http://dx.doi.org/10.5392/JKCA.2015.15.04.009

# 차세대 저장 장치에 최적화된 SWAP 시스템 설계

## Design of Optimized SWAP System for Next-Generation Storage Devices

**한혁**
동덕여자대학교 컴퓨터학과

Hyuck Han(hhyuck96@dongduk.ac.kr)

**요약**

Linux와 같은 발전된 운영 체제의 가상 메모리 관리 기술은 메인 메모리와 하드디스크와 같은 저장 장치를 이용하여 응용 프로그램에게 가상의 큰 주소 공간을 제공해준다. 최근 저장 장치는 속도의 측면에서 비약적인 발전을 보이고 있기 때문에 고속의 차세대 저장 장치를 메모리 확장에 이용하면 메모리를 많이 사용하는 응용의 성능이 좋아질 것이다. 그러나 기존 운영체제의 가상 메모리 관리 오버헤드 때문에 응용의 성능을 극대화시킬 수 없다. 이러한 문제를 해결하기 위해 본 논문은 차세대 저장 장치를 메모리 확장에 사용했을 때 쓰기 연산을 위한 블록 주소를 할당하는 향상된 알고리즘 및 시스템 튜닝 기법들에 대해 제안하였고, 제안된 기법들을 Linux 3.14.3의 가상 메모리 관리 시스템에 구현하였다. 그리고 구현된 시스템을 벤치마크를 이용하여 실험을 하였고, 마이크로 벤치마크의 경우에 평균 3배, 과학 계산 벤치마크 응용의 경우에 24%의 성능 향상이 있음을 보였다.

■ 중심어 : | 운영체제 | 가상 메모리 | 메모리 확장 | SWAP | 차세대 저장 장치 | 리눅스 |

**Abstract**

On modern operating systems such as Linux, virtual memory is a general way to provide a large address space to applications by using main memory and storage devices. Recently, storage devices have been improved in terms of latency and bandwidth, and it is expected that applications with large memory show high-performance if next-generation storage devices are considered. However, due to the overhead of virtual memory subsystem, the paging system can not exploit the performance of next-generation storage devices. In this study, we propose several optimization techniques to extend memory with next-generation storage devices. The techniques are to allocate block addresses of storage devices for write-back operations as well as to configure the system parameters, and we implement the techniques on Linux 3.14.3. Our evaluation through using multiple benchmarks shows that our system has 3 times (/24%) better performance on average than the baseline system in the micro(/macro)-benchmark.

■ keyword : | Operating System | Virtual Memory | Memory Extension | SWAP | Next-Generation Storage Device | Linux |

* 이 논문은 2014년도 동덕여자대학교 신임교원정착연구비 지원에 의하여 수행된 것임.

접수일자 : 2014년 12월 10일
수정일자 : 2015년 02월 06일
심사완료일 : 2015년 02월 09일
교신저자 : 한혁, e-mail : hhyuck96@dongduk.ac.kr

## I. 서 론

최근 스마트폰, 태블릿과 같은 모바일 휴대 기기의 급격한 증가로 인해 기존 인터넷 서비스 시스템들이 효율적인 서비스 요청 처리를 위해 대용량 메모리를 사용하는 소프트웨어를 사용하기 시작했다. 예를 들어 OLTP(On-Line Transaction Processing) 분야에서는 인 메모리 데이터베이스, 클라우드 컴퓨팅 분야에서는 memcached[1] 및 mongoDB와 같은 NoSQL(Not-Only SQL) 서버들이 사용되고 있다. 2012년 Twitter의 경우 빠른 데이터 요청 처리를 위해 수백 대의 서버를 이용하여 20TB 정도의 twemcache[1] 기반 메모리 캐쉬를 사용해서 폭발적인 데이터 요청에 대응했다. 이 밖에도 가상 데스크탑 인프라(VDI, Virtual Desktop Infrastructure) 분야, 대용량 데이터 처리 분야[14], HPC 분야에서도[15] 요구하는 메모리가 폭발적으로 증가하는 등 대용량 메모리를 요구하는 분야가 크게 늘어가고 있다.

