JDCS
디지털콘텐츠학회논문지
Journal of Digital Contents Society
Vol. 20, No. 12, pp. 2545-2553, Dec. 2019

리눅스 FUSE 기반 경량 SSD 성능 평가 시뮬레이터 구현

이 형 봉[1] · 권 기 현[2*]
[1]강릉원주대학교 컴퓨터공학과
[2]강원대학교 전자정보통신공학부

Implementation of a Light Weight SSD Perpormance Evaluation Simulator based on Linux FUSE

Hyung-Bong Lee[1] · Ki-Hyeon Kwon[2*]

[1]Department of Computer Science & Engineering, Gangneung-Wonju National University, Wonju 25457, Korea

[2]Department of Electronics, Information & Communication Engineering, Kangwon National University, Samcheok 25913, Korea

[요 약]

SSD를 구성하는 플래시 메모리는 쓰기 전 지우기가 필요하고, 읽기·쓰기 단위의 크기와 지우기 단위의 크기가 다르며, 지우기에 따라 수명이 단축되는 특징을 가지고 있다. 이러한 불리한 특징들을 극복하고 SSD의 최대 장점인 빠른 접근 속도의 이점을 살리기 위해서 스트리밍 서비스 등 다양한 관점에서의 FTL 알고리즘 최적화 실험이 시뮬레이션을 통해 활발하게 이루어지고 있다. 이 연구에서는 기존의 무겁고 복잡한 DiskSim 기반 SSD FTL 시뮬레이터를 대신할 리눅스 FUSE 기반의 가볍고 유연한 시뮬레이터를 구현하고 검증한다. 기능 검증을 위한 시범 시뮬레이션 적용결과 다른 연구와의 동일성이 확인되었고, 리눅스 커널 캐시의 영향을 분석할 수 있는 차별성도 보였다.

[Abstract]

The flash memory constituting SSD needs to be erased before writing. And in the flash memory, the size of read•write/erase unit are different and the lifespan is reduced by erasing. In order to overcome these disadvantages and keep up the fast access speed, which is the biggest advantage of SSD, FTL algorithm optimization experiments from various viewpoints such as streaming service are being vigorously conducted through simulation. In this study, we implement and verify a lightweight and flexible Linux FUSE-based simulator that can replace the existing heavy and complex DiskSim-based SSD FTL simulator. As a result of a sample simulation for functional verification of the implemented simulator, we confirmed the consistency with other studies and showed the excellency capable of analyzing the effect of Linux kernel cache.

색인어 : 캐시, 플래시 메모리, FTL, FUSE, SSD

Key word : Cache, Flash Memory, FTL(Flash Translation Layer), FUSE(Filesystem in Userspace), SSD(Solid State Disk)

http://dx.doi.org/10.9728/dcs.2019.20.12.2545

Received 04 October 2019; **Revised** 20 November 2019
Accepted 15 December 2019

***Corresponding Author; Ki-Hyeon Kwon**

Tel: +82-33-570-6400
E-mail: kweon@kangwon.ac.kr

Ⅰ. 서 론

SSD (Solid State Disk)는 기계 장치가 아닌 고체 상태의 반도체 기억장치 칩들을 이용하여 구성한 대용량 저장장치로서, 여기에 사용되는 반도체 기억 소자를 플래시 메모리라 한다 [1]. ROM의 성격을 가지면서도 저장 내용을 지우고 다시 기록할 수 있는 EPROM이나 EEPROM은 그 과정이 번거로워 디스크와 같이 저장 내용의 수정 빈도가 높은 환경에는 적합하지 않다. EEPROM을 개선한 플래시 메모리는 '플래시'라는 어원에서 보는 바와 같이 지우는 과정이 신속・단순하여, 저장 내용에 대한 빈번한 수정 쓰기가 필수적인 보조기억장치로의 활용이 가능하다 [2]-[3]. 플래시 메모리의 가장 큰 특징은 한 번 기록된 영역은 반드시 초기화(지우기) 후 다시 기록해야 한다는 점에 있다. 하드 디스크와 같이 대용량 저장을 위한 SSD에는 직접도를 높인 NAND 플래시 메모리가 사용되는데 (이하 SSD는 NAND 플래시 메모리 기반을, 플래시는 NAND 플래시 메모리를 의미함), 이 경우 쓰기·읽기 단위의 크기(페이지)와 지우기 단위의 크기(블록)가 달라 디스크 운영 관리가 단순하지 않다 [4]. 또한, 지우기 빈도에 비례하여 해당 영역의 수명이 단축되는 마모 현상은 SSD의 효율적 이용 방안을 더욱 복잡하게 만든다. 이런 특징들 외에 표 1에서 보는 바와 같이 읽기, 쓰기, 지우기 작업에 소요되는 시간과 에너지가 모두 달라 [1][5], 스트리밍 서비스, 트랜잭션 처리, 휴대형 기기 등 사용 환경에 따른 SSD 운영의 최적화가 필수적이다.

이 논문에서는 SSD의 성능평가 시뮬레이터를 리눅스 FUSE 기반으로 가볍고 유연하게 구현하여 사용 환경에 따른 SSD의 운영 최적화 연구에 기여하고자 한다. 이를 위하여 2장에서 SSD 관련 연구와 리눅스 FUSE를 살펴보고, 3장에서 리눅스 FUSE 기반 SSD 성능평가 시뮬레이터를 구현한다. 4장에서는 구현된 시뮬레이터를 검증하며 마지막으로 5장에서 본 논문의 결론을 맺는다.

