Journal of Korea Multimedia Society Vol. 13, No. 9, September 2010(pp. 1272-1284)

모바일 컴퓨터를 위한 플래시 메모리 스왑 시스템

전선수[†], 류연승[††]

요 약

모바일 컴퓨터가 고성능화되고 범용 컴퓨터처럼 사용되면서 모바일 컴퓨터의 운영체제에서도 주 기억장치를 효율적으로 사용할 수 있게 해주는 스왑 시스템 기능이 요구되고 있다. 모바일 컴퓨터의 저장 장치는 플래시 메모리가 널리 쓰이고 있는데 현재의 리눅스 스왑 시스템은 플래시 메모리를 고려하지 않고 있다. 스왑 시스템은 실행 중인 프로세스의 내용을 저장하기 때문에 프로세스 실행과 밀접한 관련이 있다. 이러한 성질을 고려하여, 본 논문에서는 프로세스 별로 플래시 메모리 블록을 할당하는 PASS(Process-Aware Swap System)라는 새로운 리눅스 스왑 시스템을 연구하였다. 트레이스 기반의 실험을 통해 PASS의 가비지 수집 성능이 기존 가비지 수집 기법을 사용하는 리눅스 스왑 시스템보다 우수함을 보였다.

A Flash Memory Swap System for Mobile Computers

Seonsu Jeon[†], Yeonseung Ryu[††]

ABSTRACT

As the mobile computers are becoming powerful and are used like general-purpose computers, operating systems for mobile computers also require swap system functionality that utilizes main memory efficiently. Flash memory is widely used as storage device for mobile computers but current linux swap system does not consider flash memory. Swap system is tightly related with process execution since it stores the contents of process in execution. By taking advantage of this characteristics, in this paper, we study a new linux swap system called PASS(Process-Aware Swap System), which allocates the different flash memory blocks to each process. Trace-driven experimental results show that PASS outperforms existing linux swap system with existing garbage collection schemes in terms of garbage collection cost.

Key words: Flash Memory(플래시 메모리), Swap System(스왑 시스템), Virtual Memory(가상 메모리), Mobile Computer(모바일 컴퓨터)

1. 서 론

최근 스마트 폰과 같은 모바일 컴퓨터들이 고성능화되면서 기존의 PC처럼 다양한 응응 프로그램을 수행하는 범용 컴퓨터 시스템으로서 사용되고 있다. 이에 따라 모바일 컴퓨터의 운영체제에서도 주 기억장치와 보조 기억장치를 효율적으로 관리하기 위한 가상 메모리 시스템과 스왑 시스템의 사용이 요구되고 있다.

운영체제의 가상 메모리(virtual memory)는 응용 프로그램이 수행될 때 보조 기억장치에 있는 프로그램 파일에서 필요한 부분만 주기억장치로 적재시키

※ 교신저자(Corresponding Author) : 류연승, 주소 : 경기도 용인시 처인구 남동 명지대학교 컴퓨터공학과(449-728), 전화 : 031)330-6781, FAX : 031)330-6967, E-mail : ysryu@mju.ac.kr
접수일 : 2010년 1월 18일, 수정일 : 2010년 5월 13일
완료일 : 2010년 6월 29일

[†] 정회원, (주)팬택 중앙연구소 연구원
(E-mail : junplayer@hanmail.net)
[††] 종신회원, 명지대학교 컴퓨터공학과 부교수
※ 이 논문은 2007년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구임(KRF-2007-313-D00637).

는 메모리 관리 기법이다[1]. 가상 메모리 기법 중에서 페이징(paging) 기법은 주 기억장치를 페이지 프레임으로 나누고 프로그램을 페이지 프레임과 같은 크기인 페이지 단위로 나누어 적재하는 기법으로서 대부분의 운영체제에서 사용되고 있다. 응용 프로그램이 수행되면 프로세스가 되며 프로세스 이미지 중에서 코드와 데이터는 보조기억장치에 있는 파일로부터 읽어서 주 기억장치에 적재되고, 스택과 힙은 동적으로 주 기억장치에서 생성된다. 주 기억장치의 가용 페이지 프레임들이 정해진 한도보다 적어지면 가상 메모리 시스템은 가용 페이지 프레임을 만들기 위해서 페이지 재생(reclamation) 작업을 수행한다. 페이지 재생 작업은 주 기억장치에서 재생할 페이지 프레임을 선택하고 선택된 페이지 프레임의 내용을 원래 위치로 저장해야 한다. 선택된 페이지 프레임의 내용이 프로세스의 데이터라면 보조 기억 장치의 원래 파일에 저장하면 되고, 프로세스의 스택과 힙이라면 원래 파일이 없으므로 보조 기억장치의 임시 영역에 이를 저장하게 된다. 이때, 보조 기억장치의 임시 영역을 스왑 영역(swap area)이라 하고 스왑 영역에 저장하는 연산을 스왑 아웃(swap-out), 스왑 영역에서 읽어 가는 연산을 스왑 인(swap-in)이라 한다.

모바일 컴퓨터의 보조 기억 장치로서 NAND 플래시 메모리가 널리 사용되고 있다. 플래시 메모리는 빠른 연산 속도, 저 전력소비, 강한 충격 저항 등의 장점을 가지고 있어 하드 디스크를 대체할 저장 장치로 부각되고 있다. 표 1은 플래시 메모리와 다른 저장 매체의 연산 속도를 비교하여 보여주고 있다. 이와 같이 플래시 메모리가 많은 장점을 가지고 있지만, 다음과 같은 몇가지 특별한 하드웨어적 특징을 가지고 있다[2].