그러나 DRAM의 경우 공정 미세화의 물리적 한계에 봉착하였고, 가격적인 측면에서 DRAM으로 메모리 요구량 증가에 대응하는 것은 현실적으로 어렵다. 이러한 이유로 DRAM 보다 저렴한 메모리 소자 기반의 저장 장치를 이용하여 메인 메모리를 확장하려는 많은 시도가 있었다. 대표적으로 플래시 SSD(Solid State Drive)를 활용하여 시스템 메모리를 확장하는 SWAP과 MMAP(Memory Mapped) 기반 기술들이 개발되었다[2-4]. SWAP은 메인 메모리(DRAM) 용량보다 더 큰 메모리 공간을 프로세스에 제공해주는 운영 체제 기술이며, 자주 접근하는 데이터를 메인 메모리에 위치하게 하고 자주 접근하지 않는 데이터는 하드디스크 혹은 플래시 SSD와 같은 저장장치에 저장한다.

하지만, 삼성전자의 최신 PCIe 기반 플래시 SSD인 XS1715를 SWAP 디바이스로 이용하여 memcached로 성능을 평가한 결과, DRAM이 충분한 경우와 비교할 때 SWAP 시스템은 8% 정도의 성능만을 보였다. 따라서 플래시 SSD는 접근 지연 시간이 크다는 하드웨어적인 한계로 인해 접근 지역성(spatial locality)이 아주 큰 응용 외에는 SWAP을 위한 디바이스로 사용하기는 어려운 측면이 있다.

차세대 비활성 메모리 분야에서 활발하게 연구 및 개발이 진행되고 있는 MRAM(Magnetoresistive Random-Access Memory)과 PRAM(Phase-change Memory)은 DRAM에 비해 집적도가 높고, 접근 속도는 궁극적으로 DRAM과 비슷해질 것으로 전망된다. 인텔, 삼성과 같은 세계적인 반도체 업체들이 예상하는 PCIe 기반 차세대 저장장치의 특징은 큰 대역폭 외에도 매우 짧은 접근 지연 시간을 보인다. 인텔이 2013년 발표한 자료에 의하면[12], PCIe 플래시 SSD의 경우 4KB의 기준의 읽기 접근 지연 시간이 80us이며 이 중에서 NAND 플래시 메모리에서 4KB의 데이터를 읽는데 걸리는 시간은 50us 정도이다. 그리고 차세대 저장 장치의 경우에는 읽기 접근 시간이 7-9us 정도이며 이 중에서 차세대 메모리 소자에서 4KB의 데이터를 읽는데 걸리는 시간은 1-2us 정도이다.

이러한 고속의 차세대 저장 장치를 이용하여 메모리를 확장하면 메모리를 많이 요구하는 응용의 성능이 좋아질 것이다. 하지만 기존 운영체제의 가상 메모리 관리 오버헤드로 인하여 응용의 성능을 극대화할 수 없다. 본 논문에서는 이러한 오버헤드를 극복하는 방법들을 제안하여 이를 Linux 3.14.3의 SWAP에 구현하였다. SWAP 저장 장치에 쓰기 연산을 할 때 저장 장치의 블록 주소를 결정하는 알고리즘(SWAP 슬롯 할당 알고리즘)을 개선하였고, 차세대 저장 장치의 성능을 최대로 이용하기 위해 시스템 파라미터를 최적화하였다. 평가를 통해 마이크로 벤치마크에서는 기존 시스템 대비 3배, 과학 계산 벤치마크에서는 평균 24%의 성능 향상이 있음을 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구를 소개하고 3장에서는 최적화 기법에 대해서 설명한다. 4장에서는 성능 평가를 설명하고 5장에서는 이 논문의 결론을 제시한다.