표 1. 플래시 메모리의 접근 시간과 에너지 소모
Table 1. Access Time and Energy Consumption of Flash Memory

	read	write	erase
time(㎲)	25	250	1500
energy(㎌)	0.5	7.5	40

Ⅱ. SSD 관련 연구 및 리눅스 FUSE 고찰

2-1 SSD 관련 연구

SSD는 하드 디스크와 같이 블록 디바이스로 동작하지만 기존에 할당된 블록(섹터)의 데이터를 수정하는 순간 새로운 데이터는 다른 블록에 저장되어야 하고, 기존 블록은 가비지(garbage)가 된다. 이는 플래시 메모리의 읽기·쓰기 단위의 크기와 지우기 단위의 크기가 달라 동일한 블록에서의 읽기→지우기→쓰기가 불가능하기 때문인데, 이로 인하여 운영체제에서는 SSD의 물리 블럭을 고정적으로 지정할 수 없다. 이를 해결하기 위해서 그림 1과 같이 SSD는 운영체제에게 고정된 블럭번호를 제공하되, SSD 내부에서 실제로 할당된 블럭번호로 변환하는 방법을 사용하는데 이러한 변환 기능의 범주가 FTL (Flash Translation Layer)이고, FTL은 SSD의 탄생과 함께 수반된 태생적 필수 기능이다 [6]. FTL은 SSD가 제공하는 제한된 RAM 공간에서 핵심인 변환 기능 외에 최적의 블럭 할당 알고리즘 [7], 캐싱 알고리즘 [8]-[9], 가비지 수집 알고리즘 [4][10] 등 다양한 기능을 수행해야 하고, 이들의 최적 조합에 대한 탐구는 시뮬레이션으로 이루어진다.

대부분의 SSD 성능평가 시뮬레이션은 널리 알려진 하드 디스크 성능평가 도구인 DiskSim [11]을 기반으로 변형하여 사용한다 [12]-[14]. 그런데 DiskSim은 입·출력 버스나 제어기 특성 등 하드웨어를 포함하여 저장장치 하부의 모든 특성에 대한 모델링을 지원하기 때문에 그 규모가 방대하고 이식 과정이 간단하지 않다 [15]. 즉, DiskSim은 운영체제 관점에서의 FTL 기능 구현 시뮬레이션 도구로서는 너무 무겁다.

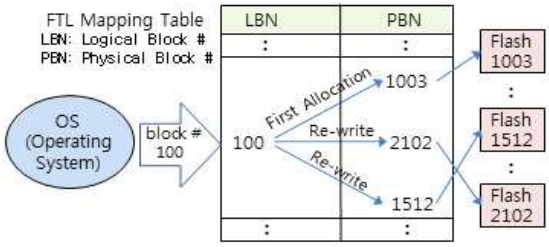

그림 1. FTL의 변환 기능
Fig. 1. Translation Function of FTL

2-2 리눅스 FUSE

파일시스템은 운영체제의 핵심 구성요소 중의 하나로서 디스크의 물리 저장 공간을 바탕으로 사용자에게 논리적인 저장 조직체계를 제공한다. 운영체제 내에 탑재된 파일시스템과 달리 사용자가 구성하여 제공하는 파일시스템을 사용자 영역 파일시스템이라 하고 [16], 그 대표적인 사례가 리눅스의 FUSE (Filesystem in Userspace) [17]이다. 그림 2는 인터넷 데이터를 사용자에게 파일시스템 뷰로 조직하여 제공하는 리눅스

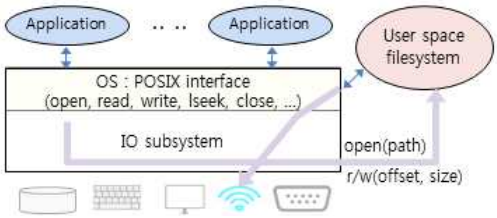

그림 2. 리눅스 FUSE 개념
Fig. 2. Concept of Linux FUSE

FUSE 적용 연구 [18]의 개념도이다. [19]는 리눅스 FUSE를 이용하여 파일 업데이트(write) 시 해당 부분을 이미 저장된 내용과 비교하여 일치할 경우 쓰기 작업을 생략하여 SSD 등의 디스크 수명을 연장하는 일종의 래퍼 파일시스템을 구현하였다.

리눅스 FUSE는 커널이 파일시스템을 운영하기 위해 사용자에게 요구하는 파일시스템 접근 프로시저들로 이루어지고, 그 중 대표적인 것들은 아래와 같다.

- create(path, mode) : FUSE 영역 즉, 사용자 파일시스템이 마운트된 지점 이하에서의 경로명과 모드를 전달받아 해당 파일을 생성하고, 이 파일을 특정할 자체 id를 돌려준다.
- open(path, mode) : 이미 존재하는 path가 가르키는 파일을 주어진 모드로 개방하여 이 파일을 특정하는 id를 돌려준다.
- mkdir(path, mode) : 주어진 경로명에 디렉터리를 생성한다.
- getattr(path, buf) : 주어진 경로명의 파일이나 디렉터리의 속성(stat)을 돌려준다.
- readdir(path, buf) : 주어진 경로명의 디렉터리 엔트리들의 속성(name, stat)을 돌려준다.
- read(id, buf, offset, size) : open(), create()에 의해 개방된 파일을 읽어 돌려준다.
- write(id, buf, offset, size) : open(), create()에 의해 개방된 파일에 주어진 내용을 쓴다.
- release(id) : open(), create()에 의해 개방된 파일을 폐쇄한다.
- ioctl(id, cmd, arg) : open(), create()에 의해 개방된 파일에 대하여 특정 커멘드를 수행한다.
- unlink(path) : 주어진 경로명의 파일을 제거한다.
- rmdir(path) : 주어진 경로명의 디렉터리를 제거한다.

