로 이동한다.

```
ftl_read_LRU (lpn, buf)
{
  cgr = hash(lpn);
  cache_p = cache_search_XX(lpn, cgr);
  if (cache_p == NULL)
    cache_p = get_victim_LRU(cgr);
    ppn = get_ppn_MAP(lpn);
    flash_read(cache_p, ppn);
  }
  data_copy(buf, cache_p);
  update_list_LRU(cache_p);
}

ftl_write_LRU (lpn, buf)
{
  ppn = get_ppn_MAP(lpn);
  cgr = hash(lpn);
  if (ppn == -1) { // first write for lpn
     cache_p = get_victim_LRU(cgr);
  }
  else {              // ppn is valid or NULL
     cache_p = cache_search_XX(lpn, cgr);
     if (cache_p == NULL)
       cache_p = get_victim_LRU(cgr);
  }
  data_copy(cache_p, buf);
  update_list_LRU(cache_p);
  update_MAP(lpn, NULL);
}
```

그림 1. FTL 읽기/쓰기 프로시저
Fig. 1 FTL read/write Procedure

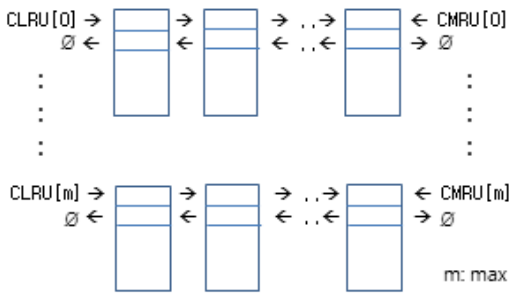

그림 2. 해싱 그룹에 의한 캐시 구조
Fig. 2 Cache Architecture using Hashing Group

이 논문에서는 128MB RAM에 8KB 캐시 16,384개를 둔다.

그림 1의 캐시 검색 프로시저 *cache_seacrh_XX()*에는 CLRU에서 CMRU 방향 (LM으로 명명)으로 검색하거나, 그 반대 방향 (ML로 명명)으로 검색하는 방안을 적용할 수 있다. 그림 3에 리스트 링크를

```
typede struct fcache {
    int              lpn;
    struct fcache *plink, *nlink;
} FCACHE;
FCACHE *cache_search_LM(lpn, cgr)
{
   cache_p = CLRU[cgr];
   while (cache_p != NULL) {
      if (cache_p→lpn == lpn)
         return cache_p;
      cache_p = cachep→nlink;
   }
   return NULL;
}
FCACHE *cache_search_ML(lpn, cgr)
{
   cache_p = CMRU[cgr];
   while (cache_p != NULL) {
      if (cache_p→lpn == lpn)
         return cache_p;
     cache_p = cache_p→plink;
   }
   return NULL;
}
```

그림 3. LM/ML 캐시 검색 알고리즘
Fig. 3 LM/ML Cache Search Algorithm

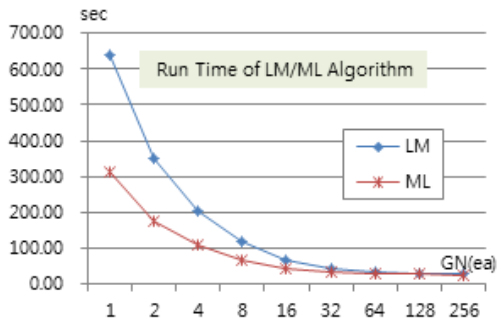

그림 4. LM/ML 알고리즘 비교
Fig. 4 Comparison of LM/ML Algorithm

활용하는 두 가지 알고리즘을 보였다. 표 2에 이 연구의 실험 환경에서 실시한 두 알고리즘 LM과 ML에 대하여 모든 워크로드가 종료할 때까지의 종료 시간 측정 결과를 보였고, 그림 4에 그래프로 그 추이를 보였다. 표 2와 그림 4에서 GN은 캐시 그룹 개수를, AL은 알고리즘의 유형을 각각 의미한다. 또한 여기서 실행 시간은 실제 디스크 입·출력이나 버퍼 복사가 없는 상황이므로 거의 대부분 캐시 운영 시간이라 보아도 무리가 없다.