## II. 관련 연구

플래시 SSD의 비약적인 발전과 함께 플래시 SSD를

1. Twitter가 확장한 memcached 소프트웨어

이용하여 메모리를 확장하는 많은 연구가 수행되었다. CFLRU 연구에서는 플래시 기반 저장 장치가 쓰기 및 지우기 성능이 읽기 성능보다 떨어지는 특성을 이용하여 SWAP 시스템의 성능을 높이는 방법을 소개하였다[6]. 페이지 회수 시에 쓰기 연산이 발생하지 않는 clean 페이지를 먼저 회수하는 CFLRU 알고리즘을 제시하였다. FlashVM 연구에서는 플래시 SSD에 최적화된 SWAP 시스템을 제안하였다[2]. FlashVM은 zero-page[2]들을 공유하고, clean 페이지를 회수율 조정을 위해 샘플링 기법을 소개하였다. 이 밖에도 플래시 SSD에 제공하는 복수개의 페이지를 한 번에 회수하는 명령어를 사용하여 성능을 개선시켰다. 산업계에서는 플래시 SSD 업체인 FusionIO 사가 플래시 SSD에 최적화된 리눅스 SWAP 커널 모듈인 Fast SWAP을 오픈 소스로 공개하였고[7], Fast SWAP이 Linux 커널에 반영되었다. 위의 연구들은 플래시 SSD에 최적화된 메모리 확장 기술 연구들이며 플래시 SSD보다 접근 지연 시간이 작은 차세대 저장 장치에 문제가 되는 SWAP 슬롯 할당 알고리즘은 최적화하지 않았다.

이러한 메모리 확장 연구들은 주로 운영 체제 수준에서 초점이 맞추어져 있다. SSDAlloc 연구에서는 운영체제 수준이 아니라 응용 프로그램 라이브러리 수준의 메모리 확장 기술을 제안하였다[8]. *ssd_alloc()*이라는 응용 레벨 함수를 제공하여 *ssd_alloc()*을 이용해서 응용 프로그램이 메모리를 할당 받으면 플래시 SSD를 이용해서 메모리가 확장된다. 하지만 SSDAlloc은 *ssd_alloc()*이라는 함수를 응용이 사용해야 하기 때문에 응용 소프트웨어의 수정을 요구한다는 단점이 있다.

본 논문과 같이 플래시 SSD 보다 빠른 고속의 장치에 소프트웨어를 최적화하는 연구도 있다. SCMFS(File System for Storage Class Memory)[9] 및 BPFS(File System for Byte-Addressable NVM)[10] 연구는 차세대 메모리 소자가 메모리 버스에 연결된 시스템에서 파일 시스템의 성능 향상에 관한 연구이다. J. Coburn의 연구에서는 DRAM 기반 SSD와 데이터베이스 시스템 소프트웨어를 상호 최적화하여 성능을 개선시키고자 했다[11]. 이러한 연구들은 메모리 확장 기술이 아닌 파일시스템 및 데이터베이스 연구들이다.

고속의 장치를 이용한 메모리 확장 기술 연구로 [8]과 같은 연구에서는 SWAP을 위한 저장 장치를 고속의 InfiniBand를 이용하여 원격 서버의 DRAM으로 구현하였다. 그러나 이 연구는 하드디스크를 고속의 원격 메모리 기반 저장 장치로 대체하는 수준이고, 운영 체제의 SWAP 슬롯 알고리즘 최적화와 같은 소프트웨어 최적화는 하지 않았다. 본 논문은 고속의 저장 장치를 SWAP을 위한 저장 장치로 사용했을 때 SWAP 슬롯 할당 알고리즘과 같은 SWAP 서브시스템을 최적화하는 기법들을 제안한다.

## III. SWAP 시스템 설계 및 구현

### 3.1 Linux 가상 메모리

이 절에서는 Linux의 가상 메모리 관리 부분에서 I/O와 관련되어 있는 부분을 분석한다. 시스템이 물리적인 DRAM 용량보다 더 많은 메모리를 사용하게 된다면 자주 접근되는 데이터는 메인 메모리에 자주 접근되지 않는 데이터는 저장 장치에 저장되어 관리된다.

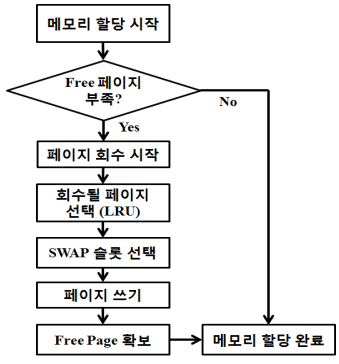


**그림 1. 운영체제의 메모리 할당 방식**

[그림 1]과 같이 운영 체제가 관리하는 free 페이지가 부족한 상황에서 메모리 할당 요청이 오면 운영 체제는 페이지 회수 작업을 시작한다. 회수를 위한 페이지를 선택할 때 LRU(Least Recently Used) 기반의 알고리

2. 0x00으로 채워진 페이지

즘이 사용된다. 그리고 선택된 페이지가 SWAP을 위한 블록 디바이스에 저장될 위치를 (SWAP 슬롯) 선택한다. SWAP 슬롯이 결정되면 해당 위치로 선택된 페이지가 써지고, 그 페이지는 운영 체제가 회수하여 메모리 할당을 완료한다.