위 프로시저 들은 그림 3과 같이 fuse_main() 라이브러리를 통해서 커널에 등록하고, 이 때 ac, av에는 해당 사용자 파일시스템을 가상으로 마운트할 지점(디렉터리 경로명)을 전달한다.

```
static struct fuse_operations ftlfs_oper = {
    .chmod          = ftlfs_chmod,
    .chown          = ftlfs_chown,
    .utimens        = ftlfs_utimens,
    .getattr        = ftlfs_getattr,
    .readdir        = ftlfs_readdir,
    .truncate       = ftlfs_truncate,
    .mkdir          = ftlfs_mkdir,
    .mknod          = ftlfs_mknod,
    .create         = ftlfs_create,
    .open           = ftlfs_open,
    .access         = ftlfs_access,
    .read           = ftlfs_read,
    .write          = ftlfs_write,
    .ioctl          = ftlfs_ioctl,
    .rmdir          = ftlfs_rmdir,
    .unlink         = ftlfs_unlink,
    .flush          = ftlfs_flush,
    .release        = ftlfs_release,
};
int main(int ac, char *av[])
{    int    r;   /* Usage: ./ftlfs /home/ftlfs */
     r = fuse_main(ac, av, &ftlfs_oper, NULL);
     return r;
}
```

그림 3. FUSE 프로시저의 커널 등록
Fig. 3. Enrollment of FUSE Procedures in Kernel

Ⅲ. SSD FTL 시뮬레이터 구현

3-1 가상 SSD 파일 및 디렉터리

FUSE로 구현된 사용자 파일시스템(이하 FTL 파일시스템)에 접근하고자 하는 어플리케이션은 해당 파일시스템의 마운트 지점 이하를 접근한다. 이 때, 커널은 마운트 지점 이하의 경로명만을 FTL 파일시스템에 전달한다. FTL 파일시스템은 전달된 경로명을 의도한 실제 경로명으로 대응시켜 처리한다. 이를테면 FTL 파일시스템이 “/home/ftlfs"에 마운트되어 있을 때, 어플리케이션이 경로명으로 “/home/ftlfs/files/f000"을 지정하면 커널은 ”/files/f000"을 FTL 파일시스템에 전달하고, FTL 파일시스템은 실제 파일시스템의 특정 디렉터리(예 “/FTL_FS/”) 아래에 해당 경로명을 주입하여 “/FTL_FS/files/f000"를 최종 경로명으로 설정한다. 이와 같은 경로명 변환은 create(), open(), unlink(), mkdir(), rmdir() 등 경로명을 직접 필요로하는 모든 FUSE 프로시저들에게 적용된다. 그림 4에 FTL 파일시스템의 open() 프로시저를 보였다.

```
typedef struct ftable {
    size_t  size;
    pthread_mutex_t    Mutex_fd;
} O_FTBL;
O_FTBL Ftbl[O_NFTBL];
static int ftlfs_open(const char *path,
                                    struct fuse_file_info *fi)
{   char    fpath[L_PATH];
    int     id;
    sprintf(fpath, "/FTL_FS/%s", path);
    if ((id = open(buf, O_RDWR)) < 0)
          return -errno;
    Ftbl[fd].size = st.st_size / sizeof(int) * F_PGSIZE;
    fi->fh = id;
    fi->direct_io = 1; // bypass Linux OS cache
    fi->nonseekable = 0;
    return 0;
}
```

그림 4. FTL 파일시스템의 open() 프로시저
Fig. 4. Open() Procedure of FTL Filesystem

3-2 FTL 파일시스템의 논리 및 물리 페이지

SSD를 사용하는 운영체제는 SSD를 논리 페이지(섹터)들의 집합으로 보고 할당된 논리 페이지를 파일시스템에 기록할 것이므로 모든 논리 페이지에 대한 할당 여부가 관리되어야 한다. 이를 위하여 FAT(File Allocation Table) 방법을 적용하여 ftlfs_get_lpn(), ftlfs_put_lpn() 프로시저를 둔다. 할당된 논리 페이지 번호는 FTL의 쓰기 프로시저(ftl_write())에 전달되어 실제 할당된 SSD 물리 페이지 번호로의 대응을 위해 매핑 테이블에 삽입된다. 이 때, 운영체제 관점에서의 논리 페이지 번호는 개방된 해당 파일의 내용으로 저장하여 관리한다. 이를테면 “/home/ftlfs/files/f000"에 논리 페이지 3241, 4352, 5463 번이 할당되었다면 이 파일의 내용으로 그림 5와 같이 이들 세 개의 번

호를 저장한다. 이 때, 주어진 입·출력 위치 offset에 대응되는 논리 페이지는 offset/page_size 식으로 얻을 수 있으며, 이 파일이 차지하는 페이지 단위 공간은 3*page_size로 얻을 수 있으나 정확한 크기는 자체로 관리하는 파일 테이블(Ftbl[])에 관리한다. 그림 4를 기반으로 작성된 FTL 파일시스템의 write() 프로시저를 그림 5에 보였고, read() 프로시저도 이와 유사하게 작성된다.