표 2. LM/ML 알고리즘 검색시간 (초)
Table 2. Search Time of LM/ML Algorithm (sec)

AL \ GN	1	2	4	8	16	32	64	128	256
LM	638.4	352.3	206.7	118.0	65.7	45.6	34.5	30.6	30.0
ML	313.3	175.1	108.4	66.9	42.8	32.7	28.3	27.0	27.0

표 3. ML/SQ 알고리즘 검색시간 (초)
Table 3. Search Time of ML/SQ Algorithm (sec)

AL \ GN	1	2	4	8	16	32	64	128	256
ML	313.3	175.1	108.4	66.9	42.8	32.7	28.3	27.0	27.0
SQ	142.1	83.2	57.0	40.4	31.7	28.9	34.5	27.1	26.1

```
FCACHE *cache_search_SQ(lpn, cgr)
{
   cache_p = FCache+  cgr;
   while (cache_p < FCache_end) {
      if (cache_p→lpn == lpn)
         return cache_p;
      cache_p = cache_p +  NG;
   }
   return NULL;
}
```

그림 5. SQ 캐시 검색 알고리즘
Fig. 5 Search Algorithm for SQ Cache

그림 4는 MRU 캐시를 먼저 검색하는 것이 우수함을 의미하는데 이는 최근 참조된 페이지가 다시 참조될 가능성이 높다는 시간적 지역성 때문인 것으로 분석되고 참조 빈도가 왕성한 이 연구의 실험 환경과 일치한다.

Ⅳ. FTL을 위한 캐시 개선 방안

1. 캐시 영역의 순차 검색

그림 3의 일반적인 캐시 검색은 연결 리스트에 따라 LRU 혹은 MRU 순서에 의해서만 이루어진다. 이와는 달리 그림 5와 같이 캐시 그룹의 해당 영역을 스캔하며 순차 검색 (SQ로 명명)하는 방안을 시도해볼 수 있다. 표 3에 ML과 SQ 알고리즘의 검색 시간을 보였고 그림 6에 이들 두 알고리즘의 실행 시간을 그래프로 비교해 보였다.

이 그림으로부터 SQ 알고리즘의 개선 효과가 의외로 매우 크다는 사실을 알 수 있다. SQ 알고리즘은 LM이나 ML 알고리즘과 달리 LRU·MRU 리스트를 랜덤 순서로 방문하는 결과인데, 이는 리스트

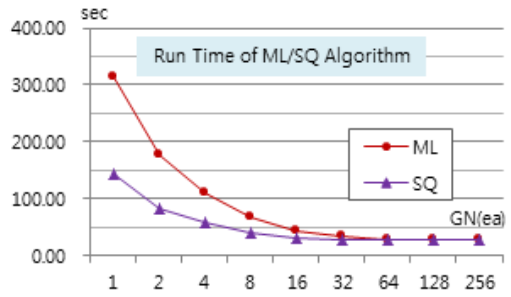

그림 6. ML/SQ 알고리즘 비교
Fig. 6 Comparison of ML/SQ Algorithm

```
get_search_group(cache_p)
{
  sgr = (cache_p - FCache) /
            n_of_caches_per_search_group;
  return (sgr + 1);
}

FCACHE *cache_search_SG(lpn, sgr)
{
  cache_p = FCache+ (sgr -1);
  cache_end_p = cache_p +
            n_of_caches_per_search_group;
  while (cache_p < cache_end_p) {
     if (cache_p→lpn == lpn)
        return cache_p;
     cache_p = cache_p + 1;
  }
}
```

그림 7. SG 캐시 검색 알고리즘
Fig. 7 SG Cache Search Algorithm

조작 연산의 부담이 적을 뿐 아니라 MRU 우선 방문이 적합하지 않은 경우도 존재한다는 점을 시사하는 것으로 분석된다.

2. 맵 테이블과 캐시 구간의 연결

FTL에서 캐시 성능 향상을 위한 가장 이상적인 방법은 맵 테이블 엔트리에 할당된 캐시 위치를 직접 등록하는 것이겠지만 메모리 제한으로 불가능하다. 다른 방법으로 맵 테이블의 물리 페이지 번호의 남는 비트 영역에 캐시 그룹을 표시하는 방안이 가능하다. 하나의 블록이 256 페이지로 구성되고 블록 수가 128M개인 256GB SSD의 경우 총 32M개의 페이지가 존재하는데, 32M는 범주는 25비트면 표현이 가능하다. 이 연구에서는 맵 테이블 엔트리로 32비트 정수를 사용하고, 이중 상위 7 비트 (SG라 명명함)를 아래와 같이 캐시 검색 정보용으로 설정한다.