첫째, 플래시 메모리는 블록들의 배열로 구성되며, 각 블록은 고정된 개수의 페이지[1]로 구성된다. 플래시 메모리에 대한 기본 명령어는 읽기, 쓰기, 삭제(erase)인데 읽기와 쓰기의 기본 단위는 페이지이며, 삭제는 블록 단위로 수행된다.

둘째, 플래시 메모리는 저장되어 있는 데이터를 바로 변경할 수 없으며 데이터가 저장되어 있는 블록에 대해 삭제 연산을 한 후에 변경할 내용을 쓸 수 있다. 그런데, 표 1에서 보는 바와 같이 삭제 연산은 읽기와 쓰기 연산에 비해 상대적으로 많은 시간이 소요된다.

표 1. 저장 매체의 연산 비교

저장 매체	Access time		
	읽기(512B)	쓰기(512B)	삭제
DRAM	2.56 μs	2.56 μs	해당없음
NAND 플래시[2]	25 μs	200 μs	1.5 ms
디스크	12.4 ms	12.4 ms	해당없음

셋째, 한 블록에 대한 삭제 연산의 횟수가 제한되어있다(10,000번~100,000번). 만약 특정한 블록에 삭제 연산이 집중되면 해당 블록은 쓰이지 못하게 되며 플래시 메모리 장치의 수명에 영향을 미치게 된다. 이와 같은 특정 블록의 마모를 피하기 위해 삭제 연산이 모든 블록에 고르게 수행되는 것이 좋다. 이를 마모도 평준화(wear leveling)라고 부른다.

플래시 메모리 상의 데이터의 변경이 요구될 때 데이터의 원래 위치에 변경할 내용을 저장하는 원 위치 변경(in-place update) 방식은 블록의 삭제를 요구하기 때문에 쓰기 연산의 성능을 저하시킨다. 이러한 문제점을 해결하기 위해, 변경할 데이터를 원 위치가 아닌 다른 위치에 저장하고 위치 사상 테이블(address mapping table)을 두어 관리하는 타 위치 변경(out-of-place update) 방식이 널리 쓰이고 있다. 이때, 원래 위치에 있던 페이지는 무효 페이지(invalid page)로 표시된다. 타 위치 변경 방식에서는 이전 데이터가 있던 블록에 무효 페이지들이 남게 되므로 이를 다시 가용 공간으로 재생하기 위한 가비지 수집(garbage collection) 작업을 수행해야 한다. 가비지 수집 작업을 할 때, 삭제 대상 블록에서 아직 유효한 페이지들이 있다면 이들을 타 위치로 복사시킨 후 블록을 삭제해야 한다. 가비지 수집 알고리즘은 어떤 블록을 재생할 지, 재생되는 블록 내의 유효한 데이터를 어디로 옮길 것인지와 같은 문제를 다룬다. 이때, 가비지 수집 알고리즘은 수행되는 삭제 연산의 수와 데이터 복사 수와 같은 가비지 수집의 비용을 줄여야 한다.

본 논문에서는 플래시 메모리를 저장 장치로 사용하는 모바일 컴퓨터에서 PASS(Process-Aware Swap System)라고 부르는 새로운 리눅스 스왑 시스템을 연구하였다. 모바일 컴퓨터의 리눅스는 플래시 메모

1) 가상 메모리의 페이지와 플래시 메모리의 페이지가 혼동이 되고 있으나 관례에 따라 그대로 사용하되 가능한 혼동되지 않도록 주의하였다.

리를 직접 제어하기 때문에 스왑 시스템에서 플래시 메모리 블록의 할당 및 가비지 수집을 다루어야 한다. 기존 리눅스 스왑 시스템은 디스크의 탐색 시간을 최소화하기 위해 최적화되어 있으나 플래시 메모리에서는 탐색이 요구되지 않는다. 또한, 스왑 시스템은 프로세스의 실행 중에 사용되는 페이지들을 저장하기 때문에 프로세스 실행과 밀접한 관련이 있다. 이러한 성질을 이용하여 제안한 PASS는 같은 프로세스에 속한 페이지들을 같은 블록에 저장시킴으로써 가비지 수집 비용을 줄인다. 제안한 PASS의 성능을 검증하기 위해 트레이스 기반의 시뮬레이션 실험을 수행하였으며 기존 리눅스 스왑 시스템보다 가비지 수집의 성능이 우수함을 보였다.

본 논문의 구성은 다음과 같다. 2장에서는 리눅스 스왑 시스템과 플래시 메모리 저장 시스템에서 요구되는 가비지 수집에 대해 설명하고, 현재 스왑 시스템의 문제점을 살펴본다. 3장에서는 제안하는 PASS의 구조, 스왑 연산 및 가비지 수집 알고리즘을 기술한다. 4장에서는 제안한 기법의 성능을 검증하기 위해 수행한 실험 결과를 기술한다. 이 논문의 결론은 5장에서 맺는다.

2. 관련 연구

2.1 리눅스 스왑 시스템

그림 1은 리눅스의 스왑 시스템 및 관련 커널 데이터 구조를 간략하게 도시하여 보여주고 있다[1].