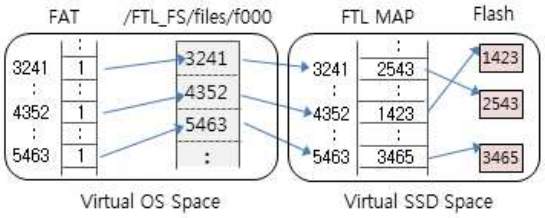

그림 5. FTL 파일시스템의 논리 및 물리 페이지
Fig. 5. Logical and Physical Page of FTL Filesystem

```
static int ftlfs_write(const char *path, const char *buf,
            size_t size, off_t offset, struct fuse_file_info *fi)
{   int  fd = fi->fh, s_pgn, npage, p_off, fseek,
         rpage, *ptbl, tsize = size, csize, i;
    O_FTBL *fp = Ftbl + fd;
    char     p_buf[F_PGSIZE]; const char *bp;
    s_pgn = offset / F_PGSIZE; //first page
    p_off = offset % F_PGSIZE; //offset in the first
    npage = ((p_off+size) + F_PGSIZE -1) / F_PGSIZE;
    if (npage <= 0) return(0);
    ptbl = (int *)malloc(npage * sizeof(int));
    memset(ptbl, 0, npage * sizeof(int));
    fseek = s_pgn * sizeof(int);
    rpage = (fp->size - fseek + sizeof(int) - 1) / sizeof(int);
    if (rpage > npage) rpage = npage;
    pthread_mutex_lock(&fp->Mutex_fd);
    if (rpage > 0) {
        lseek(fd, fseek, SEEK_SET);
        read(fd, ptbl, rpage * sizeof(int));
    }
    for (i = 0, bp = buf; i < npage; i++) {
        if (!ptbl[i]) {
            ptbl[i] = ftlfs_get_lpn();
            memset(p_buf, 0, sizeof(F_PGSIZE));
        } else
            ftl_read(ptbl[i], p_buf);
        csize = size > F_PGSIZE-p_off ? F_PGSIZE-p_off : size;
        memcpy(p_buf + p_off, bp, csize);
        ftl_write(ptbl[i], p_buf);
        p_off = 0; offset += csize; bp += csize; size -= csize;
    }
    if (offset > fp->size) fp->size = offset;
    lseek(fd, fseek, SEEK_SET);
    write(fd, ptbl, npage * sizeof(int));
    pthread_mutex_unlock(&fp->Mutex_fd);
    return(tsize);
}
```

그림 6. FTL 파일시스템의 write() 프로시저
Fig. 6. Write() Procedure of FTL Filesystem

3-3 FTL 변환 및 쓰기·읽기 모듈

FTL 모듈은 FTL 파일시스템으로부터 논리 페이지 번호를 전달받아 새로운 플래시 물리 페이지를 할당하여 맵 테이블에 대응시킨 후 쓰거나, 이미 대응되어있는 페이지에 대하여 읽기를 수행한다. 이 때, FTL 캐시 전략에 따라 플래시 물리 페이지에 대한 쓰기·읽기는 일어나지 않을 수 있다. 이 과정에서 운영체제 수준에서의 입·출력 요구 대비 물리적 입·출력 비율을 얻을 수 있다. 그림 7 ~ 그림 8은 FTL 쓰기·읽기 프로시저이고, 여기서 lpn은 논리 페이지 번호를, ppn은 물리 페이지 번호를, fbn과 fpn은 플래시 블록 및 페이지 번호를, OWrite와 ORead는 운영체제의 요구 빈도를, TGpage와 TBpage는 플래시의 가비지 페이지와 데이터 페이지를 각각 카운트한다.

```
void  ftl_write(int lpn, void *buf)
{   pthread_mutex_lock(&Mutex_ftl);  OWrite++;
    switch(Flag_cache1) {
    case FTL_RWCACHE: ftl_write_rwcache(lpn, buf); break;
    default         : ftl_write_nocache(lpn, buf); break;
    }
    ftl_collect_garbage();
    pthread_mutex_unlock(&Mutex_ftl);
}
void  ftl_write_nocache(int lpn, void *buf)
{   int     ppn, fbn, fpn;
    GET_PPN(lpn) > 0 ? TGpage++ : TBpage++;
    ppn = ftl_get_a_free_page(&fbn, &fpn);
    flash_write(fbn, fpn, buf, lpn);
    PUT_PPN(lpn, ppn); // update map table
}
```

그림 7. FTL의 write() 프로시저
Fig. 7. Write() Procedure of FTL

```
void  ftl_read(int lpn, void *buf)
{   pthread_mutex_lock(&Mutex_ftl); ORead++;
    switch(Flag_cache1) {
    case FTL_RWCACHE: ftl_read_rwcache(lpn, buf); break;
    default         : ftl_read_nocache(lpn, buf); break;
    }
    pthread_mutex_unlock(&Mutex_ftl);
}
void  ftl_read_nocache(int lpn, void *buf)
{   int   ppn, fbn, fpn;
    ppn = GET_PPN(lpn);  // SSD physical page#
    fbn = GET_FBN(ppn);  // flash block#
    fpn = GET_FPB(ppn);  // flash page# in a block;
    flash_read(fbn, fpn, buf);
}
```