### 3.2 SWAP 슬롯 할당 최적화

기존 SWAP 슬롯을 할당하는 알고리즘은 [그림 2]와 같이 선형 찾기 기반이다. SWAP 공간의 크기가 작고 SWAP을 위한 저장 장치가 하드디스크와 같이 접근 지연 시간이 큰 장치라면 시스템에서 선형으로 SWAP 슬롯을 할당하는 비용은 중요하지 않다. 그러나 SWAP 공간의 크기가 크고 저장 장치의 접근 지연 시간이 작다면 SWAP 슬롯을 할당하는 비용은 큰 오버헤드가 된다. 예를 들어 SWAP 공간의 크기가 64 GB라면 SWAP 슬롯은 약 1,600만개 정도 생성되고, 응용이 큰 메모리를 지속적으로 사용한다면 약 1,600만개의 슬롯 중에서 선형으로 탐색해서 할당받는 것은 문제가 된다.

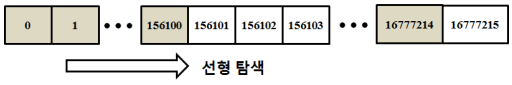


**그림 2. 기존 SWAP 슬롯 탐색 기법**
**(회색: 이미 할당, 흰색 : 할당 가능)**

이것을 확인하기 위해 다음과 같은 실험 환경을 구성하였다. 메인 메모리 8GB와 32GB의 램디스크 기반 SWAP 디바이스를 사용하는 서버 시스템과 (구체적인 사양은 성능 평가 절에서 사용한 서버와 사양이 같음) 같은 사양의 클라이언트 시스템을 이용하였다. 서버 시스템에 24GB 메모리를 사용하는 memcached 응용을 수행하고 클라이언트 시스템에서 YCSB 워크로드를 [13] 수행했을 때 서버 시스템에서 SWAP 슬롯을 할당하는데 최대 50ms 정도 걸리는 것을 관찰하였다. 차세대 저장 장치에 대한 접근 지연 시간이 7-9us이기 때문에 이것은 큰 오버헤드이다.

따라서 이러한 문제를 해결하기 위해 본 논문에서는 SWAP 슬롯을 선형으로 관리하는 것이 아니라 트리 기반 자료 구조로 관리하는 것을 제안한다. 트리 기반 자료 구조로 관리하여 O(n)의 비용이 걸리는 SWAP 슬롯 할당을 O(logn)의 비용으로 SWAP 슬롯을 할당할 수 있다. 구현을 위해 SWAP 슬롯을 위한 트리의 차수를 8로 하였고 트리의 노드는 1비트에 코딩하였다. 그리고 각각의 비트 위치가 SWAP 슬롯을 가리킨다. 이런 방법을 사용하면 64 GB의 SWAP 공간을 위해 4MB 정도의 메모리 공간을 차지하는 트리가 만들어진다. 트리를 위한 메모리 공간은 SWAP 디바이스의 공간이 할당할 때 미리 할당을 한다.

[그림 3]과 같이 말단 노드의 특정 비트가 회색이면 해당 슬롯이 할당되어 있음을 의미하고 중간 노드의 특정 비트가 회색이면 하위 노드가 모두 SWAP 슬롯으로 이미 할당되었음을 의미한다. 그리고 말단 노드의 특정 비트가 흰색이면 해당 슬롯이 사용 가능을 의미하고 중간 노드의 특정 비트가 흰색이면 그 비트를 루트로 하는 트리의 말단 노드 중에서 적어도 한 개가 SWAP 슬롯으로 할당 가능함을 의미한다.