표 4. SQ/SG 알고리즘 검색시간 (초)
Table 4. Search Time of SQ/SG Algorithm (sec)

GN / AL	1	2	4	8	16	32	64	128 (127)
SQ	142.1	83.2	57.0	40.4	31.7	28.9	34.5	27.1
SG	59.2	51.4	32.7	28.5	27.2	26.7	26.0	27.4

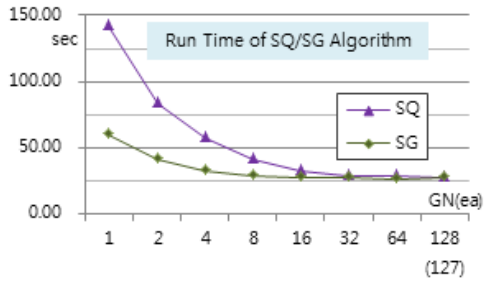

그림 8. SQ/SG 알고리즘 비교
Fig. 8 Comparison of SQ/SG Algorithm

- SG가 0이면 캐시 부재를 의미
- SG가 1~127이면 캐시가 존재하고 할당 캐시가 SG-1 검색 그룹에 위치함을 의미

위 방법을 적용하면 캐시를 최대 127개 그룹까지 표시할 수 있고, 캐시가 존재하지 않는 경우 그림 1의 *cache_search_XX()* 프로시저를 건너뛸 수 있다. 이 방법의 또 다른 장점은 그림 2의 물리적인 캐시 그룹과 관계없이 캐시들의 검색 위치를 그룹핑하여 그림 5의 순차 검색을 적용할 수 있다는 점이다. 이를테면 물리적으로는 하나의 캐시 그룹을 운영하지만 캐시의 검색 구간은 127개로 나누어 등록할 수 있다. 그림 7에 물리적 캐시 그룹과 관계없이 검색 그룹을 독립하여 설정하고 검색하는 알고리즘을 보였다.

표 4에 SQ와 SG 알고리즘의 검색시간을, 그림 8에 이들 두 알고리즘의 실행 시간 비교 그래프를 각각 보였는데 SG의 개선효과가 매우 크다는 사실을 확인할 수 있다. 여기서 SG 알고리즘에는 하나의 물리적 캐시 그룹을 설정했고 그룹 수 GN은 검색 구간 개수를 의미한다.

3. B tree의 적용

캐시 검색 성능을 비교하기 위해 그림 9와 같이 LRU 연결 리스트 위에 논리 페이지 번호 (lpn)을 키로 하는 Btree 배치를 고려 (BT라 명명함)할 수 있다. B tree를 도입하는 경우 검색 과정에서 캐시