그림 8. FTL의 read() 프로시저
Fig. 8. Read() Procedure of FTL

3-4 FTL 캐시 모듈

실험하고자 하는 캐시 정책에 따라 FTL 쓰기·읽기 프로시저에 캐시 투과 과정을 둔다. 캐시의 크기는 실험 대상인 SSD가 제공하는 RAM의 범위로 설정하되, 실제 데이터를 기록할 필요는 없으므로 캐시 검색 키 등 운영상 필요한 항목만을 설정한다. 그림 9는 캐시를 그룹핑하고, 쓰기 시 곧바로 플래시까지 저장하지 않고 일단 캐시에만 저장하는 이른바 Read/Write 캐싱 기법을 사용하는 예이고, 여기서 TDcache는 캐시와 플래시의 내용이 불일치하는 불결(dirty) 캐시를 카운트한다. 그 밖에 캐시 히트 읽기와 쓰기를 HRead와 HWrite에 카운트 한다.

```
typedef struct ftl_cache {
    int     lpn, ppn, ref_count, dirty;
    char    buf[1];
} FCACHE;
void ftl_write_rwcache(int lpn, void *buf)
{   FCACHE  *fcp;
    int     ppn, cgr, fbn, fpn;
    ppn = GET_PPN(lpn), cgr = GET_CGR(lpn);
    fcp = ftl_find_rwcache(lpn, ppn, cgr);
    if (fcp->lpn == lpn) {          // hit
        fcp->ref_count++; HWrite++;
        if (fcp->dirty++ == 0) TDcache++;
    }
    else {  // free, clean, dirty
        if (fcp->lpn && fcp->dirty)
            ftl_cache_flush_fcp(fcp),PUT_PPN(fcp->lpn, fcp->ppn);
        fcp->lpn = lpn, fcp->ppn = ppn,
        fcp->ref_count = fcp->dirty = 1; TDcache++;
        PUT_CGR(lpn, ppn, FTL_CGR(fcp));
    }  // copy_cache(fcp->buf, buf);
}
```

그림 9. FTL 캐시의 read() 프로시저
Fig. 9. Read() Procedure of FTL Cache

3-5 FTL 가비지 수집 모듈

그림 9에 플래시 블록 하나를 랜덤하게 선택하여 그곳에 포함된 가비지를 회수하는 예를 보였다. 이 때 가비지 수집 대상이 되는 플래시 블록에 포함된 각 페이지가 빈 것인지, 유효한 것인지, 가비지인 것인지의 판단은 해당 페이지의 여분(spare) 영역에 기록된 lpn(논리 페이지) 값에 의해 아래와 같이 결정된다.

- lpn이 0(초기상태)이면 빈 페이지
- lpn에 대한 맵 테이블 내용과 해당 페이지가 일치하면 유효 페이지, 그렇지 않으면 가비지 페이지

가비지 수집 대상 블록에 포함된 유효 페이지 들은 다른 블록의 빈 페이지에 복사되어야 하므로, 이들 유효 페이지와 관련된 맵 테이블과 캐시의 수정이 필요하다. 그림 10에 보인 ftl_

```
typedef struct          collect {
   int      pbn, tbpage, tgpage, tepage;
   FCACHE   *fcp;
} COLL;
void  ftl_collect_garbage()
{ FCACHE    *fcp, *tfcp;
  COLL      col;
  while ((Flag_cache1 != FTL_RWCACHE && TEpage < FTL_HWATER) ||
           (Flag_cache1 == FTL_RWCACHE && TEpage < TDcache)) {
     fcp = ftl_get_random_collect_block(&col);
     switch(Flag_cache) {
     case FTL_RWCACHE:
       for (tfcp = fcp; tfcp < FCache + FTL_NCACHE; tfcp++) {
         if (!tfcp->lpn || !tfcp->ppn) continue;
         if (GET_BLKN(tfcp->ppn) == col.pbn) {
            PUT_CGR(tfcp->lpn,0,GET_CGR(tfcp->lpn));
            tfcp->ppn = 0; if (tfcp->dirty++ == 0) TDcache++;
         }
       }
     }
     ftl_copy_collect_busy_page(&col);
     ftl_erase_collect_block(&col);
  }
}
```

그림 10. FTL의 Garbage_collect() 프로시저
Fig. 10. Garbage_collect() Procedure of FTL

get_random_collect_block() 프로시저는 선택된 블럭의 유효 페이지들을 캐시의 앞 부분에 읽고, 바로 다음 캐시 위치를 돌려준다.

가비지 수집 시점은 다양한 시점에서 다양한 방법으로 이루어질 수 있는데, 여기서는 빈 페이지 개수가 하한선 아래로 내려가면 그림 7의 FTL의 write() 프로시저에서 수행하도록 한다.