리눅스는 요구 페이징(demand paging) 방식의 가상 메모리 시스템을 사용한다. 프로세스의 주소 변환을 위하여 페이지 테이블(page table)을 사용한다. 프로세스를 관리하기 위하여 각 프로세스마다 task 구조체를 둔다. 또한, 프로세스 별로 프로세스의 페이지들이 어디에 있는 지를 관리하기 위하여 mm 구조체를 둔다. task 구조체의 필드 중에서 mm 필드는 프로세스의 mm 구조체의 주소를 가지며, mm 구조체의 pgd 필드는 페이지 테이블의 주소를 가진다. 페이지 테이블의 각 엔트리는 해당 페이지의 위치 정보를 가지고 있다.

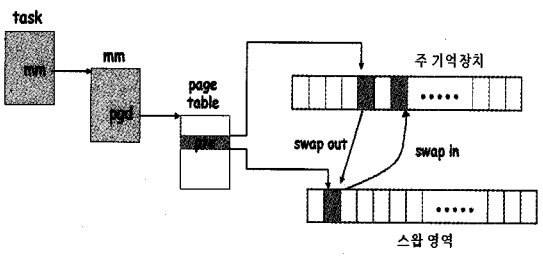
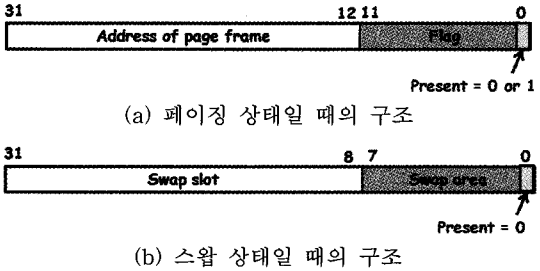


그림 1. 리눅스 스왑 시스템 및 관련 데이터 구조

페이지 테이블의 엔트리는 그림 2의 (a)와 같이 크게 두 부분으로 이루어져 있다. 하위 12비트는 flag이고 나머지 30비트는 페이지 프레임의 주소이다. flag에서 첫 번째 비트인 Present가 1이고 페이지 프레임 주소 값이 0이 아니면 주 기억장치의 해당 페이지 프레임에 적재되어 있음을 나타낸다. Present가 0인 경우에는 두 경우로 나뉜다. Present가 0일 때, 상위 30비트가 0이면 해당 페이지가 주기억장치에 부재함을 나타내고, 상위 30비트가 0이 아니면 해당 페이지가 스왑 영역(swap area)에 저장되어 있음을 나타낸다. 페이지가 스왑 영역에 저장되면 페이지 테이블 엔트리는 스왑 아웃된 페이지의 위치 정보를 관리하기 위해 사용된다. 이때 페이지 테이블 엔트리의 구조는 그림 2의 (b)에 보이고 있다. 그림에서 보이는 스왑 영역과 스왑 슬롯(swap slot)은 뒤에서 설명하기로 한다.

리눅스는 주 기억장치의 페이지 프레임들을 LRU (Least Recently Used) 리스트로 관리한다. 주 기억장치의 사용 중인 페이지 프레임의 수가 정해진 한도를 넘게 되면 가용 프레임(free frame)을 확보하기 위하여 페이지 재생(reclamation)을 수행한다. 이때, LRU 리스트에서 가장 최근에 참조되지 않은 페이지 프레임을 선택하여 가용 프레임으로 만든다. 만약 재생되는 페이지 프레임 내용이 프로세스의 데이터라면 디스크에 있는 원래 위치에 저장해준다. 재생되는 페이지 프레임 내용이 프로세스의 스택, 힙, 공유 메모리인 경우라면 스왑 영역으로 스왑 아웃(swap-out)한다. 스왑 아웃된 페이지는 나중에 다시 참조될



(a) 페이징 상태일 때의 구조

(b) 스왑 상태일 때의 구조

그림 2. 리눅스 페이지 테이블 엔트리의 구조

때 주 기억장치로 스왑 인(swap-in) 된다.

스왑 영역은 스왑 아웃된 페이지를 저장하는 보조 기억장치 내의 공간이다. 스왑 영역은 독자적인 디스크 파티션 또는 하나의 파일로 설정이 가능하고 여러 개의 스왑 영역을 설정할 수 있다. 스왑 영역은 여러 개의 스왑 슬롯(slot)으로 구성되어 있고 한 슬롯은 한 페이지를 저장할 수 있다. 한 개의 슬롯의 크기는 4KB이며, 페이지 프레임의 크기와 같다. 스왑 영역을 관리하기 위하여 swap_map 배열을 사용한다. swap_map 배열은 각 슬롯의 상태를 저장하는데, 값이 0이면 유휴 슬롯이고 값이 양수이면 스왑 아웃된 페이지가 저장되어 있으며 해당 페이지를 공유하는 프로세스의 수를 나타낸다.

스왑 아웃되는 페이지들은 스왑 영역에 순차적으로 저장된다. 주 기억장치의 가용 페이지 프레임이 부족하게 되면 커널은 일시적으로 수십 개의 페이지들을 스왑 아웃하게 되는데 디스크 헤드의 움직임을 최소화하기 위하여 스왑 아웃되는 페이지들을 순차적으로 저장한다. 스왑 인은 프로세스가 페이지를 다시 참조하면 수행되는데, 디스크 헤드 움직임을 줄이기 위하여 최대 8개의 연속된 슬롯의 페이지들을 선반입(read-ahead)한다. 스왑 아웃은 순차적으로 슬롯에 저장해 가고 스왑 인은 임의의 슬롯에서 발생하므로 스왑 아웃과 스왑 인의 수행이 혼합되어 수행하면서 디스크 헤드 움직임이 커질 수 있다. 이러한 디스크 헤드 움직임을 고려하여 리눅스 커널은 256개의 스왑 아웃된 페이지를 저장하고 나면, 앞부분의 빈 슬롯부터 페이지를 저장한다.