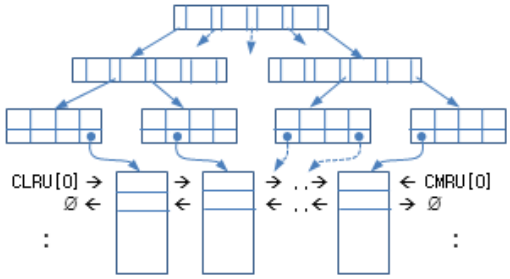

그림 9. ML/SQ 알고리즘 비교
Fig. 9 Comparison of ML/SQ Algorithm

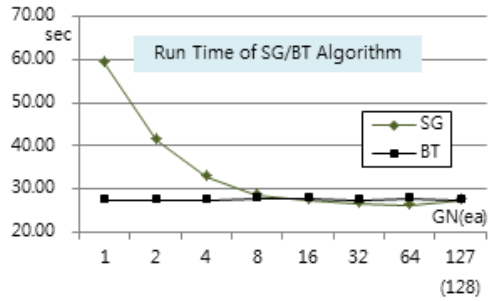

그림 10. SG/BT 알고리즘 비교
Fig. 10 Comparison of SG/BT Algorithm

그룹핑 상태가 전혀 개입되지 않으므로 검색 시간이 일정하다. 이러한 BT 알고리즘의 특징은 B tree의 차원을 5로 설정하고 실시한 표 5, 그림 10에서 확인할 수 있다. B tree의 차원을 6~10으로 변화시키면서 실시한 결과, 검색 시간에서 유의미한 차이는 나타나지 않았고 노드 구성을 위한 메모리 소요량은 5KB 안팎을 유지했다. 즉, SG 알고리즘의 검색 구간을 8개 미만으로 해야 할 경우에는 약간의 메모리 소모량을 감수한다면 BT 알고리즘 선택이 유리하다. SG 알고리즘의 검색 구간은 SSD의 용량 증가에 따라 플래시 페이지 개수가 점차 늘어나면 맵 테이블의 SG 비트 영역이 축소되어 검색 구간도 127개→63개→31개→15개→7개 형태로 감소할 수밖에 없다.

4. 분석 종합

어떠한 추가 자원 없이 개선된 SG 알고리즘을 기준으로 ML 및 SQ 알고리즘의 시간적 성능을 표 6에 요약하였다. 이 표로부터 SG 알고리즘은 다른 알고리즘보다 평균 1.5~2.5배 우수함을 알 수 있다.

표 5. SG/BT 알고리즘 검색시간 (초)
Table 5. Search Time of SG/BT Algorithm (sec)

GN / AL	1	2	4	8	16	32	64	127 (128)
SG	59.2	51.4	32.7	28.5	27.2	26.7	26.0	27.4
BT	27.4	27.44	27.5	27.7	27.7	27.4	27.6	27.5

표 6. ML/SQ/SG 알고리즘 비교 (배)
Table 6. Comparison of ML/SQ/SG Algorithm (times)

GN / AL	1	2	4	8	16	32	64	128 (127)	avg
ML/SG	5.3	4.2	3.3	2.3	1.6	1.2	1.1	1.0	2.5
SQ/SG	2.4	2.0	1.7	1.4	1.2	1.1	1.1	1.0	1.5

V. 결 론

제자리 수정하기가 불가능한 SSD의 근본적 특성을 해결하기 위한 FTL은 논리 페이지와 물리 페이지 번호 사이의 변환, 쓰인 후 초기화를 기다리는 가비지 수집, 물리 페이지 입·출력을 최소화하기 위한 캐싱 등 세 가지 핵심 기능을 담당한다. 이 논문에서는 FTL의 캐싱 기능에 집중하여 캐시 구현 방안들을 분석하고 개선 방안을 제시하였다. 개선 방안 중, LRU 캐시 그룹과 독립적인 검색 구간을 설정하고 할당 캐시가 속한 검색 구간을 맵 테이블에 표시하는 캐시 구간 순차 검색 방법은 LRU 캐시 그룹을 연결 리스트로 검색하는 일반적인 방법보다 평균 1.5~2.5배 우수함을 확인하였다. B tree 검색 방법은 캐시 그룹핑에 관한 어떠한 영향도 받지 않고 안정적인 최고 성능을 보이지만 약 5KB 정도의 추가 메모리를 소모하는 단점이 있다.

이 연구의 결과는 FTL 시뮬레이션 과정에서 캐시를 어떻게 구현해야 할 것인가에 대한 시행착오 예방에 기여할 것이며, 이를 바탕으로 가비지 수집에 효과적인 캐시 교체 전략에 대한 연구가 계속 이어질 것이다.

References

[1] D.H. Gouk, M.R. Kwon, M.S. Jung, "Practical Analysis and Characterization Methods of Moders SSD using Full-System Integrated SimpleSSD," Journal of Communications of the Korean Institute of Information Scientists and Engineers, Vol. 36, No. 6, pp. 37-55, 2018 (in

Korean).

[2] V. Mohan, T. Bunker, L. Grupp, S. Gurumurthi, M.R. Stan, S. Swanson, “Modeling Power Consumption of NAND Flash Memories Using FlashPower,” Journal of IEEE Transactions on Computer-Aided Design of Integrated Circuit and Stsrems, Vol. 32, No. 7, pp. 1031-1044, 2013.