3-6 플래시 메모리 모듈

FTL 캐시와 마찬가지로 플래시 메모리에도 실제 데이터를 저장할 필요는 없으므로 페이지의 데이터 영역은 제거하고 여분 영역에 논리 페이지 번호(lpn) 항목만 배치하되, 그 개수가 방대하므로 디스크 파일에 메모리 주소를 대응시켜 할당한다. 그림 11에 플래시 메모리 관련 프로시저를 보였고, 이들 프로시저에서는 물리적 입·출력 및 지우기 빈도를 FRead, FWrite, FErase에 각각 카운트한다.

```
typedef struct sector_spare {
     uint   lpn, eeount;
} SPARE;
typedef struct flash_block {
     SPARE  sector_spare[F_NBPAGE];
} FBLK;
void flash_init()
{    int    i, fd;
     FBLK   fblk;
     fd = open("File/Flash", O_RDWR | O_CREAT, 0770);
     memset(&fblk, 0, sizeof(FBLK));
     for (i = 0; i < F_NBLOCK; i++)
         write(FlashFd, &fblk, sizeof(FBLK));
     Block = (FBLK *)mmap(NULL, F_NBLK*sizeof(FBLK),
                PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED,fd,0);
}
void flash_write(int fbn, int fpn,void *buf,int lpn)
{    FWrite++
     Block[pbn].sector_spare[ppo].lpn = lpn;
}
int flash_read(int fbn, int fpn, void *buf)
{    FRread++;
     return(Block[pbn].sector_spare[ppo].lpn);
}
void flash_erase(int pbn)
{    FErase++;
     memset(&Block[pbn], 0, sizeof(FBLK));
}
```

그림 11. 플래시 메모리 write()/read() 프로시저
Fig. 11. Write()/read() Procedure of Flash

Ⅳ. 구현 시뮬레이터 검증

4-1 검증 방법 및 환경

구현 시뮬레이터의 기능 검증을 위해 에너지 소모 관점에서 캐시 전략이 미치는 영향을 분석한 연구 [9]의 실험을 동일하게 수행하여 동일한 결과가 도출되는지를 비교하여 기능을 검증하고, 리눅스 운영체제 자체의 캐시 정책을 반영한 새로운 분석 기능을 확인한다. 기능 검증은 그림 12 형태의 입·출력 어플리

케이션 유형을 표 1과 같이 분류하고, 이들을 표 2와 같이 3 가지로 조합한 각 워크로드를 LRU(Least Recently Used)와 NUR (Not Used Recently) 두 가지 캐시정 책별로 실행한 결과를 분석한다. 표 1의 각 어플리케이션들은 약 40MB 크기의 파일에 대하여 1,000~9,000 바이트의 레코드를 50,000번 입·출력하고, 실험 플랫폼은 표 3과 같다.

```
#include <pthread.h>
typedef struct arg {
    char   fname[32];  int  thn;
} ARG;
pvoid *do_rndom_rdwr(void *arg)
{   int     fd, csize, tsize, i; char buf[1024*10];
    ARG     *ap = (ARG *)arg;
    fd = open(ap->fname, O_RDWR|C_CREAT, 0600);
    for (i = tsize = 0; i < Nloop_rec; i++) {  // file creation
        csize = (rand_r(&seed) % 8000) + 1000;
        write(fd, buf, csize); tsize += csize;
    }
    for (i = 0; i < Nloop_rec; i++) {
        csize = (rand_r(&seed) % 8000) + 1000;
        loc = rand_r(&seed) % tsize;
        lseek(fd, loc, SEEK_SET); read(fd, buf, csize);
        lseek(fd, loc, SEEK_SET); write(fd, buf, csize);
    }
    close(fd); unlink(ap->fanme); pthread_exit((void *)NULL);
}
int main(int ac, char *av[])
{  ARG        arg[MAX_THREAD], *ap;
   pthread_t  tid[MAX_TGREAD], *tp; int  i;
   for (i = 0, ap = arg, tp = tid; i < Nthread; i++, ap++, tp++) {
       sprintf(ap->fname, "/home/ftlfs/f%.3d", i); ap->thn = i;
       pthread_create(tp, NULL, do_random_rdwr, (void *)ap);
   }
   for (i = 0, tp = tid; i < Nthread; i++, tp++)
       pthread_join(*tp, NULL);
   return(0);
}
```

그림 12. 워크로드 어플리케이션 예
Fig. 12. A Sample Workload Application

표 1. 입·출력 어플리케이션 분류
Table 1. Classification of I/O Application

Name	I/O Pattern of Applications
AP0	squential read from offset 0
AP1	squential write from offset 0
AP2	random read with random offset
AP3	random write with random offset
AP4	random read and rewrite with random offset

표 2. 입·출력 워크로드 분류
Table 2. Classification of I/O Workload

Name	Number of I/O Application Thread
WL0	AP0~AP1: 15 threads for each AP
WL1	AP2~AP4: 10 threads for each AP
WL2	AP0~AP4: 6 threads for each AP

표 3. 실험 환경
Table 3. Experimental Environment

Items	Specification
CPU	Intel E7500(Duo Core), 2.93GHz
Memory	2.0GB
OS	LINUX(Ubuntu 16.04)

4-2 실험 결과 및 분석

1) 시뮬레이터 기능 검증

표 4에 표 2의 각 워크로드들이 FTL에서 유발한 캐시 읽기 적중(CR), 캐시 쓰기(CW), 플래시 읽기(FR), 플래시 쓰기(FW) 등의 빈도에 따른 캐시 적중률을 보였고, 그림 13에는 그 추이를 그래프로 보였다. 이 결과는 연구 [9]와 동일하다.

표 5에는 표 4의 FW 및 FW와 플래시 지우기(FE), 가비지 수집에 따른 플래시 읽기(GR) 및 플래시 쓰기(GW) 빈도를 종합한 결과에 표 1의 플래시 연산별 에너지 소모량을 적용하여 산출한 워크로드별 총 에너지 소모량을 보였고, 그림 14에 그 추이를 그래프로 보였다. 이는 LRU가 NUR보다 적중률은 높지만 에너지 효율성은 오히려 낮을 수 있다는 연구 [9]와 동일한 결과이고, 이는 시뮬레이터의 구현 방향이 적합함을 의미한다.