2.2 플래시 메모리 저장 시스템의 가비지 수집

플래시 메모리는 저장되어 있는 데이터를 변경(overwrite)하려면 먼저 삭제 연산을 수행하여야 한다. 플래시 메모리의 한 페이지가 변경될 때 마다 페이지가 속한 블록을 삭제하고 블록 내의 유효 페이지(valid page)들을 다른 블록에 복사해야 한다면, 쓰기 성능은 매우 좋지 않게 된다. 이러한 문제를 해결하기 위하여 대부분의 플래시 메모리 저장 시스템에서는 타 위치 변경(out-of-place update) 방식을 사용한다. 타 위치 변경 방식은 변경되는 데이터를 다른 빈 페이지에 저장시키고 이를 사상 테이블(mapping table)로 관리하는 방식으로서 쓰기 성능을 좋게 해준다.

그러나, 타 위치 변경 방식은 가비지 수집을 필요로 한다. 가비지 수집은 이전 데이터가 있는 페이지들을 재사용하기 위하여 해당 블록 내에 유효 페이지가 있다면 이들을 복사한 후 해당 블록을 삭제하는 작업이다. 일반적으로 가비지 수집의 수행은 세 단계로 수행된다. 첫째, 삭제할 희생 블록(victim block)을 정한다. 둘째, 희생 블록에서 유효(valid) 페이지가 있다면 다른 블록에 복사한다. 셋째, 희생 블록에 삭제 연산을 수행한다.

가비지 수집은 일반 입출력 연산과는 별도로 발생하는 것이므로 그 수행 비용을 최소화하는 것이 필요하다. 특정 시간 동안에 수행한 가비지 수집의 비용은 식 (1)과 같이 정의된다.

$$GC\_cost = N_E * T_E + N_C * T_C \quad (1)$$

식 (1)에서 N_E는 희생 블록의 삭제 횟수, T_E는 한 삭제 연산에 소요되는 시간, N_C는 희생 블록에서 가용 공간으로 복사한 유효 페이지의 복사 수, T_C는 한 복사 연산에 소요되는 시간으로 정의한다.

희생 블록에서 유효 페이지의 복사 수가 적을수록 비용도 적으며 가비지 수집의 결과로 확보하게 되는 가용 공간의 크기는 커진다. 예를 들어, 10개 페이지로 이루어진 블록 A가 유효 페이지가 5개, 무효 페이지가 5개이고, 블록 B는 유효 페이지가 3개, 무효 페이지가 7개라고 하자. 가비지 수집기가 블록 A를 선택하게 되면 복사 수는 5가 되며 확보하게 되는 가용 페이지는 5개이다. 그러나, 블록 B를 선택하게 되면 복사 비용은 3이고 확보하는 가용 페이지는 7개이다. 따라서, 블록 B를 선택하면 비용도 적게 되며 확보하는 가용 공간의 크기는 커진다.

이와 같이 재생할 블록의 선택에 따라 확보하는 가용 공간의 크기가 달라지게 되는데 가용 공간을 많이 확보할수록 어떤 주어진 시간 동안의 가비지 수집 수행 횟수는 적어지게 되며 결과적으로 삭제 비용도 적어지게 된다. 연산 시간과 전력 소비를 고려할 때 한 블록의 삭제 비용은 한 블록내의 페이지들의 이동 비용보다 크다. 따라서, 가비지 수집에서는 가용 공간을 가능한 많이 확보하여 삭제 연산의 회수를 최소화하는 것이 중요하다.

우수한 가비지 수집 알고리즘은 효율적인 희생 블록 선택(victim block selection) 알고리즘과 유효 페이지 복사(valid page copy) 알고리즘을 가져야 한다.

대부분의 희생 블록 선택 알고리즘들은 블록의 이

용률(utilization)을 활용한다[3-5]. 블록의 이용률이란 블록 내의 유효 페이지가 차지하는 비율을 뜻한다. 대표적인 greedy 알고리즘은 이용률이 가장 적은 블록(즉, 유효 페이지가 가장 적은 블록)을 선택하는 알고리즘이다[4,5]. cost-benefit 알고리즘은 각 블록에 대해 식 (2)의 값을 계산하고 가장 큰 값을 갖는 블록을 선택한다[3,5].

$$\frac{age \cdot (1-u)}{1+u} \tag{2}$$

u는 블록의 이용률이고 age는 마지막 변경 이후의 경과 시간이다. cost-benefit 알고리즘은 블록의 이용률이 적고 최근에 변경된 블록을 선호한다.

가비지 수집의 성능은 재생 블록의 선택 정책 뿐만 아니라 유효 페이지 복사 정책에 크게 좌우된다. 대부분의 효율적인 유효 페이지 복사 알고리즘은 파일 시스템의 지역성(locality) 특징을 이용한다[6-9]. 지역성 특징이란 파일 시스템에서 특정 영역에 특정 입출력 접근이 집중되는 특징을 말한다. 예를 들면, 파일 시스템의 메타 데이터는 빈번한 변경 작업이 집중되는 특징을 보인다. 이와 같이 빈번한 변경 작업이 발생하는 데이터를 핫(hot) 데이터라 하고, 변경 작업이 적은 데이터를 콜드(cold) 데이터라 정의한다. 핫 데이터의 특징을 활용하면 가비지 수집 비용을 줄일 수 있다.