[그림 3]의 예제 같이 트리의 높이가 3이라면 루트 노드부터 말단 노드까지 필요한 메모리 공간은 $73(=8^0+8^1+8^2)$

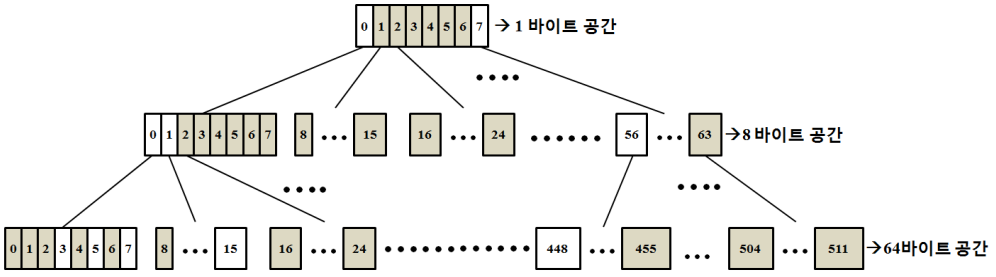


**그림 3. SWAP 탐색을 위한 트리 자료 구조 (높이가 3인 트리의 예).**

**Algorithm 1** Search Procedure for Free SWAP Slot

```
int tree_height : the tree height
char* swap_slot_tree : allocated memory chunk for the tree
int* nth_level_start_byte : offsets for each level

1: int search_free_swap_slot(int last_slot_id) {
2:    int search_level = tree_height;
3:    char slot_node;
4:    int slot_node_id;
5:    int slot_id = last_slot_id;
6:    int found_bit;
7:    int return_id;
8:    while(true) {
9:       if ( search_level == 0 ) return -1; //SWAP Device Full
10:      slot_id = slot_id/8;
11:      slot_node_id = nth_level_start_byte[search_level-1] + slot_id;
12:      slot_node = swap_slot_tree[slot_node_id];
13:      if( slot_node has an available bit ) then break;
14:      search_level--;
15:   }
16:
17:   for( i = search_level ; i ¡= tree_height; i++ ) {
18:      slot_node_id = nth_level_start_byte[search_level-1] + slot_id;
19:      slot_node = swap_slot_tree[slot_node_id];
20:      found_bit = available bit poisition in slot_node;
21:      if ( search_level == tree_height ) break;
22:      slot_id = ( found_bit + slot_id*8 )*8;
23:   }
24:   return_id = slot_id + found_bit;
25:
26:   slot_id = return_id;
27:   search_level = tree_height;
28:   while( search_level =! 1 ) {
29:      slot_id = slot_id / 8;
30:      slot_node_id = nth_level_start_byte[search_level-1] + slot_id;
31:      slot_node = swap_slot_tree[slot_node_id];
32:      if( slot_node does not have available bits ) {
33:         slot_node_id = nth_level_start_byte[search_level-2] + slot_id/8;
34:         slot_node = swap_slot_tree[slot_node_id];
35:         i = slot%8;
36:         mark the ith bit in slot_node as 1;
37:      }
38:      search_level--;
39:   }
40:
41:   return return_id;
42: }
```

**그림 4. 트리 기반 SWAP 슬롯 탐색 알고리즘**

바이트이다. 루트 노드는 1바이트의 공간을 차지하고 루트 노드의 자식 노드들을 8바이트, 그리고 말단 노드들은 64바이트를 차지한다. 73바이트의 공간이 할당되면 루트 노드는 할당된 공간의 첫 번째 바이트, 루트의 자식 노드들을 두 번째부터 아홉 번째 바이트를 그리고 말단 노드들을 열 번째부터 마지막 바이트 공간을 사용한다. 말단 노드가 64바이트 즉 512비트의 공간을 차지하므로 전체 SWAP 슬롯은 512개가 된다. 본 논문에서는 트리의 노드의 회색은 1로 흰색은 0으로 코딩한다. 즉, 말단 노드의 각각의 비트가 0이면 할당 가능하고 1이면 이미 할당했음을 의미한다. 중간 노드의 비트가 0이면 하위 노드에 할당 가능한 비트가 (SWAP 슬롯) 존재함을 의미하고 1이며 하위 노드에 할당 가능한 비트가 없음을 의미한다.

[그림 4]는 본 논문에서 제안하고자 하는 SWAP 슬롯 알고리즘이다. tree_height, swap_slot_tree, nth_level_start_byte는 SWAP을 위한 디바이스 할당될 때 초기화되는 변수들로 트리의 높이, 트리를 위한 메모리 공간, 그리고 트리를 위한 메모리 공간에서 트리의 레벨별 시작 주소를 의미한다.