표 4. 캐시 관리 정책별 적중률
Table 4. Hit Ratio by Cache Management Policy

		CR	CW	FR	FW	H(%)
L	WL0	1,789,003	1,506,102	925,475	387,217	71.51
R	WL1	972,199	1,458,055	1,742,844	299,815	54.33
U	WL2	1,299,919	1,500,643	1,415,054	325,450	54.75
N	WL0	1,513,651	1,478,524	1,200,827	247,316	67.38
U	WL1	747,404	1,426,576	1,967,639	249,454	49.50
R	WL2	1,044,628	1,474,116	1,670,345	248,657	56.75

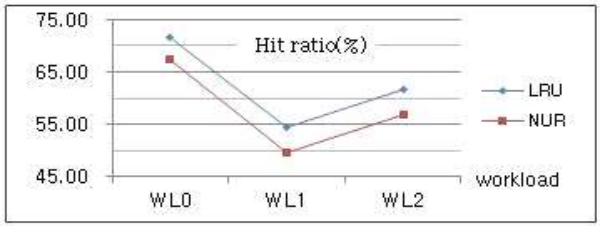

그림 13. LRU와 NUR의 캐시 적중률
Fig. 13. Cache Hit Ratio of LRU and NUR

표 5. 캐시 관리 정책별 에너지 소모량
Table 5. Energy Consumption by Cache Management Policy

		FR	FW	FE	GR	GW	E(J)
L	WL0	925,475	387,217	2,080	5,890,223	391,247	9.72
R	WL1	1,742,844	299,815	11,314	27,480,520	2,842,512	40.20
U	WL2	1,415,054	325,450	7,034	15,855,552	1,731,223	25.37
N	WL0	1,200,827	247,316	3,100	4,507,683	792,124	11.29
U	WL1	1,967,639	249,454	9,010	9,998,992	2,302,904	26.76
R	WL2	1,670,345	248,657	6,733	8,315,807	1,720,923	21.02

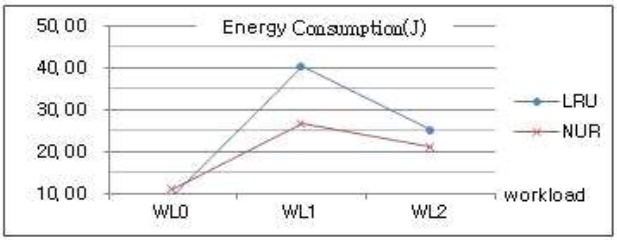

그림 14. LRU와 NUR의 에너지 소모량
Fig. 14. Energy Consumption of LRU and NUR

2) 리눅스 커널 캐시 기반 실험

표 4~5의 실험은 리눅스 커널의 캐시를 우회하여 워크로드의 입·출력 요구를 FTL에 직접 전달하는 환경이었으나, FUSE 기반의 시뮬레이터는 리눅스 커널의 캐시를 경유하여 전달함으로써 실제 운영체제 환경에서의 실험이 가능하다. 이를 위해서 그림 4의 FTL 파일 개방 프로시저에서 direct_io 항목을 0으로 설정한다. 표 6은 커널 캐시를 우회하는 경우(LRU, NUR)와 경유하는 경우(LRU-cache, NUR-cache)의 총 입·출력 량을 보였는데, 커널 캐시를 경유하는 경우 그 양이 약 8% 가량 증가하지만, 그림 15의 총 에너지 소모량을 보면 오히려 약 절반으로 감소함을 볼 수 있다. 이는 그림 14에 보인 바와 같이 커널 캐시를 경유하면 FTL의 캐시 적중률이 크게 높아지기 때문인 것으로 분석되는데, 이 경우 FTL 캐시 적중률이 높아지는 이유에 대해서는 연구가 더 이루어져야 한다.

표 6. 커널 캐시에 따른 총 입·출력 량(회수)
Table 6. Total I/O Traffic (times)