그림 3을 사용하여 예를 들어 본다. 그림에서 플래시 메모리는 8개의 블록으로 구성되고 각 블록은 4개의 페이지를 가진다고 가정하였다. 블록 1과 블록 2가 희생 블록으로 선택되었고, 이 블록들을 삭제하기 전에 블록 내에 있는 유효 페이지들을 블록 5와 블록 6에 복사하고 있다. 이때, 그림 3의 (a)는 블록 1과 블록 2의 유효 페이지들을 차례대로 복사하였고, (b)는 유효 페이지 중에서 핫 페이지와 콜드 페이지를 구별하여 다른 블록에 복사하였다. (a)의 경우, 핫 페이지와 콜드 페이지가 한 블록에 섞여 있다. 핫 페이지는 곧 변경되기 때문에 무효 페이지로 바뀔 가능성이 높다. 따라서, (b)의 경우처럼 한 블록에 핫 페이지만 모아 놓게 되면 해당 블록은 곧 무효 페이지만 가지게 될 가능성이 높고 나중에 가비지 수집에서 이 블록을 선택하여 삭제하게 되면 복사 비용이 없으므로 가비지 수집 비용이 최소화된다. 핫 데이터의 관리와 같이 효율적인 저장 공간의 관리 정책을 고려한다면 가비지 수집의 성능을 향상시킬 수 있다.

2.3 기존 연구 및 문제점

플래시 메모리 저장장치를 고려하는 가상 메모리 기법에 대한 많은 연구가 있어왔다[8-14]. 그러나, 대부분의 연구는 가상 메모리의 페이지 교체 알고리즘에 대한 연구이며 모바일 컴퓨터를 위한 리눅스 플래시 메모리 스왑 시스템에 대한 연구는 매우 적다[15,16].

[15]는 리눅스를 위한 플래시 메모리 스왑 시스템

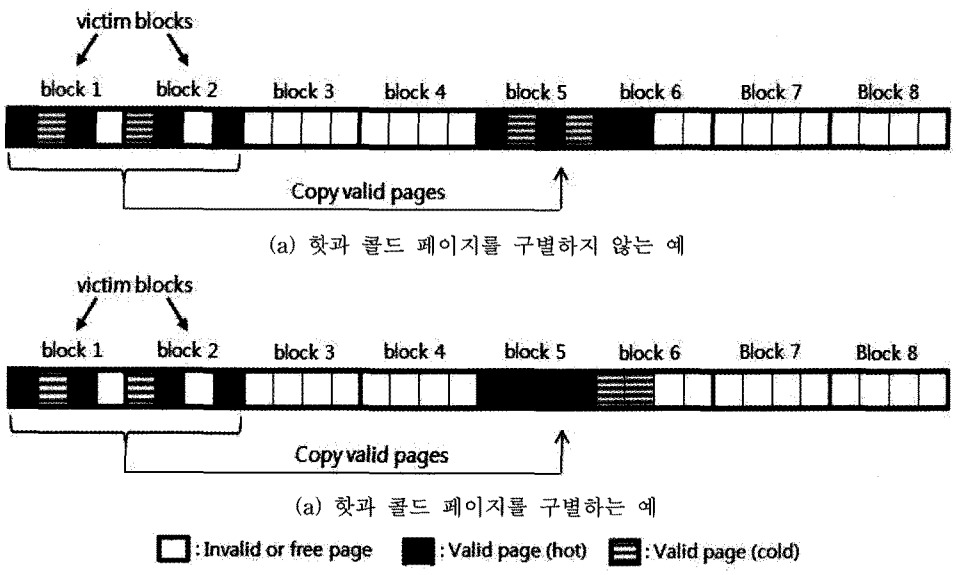


그림 3. 유효 페이지의 복사의 예

을 처음으로 연구하였다. 리눅스 커널 2.4의 스왑 시스템에서 불필요한 FTL (Flash Translation Layer) 코드를 제거하고 공유 페이지를 관리하기 위하여 swap_map 배열과 해시 테이블을 활용하는 FASS (Flash Aware Swap System)를 제안하였다. FASS는 커널 2.4 스왑 시스템을 그대로 사용하면서 최적화하는 데 목적이 있었고 가비지 수집 알고리즘으로는 cost-benefit 기법을 사용하였다.

[16]에서는 리눅스 스왑 시스템에서 스왑 인을 할 때 선반입(read-ahead)되는 페이지들이 두 개의 플래시 메모리 블록에 걸치는 경우를 방지하고 블록 내부의 페이지들만 선반입해주는 LOBI(Log-structured swap-Out, Block-aligned swap-In) 기법을 제안하였다. LOBI 기법은 가비지 수집 알고리즘으로서 기존의 greedy 및 cost-benefit 기법 등을 사용하였다.

원래의 리눅스 스왑 시스템은 디스크 탐색 시간(seek time)을 고려하여 설계되었다. 그러나, 플래시 메모리에서는 탐색 시간이 없으므로 스왑 아웃되는 페이지들을 연속된 슬롯에 저장할 필요가 없으며 스왑 인할 때 여러 개 페이지를 선반입할 필요가 없다. [15,16]의 연구는 기존 리눅스 스왑 시스템의 방식을 그대로 사용하면서 일부 기능을 추가하는 데 그쳤으며 가비지 수집 기법도 기존의 알고리즘을 적용하여 가비지 수집 비용을 줄이기에는 미흡하다.