알고리즘은 크게 세 단계로 구분된다. 첫 번째 단계는 하위 노드에 비어있는 SWAP 슬롯이 있는지 여부를 확인하면서 루트 노드까지 탐색하는 단계이다(8-15줄). 만약 하위 노드에 비어있는 SWAP 슬롯이 있으면 해당 노드에서 탐색을 멈춘다(13줄).

그리고 다시 비어있는 SWAP 슬롯을 찾기 위해 이전 단계에서 탐색한 마지막 노드부터 말단 노드까지 다시 탐색한다(17-23줄). 말단 노드까지 탐색한 후에 할당된 슬롯 번호를 (return_id) 계산한다(24줄). 마지막으로 다시 할당된 노드부터 루트 노드까지 탐색해가면서 (26-39줄) 각 노드에 더 이상 할당할 SWAP 슬롯이 없다면 부모 노드의 해당 비트를 1로 코딩한다(36줄). 이 알고리즘은 Linux 운영 체제에서 SWAP 슬롯을 탐색하는 *scan_swap_map()* 함수에 구현되었다.

예를 들어 말단에서 위치가[3] 489인 슬롯을 가장 최근에 할당하였다면 즉 말단 노드들 중에서 위치가 61인 (489/8의 몫) 바이트에서 위치가 1인(489/8의 나머지) 비트에서 할당이 성공하였다면 탐색은 해당 말단 노드를 저장하는 바이트에서 다시 시작한다. 만약 그 바이트에 비어있는 슬롯이 있다면 비어있는 비트를 (0으로 표시되어 있는 비트) 찾아서 할당한다. 그렇지 않다면 부모 노드를 탐색하기 시작한다. 이 경우에 부모 노드는 레벨 1의 위치가 7인(61/8의 몫) 바이트이다. 부모 노드에서 0인 비트가 있다면 그 비트의 하위 노드에 비어있는 슬롯이 있다는 것이므로 그 위치에서 하위 노드에 대해 탐색을 시작한다. 만약 위치가 4인 비트가 0이라면 말단 레벨의 위치가 480인(=(7*8+4)*8) 바이트에 비어있는 비트가 있다는 것이기 때문에 해당 바이트에서 0으로 표시되어 있는 비트를 찾고 그 비트의 위치를 계산하다 리턴한다. 이러한 방식으로 탐색의 범위를 루트 노드까지 한다. 할당이 끝나면 말단 노드부터 루트

3. 위치는 0부터 시작한다.

노드까지 탐색하면서 해당 바이트에 값이 0인 비트가 있는지 확인하여 없으면 부모 노드의 해당 비트를 1로 해준다.

### 3.3 그 밖의 최적화

본 논문에서는 SWAP 슬롯을 할당하는 알고리즘의 최적화 이외에도 SWAP의 성능을 위해 read-ahead, I/O 스케줄러, swappiness 등을 최적화하였다.

현재 Linux 운영 체제는 페이지 폴트 발생 시에 폴트가 발생한 페이지 외에도 그 페이지와 연속된 최대 8개의 페이지를 읽어온다. 이러한 기법을 read-ahead라고 하며, 고속의 저장 장치에서는 오히려 과도한 I/O를 발생시켜서 성능을 떨어드린다. 따라서 본 논문의 SWAP 시스템에서는 read-ahead 기능을 사용하지 않는다.

고속의 차세대 저장 장치의 응답 속도는 하드디스크보다 짧기 때문에 차세대 저장 장치를 탑재한 시스템의 경우 I/O 스케줄러의 성능상의 이점은 작다. 오히려 I/O 스케줄러가 관리하는 I/O 요청 큐에 여러 쓰레드가 동시에 접근하는 경우가 발생하면 I/O 요청 큐에 혼잡이 발생하여 오히려 성능이 떨어지는 경우도 있다. 따라서 본 논문의 SWAP 시스템은 I/O 스케줄러를 우회하도록 구현되었다. 또한, 메모리를 확장하는 고성능 SWAP 시스템을 위해 최대한 파일에 매핑된 페이지들부터 회수하게 swappiness 값을 0으로 조정하였다.

현재 Linux 운영 체제는 가용한 SWAP 공간의 크기가 전체 SWAP 공간의 50% 아래로 떨어지면 SWAP 슬롯을 회수한다. 즉. 페이지 폴트 시 저장 장치로부터 메인 메모리로 페이지가 로드되면 해당 SWAP 슬롯을 회수하여 SWAP 공간을 확보한다. 그러나 이러한 정책은 clean 페이지들의 경우 페이지 회수 대상이 되었을 때 다시 저장 장치로 써지는 등의 불필요한 I/O를 유발할 수 있다. 본 논문의 SWAP 시스템은 clean 페이지의 경우 SWAP 슬롯을 회수하지 않게 구현되었다.