LRU	NUR	LRU-cache	NUR-cache
13,621,776	13,269,137	14,162,184	14,328,573

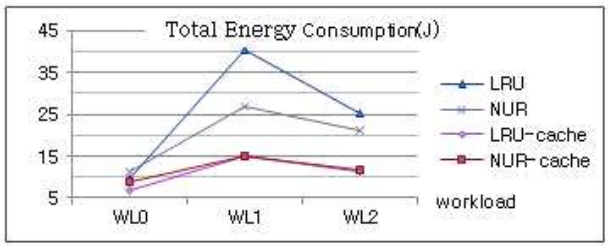

그림 15. 커널 캐시에 따른 전체 에너지 소모량
Fig. 15. Total Energy Consumption

Ⅴ. 결 론

쓰기에 사용된 페이지는 지우기 후 다시 쓸 수 있고, 지우기 단위인 블록과 쓰기·읽기 단위인 페이지의 크기가 다르며, 지우기에 따라 수명이 단축되는 등의 SSD 특성들은 SSD를 하드 디스크 대용으로 사용하기 위한 최적화 작업을 매우 어렵게 한다. SSD의 최적화 연구는 대부분 DiskSim 기반의 시뮬레이터를 사용해 이루어지는데 이 유형의 시뮬레이터는 소스 코드 규모가 총 38,000라인 정도로 방대하여 다양한 FTL 기능을 자유롭게 접목하는 작업이 용이하지 않다. 이 연구에서는 리눅스 FUSE가 제공하는 사용자 영역 파일 시스템을 기반으로 FTL의 변환 기능, 캐싱 기능, 가비지 수집 기능의 틀을 총 2,500라인 정도의 경량으로 구현하고, FTL 캐시 정책에 따른 에너지 소모량을 분석하는 시뮬레이션을 통해 그 기능을 검증하였다. 또한, 순전히 어플리케이션 수준에서 동작한 다른 툴과는 다르게 리눅스 커널 캐시를 경유함으로써 실제 리눅스 파일시스템 하에서의 SSD 최적화 작업도 가능함을 보였다. 앞으로 본 구현 시뮬레이터를 기반으로 보다 더 정확하면서도 신속한 SSD 최적화 작업 연구가 이루어질 것이다.

감사의 글

이 논문은 2018년도 강릉원주대학교 학술연구조성비 지원에 의하여 수행되었습니다.

참고문헌

[1] J. H. Kim, *Computer Architecture,* 5th ed. Life & Power Press, pp. 349-372, 2019.

[2] M. Fujio, I. Hisakazu, *Semiconductor memory device and method for manufacturing the same*, US Patent 4531203, 1985.

[3] https://en.wikipedia.org/wiki/Flash_memory

[4] C. Matsui, A. Arakawa, C. Sun, K. Takeuchi, "Write Order-Based Garbage Collection Scheme for an LBA Scrambler Integrated SSD," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 25, No. 2, pp. 510-519, 2017.

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

[6] T. S. Chung, D. J. Park, S. W. Park, D. H. Lee, S. W. Lee, H. J. Song, "A survey of Flash Translation Layer," *Journal of Systems Archicture*, Vol. 55, pp. 332-343, 2009.

[7] D. B. Yeo, J. Y. Paik, T. S. Chung, "Request-Size Aware Flash Translation Layer Based on Page-Level Mapping," *15th International Symposium on Distributed Computing and Applications for Business Engineering(DCABES)*, pp. 68-71, 2016.

[8] J. Boukhobza, P. Olivier, S. Rubini, "SCFTL: An efficient

caching strategy for page-level flash translation layer," *21st Euromicro International Conference on Parallel, Distri buted, and Network-Based Processing*, pp. 94-101, 2013.

[9] H. B. Lee, T. Y. Chung, "A Comparative Analysis on Page Caching Strategies Affecting Energy Consumption in the NAND Flash Translation Layer," *IEMEK Journal of Embedded Systems and Applications*, Vol. 13, No. 3, pp.109-116, 2018.

[10] T. H. Lee, J. H. Cha, "A File Clustering Algorithm for Wear-leveling," *Journal of Digital Contents Society(JDCS)* ,Vol. 14, No. 1, pp. 51-57, 2013.

[11] 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.

[12] Y. J. Kim, B. Tauras, A. Gupta, B. Urgaonkar, "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives," *2009 First International Conference on Advances in System Simulation*, pp. 125-131, 2009.

[13] A. Gupta, Y. Kim, B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," *Proceedings of the 14th ACM International conference on Architectural support for programming languages and operating systems (ASPLOS '09)*, 2009.

[14] C. M. Lee, Y. J. Won, "Develop SSD Performance and Power Estimation Simulator," *Korea Computer Congress 2010(KCC2010)*, Vol. 37, No. 2B, pp. 292-297, 2010.

[15] https://www.pdl.cmu.edu/DiskSim/Compiling_DiskSim3.0 _v1.0.pdf

[16] https://en.wikipedia.org/wiki/Filesystem_in_Userspace

[17] http://fuse.sourceforge.net

[18] H. B. Lee, T. Y. Chung, "A Virtual File System for IoT Service Platform Based on Linux FUSE," *IEMEK Journal of Embedded Systems and Applications*, Vol 10, No. 6, pp. 139-150, 2015.

[19] J. H. Myeong, I. C. Hwang, O. Y. Kwon, "Design Flexible Deduplication Filesystem on Personal Computer Environ ment," *International Conference on Future Information & Communication Engineering*, Vol. 10, No. 1, pp. 277-280, 2018.

이형봉 (Hyung-Bong Lee)

1984년 : 서울대학교 계산통계학과(학사)
1986년 : 서울대학교 대학원 계산통계학과(석사)
2000년 : 강원대학교 대학원 컴퓨터과학과(박사)

1986년 ~ 1994년: LG전자 컴퓨터연구소
1994년 ~ 1999년: 한국디지털(주)
2004년 ~ 현 재: 강릉원주대학교 컴퓨터공학과 교수
※관심분야 : 무선 통신 (Wireless Networks), 센서 네트워크 (Sensor Networks), 임베디드 시스템 (Embedded Systems), 사물 인터넷 (IOT) 등

권기현 (Ki-Hyeon Kwon)

1993년 : 강원대학교 컴퓨터과학과(학사)
1995년 : 강원대학교 대학원 컴퓨터과학과(석사)
2000년 : 강원대학교 대학원 컴퓨터과학과(박사)

1998년 ~ 2002년: 동원대학 인터넷정보과 교수
2002년~ 현 재: 강원대학교 전자정보통신공학부 교수
※관심분야 : 패턴 인식 (Pattern Recognition), 이미지 처리 (Image Processing), 사물 인터넷 (IOT) 등