스왑 시스템의 저장 장치의 접근은 프로세스의 실행과 매우 큰 관련성을 가지고 있다. 주 기억장치에 적재된 프로세스의 페이지가 오래동안 참조되지 않으면 스왑 아웃 대상이 되어 스왑 저장 장치에 써주고, 스왑 아웃된 페이지를 재참조하게 되면 스왑 저장 장치의 해당 내용을 읽어준다. 스왑 인이 수행되면 해당 페이지가 저장되어 있던 스왑 슬롯은 이제 가용 슬롯이 되어 다른 스왑 아웃되는 페이지를 저장하기 위해 사용된다. 또한, 프로세스가 종료되면 프로세스의 스왑 아웃된 모든 페이지들이 필요없게 된다.

이와 같이, 스왑 시스템은 스왑 인 또는 프로세스 종료에 의해 스왑 저장 장치에 저장된 내용이 필요없게 되어 해당 스왑 슬롯들이 가용 슬롯이 된다. 스왑 저장 장치로서 플래시 메모리를 사용하는 경우, 가용 슬롯을 재사용하려면 가용 슬롯을 포함하는 플래시 메모리 블록을 먼저 삭제해야만 한다. 그러나, 스왑 인을 할 때마다 바로 해당 플래시 메모리 블록을 삭제한다면 성능이 저하되므로 스왑 슬롯을 무효로 표시하고 나중에 가비지 수집을 하는 것이 필요하다.

그림 4는 두 프로세스 A와 B의 페이지가 스왑 아웃된 예를 보여주고 있다. 그림에서 스왑 영역은 5개의 플래시 메모리 블록으로 되어 있고 각 블록은 4개의 스왑 슬롯에 해당하는 플래시 메모리 페이지를 가지고 있다. 프로세스 A의 경우, 두 개 페이지 1과 2가 블록 1과 블록 2에 걸쳐서 스왑 아웃되었고, 시간이 지나 두 개 페이지 3과 4가 블록 2와 블록 3에 걸쳐서 스왑 아웃되었다. 프로세스 A가 스왑 아웃된 페이지 1, 2, 3, 4를 다시 참조하여 스왑 인하거나 프로세스 A가 종료하면 네 개의 슬롯은 가용 슬롯이 된다. 이 가용 슬롯들에 다른 프로세스의 페이지를 저장하려면 블록 1부터 블록 3의 세 개의 블록을 삭제해야 하고 삭제하기 이전에 블록 내 유효한 슬롯(예: 프로세스 B의 페이지 5, 6, 7, 8)을 다른 블록으로 복사하여야 한다.

스왑 인을 할 때마다 이와 같은 블록의 삭제 및 유효 슬롯의 복사 작업을 수행한다면 스왑 인의 성능이 저하되므로, 파일 시스템의 경우처럼 스왑 인된 슬롯을 무효 상태로 표시한 후 나중에 가비지 수집을 하는 것이 필요하다.

그림 5는 프로세스 A와 B의 스왑 아웃된 페이지가 별도의 블록 1과 블록 2에 분리되어 저장된 경우를 보여주고 있다. 이때, 프로세스 A의 페이지 1, 2, 3, 4가 스왑 인 되거나 프로세스 A가 종료하는 경우, 블록 1의 슬롯들이 가용 슬롯이 된다. 이 네 개의 슬롯을 재사용하기 위하여 블록 1만 삭제하면 되고 유효 슬롯의 복사는 필요 없다. 따라서, 가비지 수집 비용이 최소화되고 성능이 향상된다. 이와 같이, 스왑 시스템의 경우 스왑 저장 장치의 접근이 프로세스의 실행과 밀접한 관련이 있으므로 프로세스에 기반

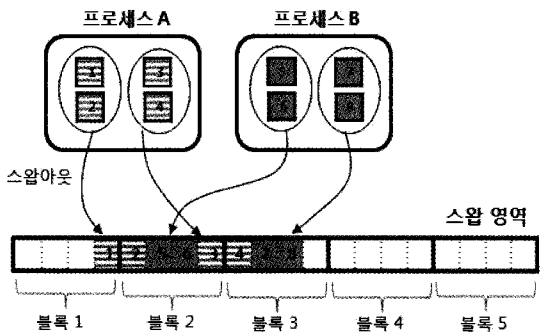


그림 4. 플래시 메모리 스왑 시스템의 예

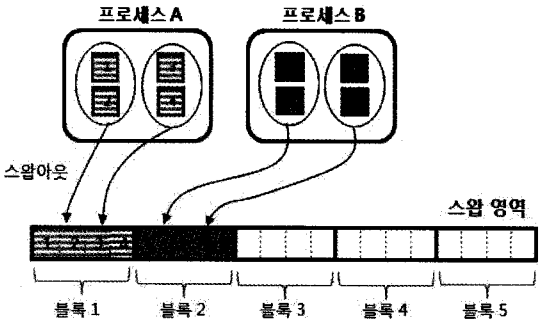


그림 5. 프로세스별로 스왑 아웃 페이지를 저장하는 예

한 스왑 슬롯의 할당 및 가비지 수집 기법을 연구하는 것이 필요하다.

3. PASS(Process-Aware Swap System)

본 논문에서는 스왑 저장 장치로서 플래시 메모리를 사용하는 모바일 컴퓨터를 위해 리눅스 스왑 시스템을 새로 고안하고 새로운 가비지 수집 기법을 연구한다.