## IV. 성능 평가

### 4.1 실험 환경

실험을 위해 인텔 Xeon CPU E5630 2.53GHz를 장착한 서버 시스템을 사용하였다. 이 시스템은 8개의 코어와 24GB 메인 메모리를 가지고 있으며 실험을 위해 메인 메모리 크기를 조정하였다. 서론에서 밝힌 것과 같이 차세대 저장 장치는 내부의 메모리 소자에 대한 접근 속도가 4KB 기준으로 1-2us이며, 실험을 위해 이와 유사한 성능을 보이는 DRAM을 미디어로 사용하는 [그림 5]와 같은 DRAM 기반 저장 장치를 사용하였다 [5]. 실험에 사용한 DRAM 기반 저장 장치는 PCIe 인터페이스로 호스트에 연결이 되며 용량은 32GB이다. 최대 데이터 전송량은 초당 읽기 1.6GB, 쓰기 1.4GB이고, 접근 지연 시간은 4KB 기준으로 읽기 연산은 5us, 쓰기 연산은 7us이다.

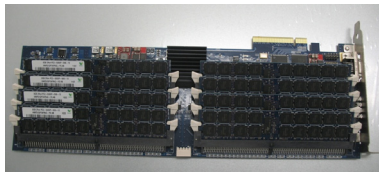

**그림 5. 실험에 사용된 DRAM 기반 SSD**

### 4.2 실험 결과 및 분석

초고속 저장 장치에 최적화된 SWAP 시스템을 평가하기 위해 synthetic 벤치마크와 과학 계산 벤치마크를 사용하였다. Synthetic 벤치마크 응용은 6GB 메모리를 할당하여 할당한 메모리를 세 번 순차/랜덤 읽기, 순차/랜덤 쓰기 연산을 수행한다. 과학 계산 벤치마크는 NPB를 사용하였고, 본 논문에서는 NPB의 ft와 ua 응용을 수행하였다. ft는 Fast Fourier Transformation (FFT)를 x, y, z 축으로 연속하여 수행하는 응용이다. ua는 대류 및 확산을 동반하는 열 방정식을 풀어내는 응용이다. 최적화된 SWAP은 본론에서 설명한 최적화 방법을 모두 구현한 Linux 3.1.14 기반 시스템이며 기존 SWAP은 수정을 하지 않은 Linux 3.1.14 시스템을 의미하며 실험값은 모두 수행 시간(초)이다.

[표 1]은 synthetic 벤치마크 수행 결과이다. Synthetic 벤치마크 응용이 6GB 메모리를 사용하기 때문에 실험을 위해 메인 메모리의 크기를 4GB로 제한하

여 강제로 SWAP을 사용하게 설정을 하였다. 즉 6GB의 응용이 할당한 메모리 공간의 일부는 메인 메모리에 다른 일부는 저장 장치에 위치해 있다. [표 1]과 같이 synthetic 벤치마크의 경우 최적화된 SWAP 시스템은 기존의 SWAP 시스템에 비해 약 3배의 성능 향상이 있음을 확인할 수 있었다.

**표 1. Synthetic 벤치마크 결과**

| | 기존 SWAP | 최적화된 SWAP |
|---|---|---|
| 순차 읽기 | 336초 | 130초 |
| 랜덤 읽기 | 387초 | 147초 |
| 순차 쓰기 | 410초 | 151초 |
| 랜덤 쓰기 | 431초 | 181초 |

[표 2]는 NPB 벤치마크의 수행 결과이다. ft 응용이 약 1.4GB의 메모리를 사용하기 때문에 SWAP을 사용하는 환경을 위해 메인 메모리를 1GB만 사용하게 시스템을 설정하였다. ua 응용의 경우에는 8GB 메모리를 사용하기 때문에 메인 메모리의 크기를 6GB로 제한하였다.