[3] Y.S. Kim, "A Compact Representation of Translation Pages for Flash Translation Layers of Solid State Drives," Journal of the Korea Society of Computer and Information, Vol. 24, No. 2, pp. 1-7, 2019 (in Korean).

[4] Samsung Electronics, "Samsung SSD 850 Pro Data Sheet Rev. 3," 2017 (avaliable on : http://downloadcenter.samsung.com/content/UM/201711/20171115103115156/Samsung_SSD_850_PRO_Data_Sheet_Rev_3.pdf).

[5] H.J. Kim, D.K. Shin, "Increasing-Order Page-Level Mapping FTL for Memory-constraint Flash Storage Systems," Proceedings of Korea Computer Congress, pp. 1181-1183, 2016 (in Korean).

[6] S.H. Kim, Garbage Collection Technique for Balanced Wear-out and Durability Enhancement with Solid State Drive on Storage Systems," Journal of the Korea Society of Computer and Information, Vol. 22, NO 4, pp. 25-32, 2017 (in Korean).

[7] J.K. Park, J.H. Kim, "Flash Storage System, A Method for Reducing Garbage Collection Overhead of SSD Using Machine Learning Algorithms", Proceedings of International Conference on Information and Communication Technology Convergence (ICTC), pp. 775-777, 2017.

[8] N. Shahidi, M.T. Kandemir, “CachedGC: Cache-Assisted Garbage Collection in Modern Solid State Drives,” Conference of IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), pp. 79-85, 2018.

[9] A. Silberschatz, P.B. Galvin, G. Gagne, "Operating Ssystem Concepts-ninth Edition," Wiley Press, pp. 604-607, 2013.

[10] H.B. Lee, T.Y. Chung, "A Comparative Analysis on Page Caching Strategies Affecting Energy Consumption in the NAND Flash Translation Layer," IEMEK J. Embed. Sys. Appl., Vol. 13, No. 3, pp. 109-116, 2018 (in Korean).

[11] S.Y. Park, D.W. Jung, J.U. Kang, J.S. Kim, J.W. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES'06), pp. 234-241, 2006.

[12] Z. Li. P. Jin, X. Su, K. Cui, L. Yue, "CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory," Journal of IEEE Transactions on Consumer Electronics, Vol. 55, No. 3, pp. 1351-1359, 2009.

[13] J.S. Bucy, J. Schindler, S.W. Schlosser, G.R. Ganger, The DiskSim Simulation Environment Version 4.0 Reference Manual, Technical Report CMU-PDL-08-101, 2008.

[14] Cygwin, Avaliable on : https://www.cygwin.com/

Hyung-Bong Lee (이 형 봉)

He received the B.S. and M.S. degrees in Computer Science from Seoul National University, Seoul, Korea, in 1984 and 1986 respectively. He received his Ph.D. degree in Computer Science from Kangwon National University, Chuncheon, Korea, in 2002. From 1986 to 1994, he was a senior engineer in Computer R&D Division of LG Electronics. From 1995 to 1998, he was with DEC(Digital Equipment Corporation) as a UNIX consultant. From 1999 to 2003, he was an Associate Professor at Honam University, Gwangju, Korea. Since 2004, he has been a Professor in the Department of Computer Science & Engineering at Gangneung-Wonju National University, Wonju, Korea. His current research interests include embedded systems, wireless sensor networks, and data mining algorithms.

Email: hblee@gwnu.ac.kr

Tae-Yun Chung (정 태 윤)

He received the B.S., M.S., and Ph.D. degrees in the School of Electrical & Computer Engineering at Yonsei University, Seoul, Korea in 1987, 1989, and 2000 respectively. From 1989 to 1996, he was a Research Engineer of Samsung Advanced Institute of Technology. From 1996 to 2001, he was with Samsung Electronics as a Senior Research Engineer. From 2000 to 2001, he was a vice chair of International DVD-Forum. Since 2001, he has been with the Department of Electronics Engineering at Gangneung -Wonju National University, Gangneung, Korea, and is currently a Professor. Since 2004, he has been with GEMS-CRC(Gangwon Embedded Software Cooperative Research Center, Gangneung, Korea) as the Chef. His major fields are image signal processing, digital video encoding, multimedia, copy protection , and his current research interests are embedded system, sensor network, video encoding.

Email: tychung@gwnu.ac.kr