기존 스왑 시스템에서는 디스크 탐색 시간을 고려했으나, 제안하는 스왑 시스템은 프로세스의 실행에 따라 스왑 저장 장치가 사용됨을 고려한다. 본 논문에서 제안한 프로세스 기반의 스왑 시스템인 PASS (Process-Aware Swap system)는 같은 프로세스에 속한 페이지들을 같은 플래시 메모리 블록에 저장시킴으로써 가비지 수집 비용을 줄인다.

3.1 전체 구조

본 절에서는 PASS의 전체 구조와 데이터 구조를 기술한다. 그림 6은 제안한 PASS의 전체 구조를 보여주고 있다. PASS는 가상 메모리 시스템에서 요청하는 스왑 아웃 및 스왑 인을 수행하는 함수들과 스왑 영역의 가비지 수집을 수행하는 가비지 수집기로 구성된다. 스왑 아웃 함수의 동작은 3.2.1절에서, 스왑 인 함수의 동작은 3.2.2절에서 자세히 설명한다. 가비지 수집은 3.3절에서 자세히 설명한다.

그림 7은 스왑 영역의 구조를 보여주고 있다. 스왑 영역은 플래시 메모리 장치의 한 파티션이고 여러 개의 플래시 메모리 블록들로 이루어진다. 각 블록은 여러 개의 스왑 슬롯들로 구성되며, 한 슬롯은 스왑 아웃되는 하나의 프로세스 페이지를 저장할 수 있다.

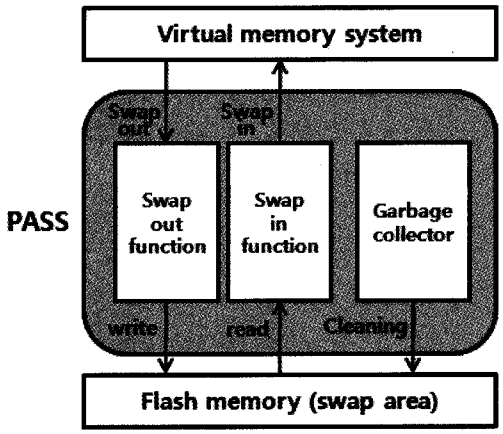


그림 6. PASS의 구조

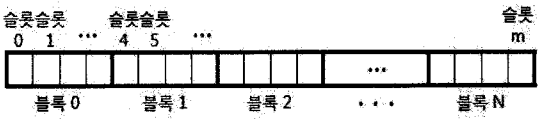


그림 7. 스왑 영역의 구조

그림은 한 블록이 4개의 슬롯으로 구성된 예를 보여주고 있다.

스왑 영역을 관리하기 위하여 block_map 구조체 배열을 사용한다. 그림 8은 block_map 구조체의 주요 내용을 개괄적으로 보여주고 있다. block_map은 블록의 상태(status), 스왑 슬롯들의 상태(slot_status) 배열, 블록을 공유하는 프로세스 수(share_proc_count)와 같은 정보를 가진다. status는 블록이 가용 상태이면 0, 사용 중이면 1 값을 가진다. slot_status는 배열로서 대응하는 슬롯이 가용 상태이면 0, 유효(valid) 상태이면 1, 무효(invalid) 상태이면 2 값을 가진다.

제안한 PASS는 프로세스 별로 블록을 할당한다. 프로세스 별로 할당된 블록을 관리하기 위하여 p_struct와 b_struct 구조체를 사용한다. 그림 9는 두 구조체의 주요 내용을 개괄적으로 보여주고 있다.

```
block_map {
    status;
    slot_status[ ];
    share_proc_count;
}
```

그림 8. block_map 구조체

```
p_struct {
    pid;
    status;
    block_count;
    block_link;
    proc_link;
}

b_struct {
    bid;
    block_link;
    end_link;
}
```

그림 9. p_struct와 b_struct 구조체

p_struct는 프로세스 번호(pid), 프로세스 상태(status), 할당된 블록 수(block_count), 블록의 연결 리스트(block_link), 프로세스의 연결 리스트(proc_link)와 같은 정보를 가진다. 프로세스 상태는 프로세스가 실행 중이면 1, 종료되었으면 0 값을 가진다. b_struct는 블록 번호(bid), 블록의 연결 리스트(block_link, end_link)와 같은 정보를 가진다. 블록 번호 bid는 block_map 배열의 인덱스로서 해당 블록의 정보를 찾을 수 있다.

그림 10은 p_struct와 b_struct의 사용 예를 보여주고 있다. 예를 들어, 프로세스 1900은 블록 5와 7을

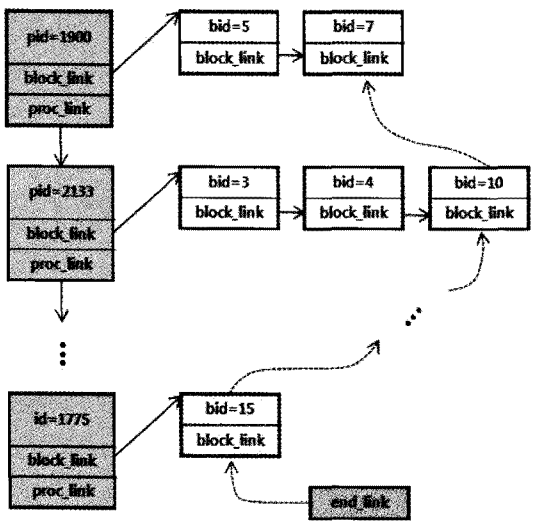


그림 10. p_struct와 b_struct의 예

할당받았고 프로세스 1775는 블록 15를 할당받았다. 각 프로세스에 할당된 마지막 블록들을 별도의 연결 리스트로 관리한다. 그림에서 end_link 구조체는 프로세스에 할당된 마지막 블록들을 연결 리스트로 관리하는데, b_struct의 end_link를 사용하여 연결한다.