**표 2. NPB 벤치마크 결과**

| | 기존 SWAP | 최적화된 SWAP |
|---|---|---|
| NPB ft | 1029초 | 813초 |
| NPB ua | 46589초 | 33737초 |

NPB 벤치마크의 ft 응용의 경우에, 최적화된 SWAP 시스템은 ft를 수행하는데 813초 그리고 기존 SWAP 시스템은 1029초 정도가 걸렸고, 최적화된 시스템이 21% 정도 성능 향상 효과가 있었다. NPB 벤치마크의 ua 응용의 경우에, 최적화된 SWAP 시스템은 약 33700여초 그리고 기존 SWAP 시스템은 46500여초 정도가 걸렸고, 최적화된 시스템이 28% 정도 성능 향상 효과가 있었다.

## V. 결론

본 논문은 향후 DRAM의 공정 기술 한계와 이를 극복하기 위해 고속의 차세대 저장 장치를 이용하여 메모리를 확장하는 시스템을 제안하였다. 운영 체제에는 저장 장치를 활용하여 가용 메모리를 확장해주는 가상 메모리 관리 시스템이 있다. 그러나 기존의 가상 메모리 시스템은 응답 속도가 플래시 SSD와 같은 기존의 저장 장치보다 빠른 차세대 메모리 기반 저장 장치에 최적화되지 않았다. 본 논문은 Linux 운영 체제의 가상 메모리 시스템을 차세대 저장 장치에 최적화하는 방법들을 제안하였고, 응답 속도가 빠른 DRAM 기반 저장 장치에서 Linux SWAP 시스템의 성능을 향상시킬 수 있음을 보였다.

향후에는 가상 메모리 시스템을 사용하는 다른 인터페이스인 메모리 사상 기반 I/O를 차세대 저장 장치에 최적화하는 연구를 진행하려고 한다.

## 참 고 문 헌

[1] http://memcached.org/

[2] M. Saxena and M. M. Swift, “Flashvm: Virtual memory management on flash,” In Proceedings of the 2010 USENIX conference on USENIX Annual Technical Conference (USENIXATC’10), 2010.

[3] A. Badam and V. S. Pai, “Ssdalloc: hybrid ssd/ram memory management made easy,” In Proceedings of the 2011 USENIX conference on Networked systems design and implementation (NSDI’11), 2011.

[4] B. Van Essen, H. Hsieh, S. Ames, and M. Gokhale, “DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications,” In Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis (SCC’12), 2012.

[5] http://www.taejin.co.kr/solutions/jetsp eed

[6] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, “CFLRU: a replacement algorithm for flash

memory," In Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES'06), 2006.

[7] http://opennvm.github.io/nvm-fast-swap-documents/ #extended memory overview.html

[8] S. Liang, R. Noronha, and D. K. Panda, "Swapping to remote memory over InfiniBand: An Approach using a High Performance Network Block Device," In Proceedings of the 2005 IEEE Cluster Computing (Cluster'05), 2005.

[9] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'11), 2011.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," In Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09), 2009.

[11] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13), 2013.

[12] A. Huffman, "NVM Express Overview & Ecosystem Update," In Proceedings of the 2013 Flash Memory Summit, 2013.

[13] https://github.com/brianfrankcooper/YCSB/

[14] 손인국, 류은경, 박준호, 복경수, 유재수, "맵-리듀스의 처리 속도 향상을 위한 데이터 접근 패턴에 따른 핫-데이터 복제 기법", 한국콘텐츠학회논문지, 제13권, 제11호, pp.21-27, 2013(11).

[15] 김석규, 김직수, 김상완, 노승우, 김서영, 황순욱, "슈퍼컴퓨팅환경에서의 대규모 계산 작업 처리 기술 연구", 한국콘텐츠학회논문지, 제14권, 제5호, pp.8-17, 2014(5).

## 저 자 소 개

**한 혁(Hyuck Han)** 정회원

- 2003년 8월 : 서울대학교 컴퓨터공학부(공학사)
- 2006년 2월 : 서울대학교 컴퓨터공학부(공학석사)
- 2011년 2월 : 서울대학교 컴퓨터공학부(공학박사)
- 2011년 3월 ~ 2012년 8월 : 서울대학교 컴퓨터공학부 박사후 연구원
- 2012년 9월 ~ 2014년 2월 : 삼성전자 메모리 사업부 책임연구원
- 2014년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수

<관심분야> : 데이터베이스 시스템, 병렬 프로그래밍, 분산 시스템