각 프로세스의 블록 리스트 중에서 마지막 블록은 슬롯이 다 사용되지 않을 수 있기 때문에 블록의 공간이 낭비될 수 있다. 제안한 PASS는 마지막 블록의 사용률을 높이기 위하여 마지막 블록인 경우에만 여러 프로세스가 공유할 수 있게 하였다. block_map의 share_proc_count는 블록을 공유하는 프로세스의 수이다. 블록의 공유 방법은 3.3절 가비지 수집에서 기술하였다. 스왑 영역의 가용 블록 수가 정해진 한도보다 적어지게 되면 가비지 수집을 수행한다.

3.2 스왑 아웃과 스왑 인

PASS의 스왑 아웃과 스왑 인의 과정을 설명하기 위하여 가상 코드를 그림 11과 그림 12에서 보여주고 있다. 스왑 아웃과 스왑 인의 인자(parameter)는 프로세스 번호(pid), 페이지 프레임 주소(page_fram_addr), 페이지의 테이블 엔트리(page_table_entry)이다. 페이지 테이블 엔트리는 2.1절의 그림 2에서 설명하였다.

3.2.1 스왑 아웃

어떤 프로세스가 처음으로 스왑 아웃되면 p_struct를 생성한다. 또, 스왑 영역의 가용 블록을 할당하고 스왑 아웃된 페이지를 저장한다. 이후에 스왑 아웃되는 페이지는 할당된 블록에 순차적으로 저장한다. 블록을 다 사용하면 새 가용 블록을 할당한다. 가용 블록을 할당할 때마다 b_struct를 생성하여 p_struct의 블록 연결 리스트의 마지막에 연결한다.

3.2.2 스왑 인

스왑 아웃된 페이지가 참조되어 스왑 인 요청을 받으면 해당 페이지가 있는 슬롯에서 읽어 준다. block_map에서 스왑 인 된 슬롯의 상태를 무효(invalid) 상태로 바꾼다.

3.3. 가비지 수집

PASS의 가비지 수집 작업을 설명하기 위하여 간략한 가상 코드를 그림 13에서 보여주고 있다. 스왑

```
swap_out(pid, page_frame_addr, page_table_entry)
{
   if(pid에 대응되는 p_struct가 없음) {
         p_struct를 생성;
         가용 블록을 할당하고 첫 번째 슬롯에 page_frame의 내용을 복사;
         block_map에서 해당 슬롯의 status = 1, share_proc_count = 1;
         b_struct를 생성하고 p_struct의 블록 리스트 및 end_link의 리스트에 연결;
         p_struct의 status = 1, block_count = 1;
   } else {
         if(프로세스에 할당된 마지막 블록에 가용 슬롯이 있음) {
             가용 슬롯에 page_frame의 내용을 복사;
             block_map에서 해당 슬롯의 status = 1;
         } else {
             가용 블록을 할당하고 첫 번째 슬롯에 page_frame의 내용을 복사;
             block_map에서 해당 슬롯의 status = 1, share_proc_count = 1;
             b_struct를 생성하고 p_struct의 블록 리스트 및 end_link의 리스트에 연결;
             p_struct의 block_count 값 증가;
         }
   }
   page_table_entry에 스왑 아웃 페이지가 저장된 스왑 영역 번호와 스왑 슬롯 번호를 삽입;
}
```

그림 11. 스왑 아웃 알고리즘

```
swap_in(pid, page_frame_addr, page_table_entry)
{
   page_table_entry에서 스왑 영역 번호와 스왑 슬롯 번호를 추출;
   block_map에서 해당 슬롯을 무효 상태로 함(status = 2);
   슬롯에서 페이지 내용을 읽어 page_frame에 복사;
}
```

그림 12. 스왑 인 알고리즘

영역의 가용 블록 수가 정해진 한도보다 적어지게 되면 가비지 수집을 수행하여 가용 블록을 확보한다. PASS의 가비지 수집은 기존의 가비지 수집과 마찬가지로, 희생 블록의 선택, 희생 블록 내의 유효 슬롯의 복사, 희생 블록의 삭제의 3단계로 수행된다.

3.3.1 희생 블록의 선택

희생 블록은 블록의 사용률이 낮은 블록을 선택한다. 블록의 사용률은 블록의 총 슬롯 수에 대한 유효 슬롯 수의 비율이다. 예를 들어, 블록의 총 슬롯 수가 10이고, 유효 슬롯 수 3, 무효 슬롯 수 2, 가용 슬롯 수 5라면 블록의 사용률은 30%가 된다. 블록의 사용률이 낮으면 블록의 유효 슬롯이 적기 때문에 플래시 메모리의 페이지의 복사(읽기와 쓰기) 연산을 줄일 수 있고 가용 슬롯을 많이 확보할 수 있다. 이를 위해서, PASS는 먼저 종료된 프로세스가 있는 지를 검사한다.

① 만약 종료된 프로세스가 있다면 그 프로세스에 할당된 블록은 모두 무효이므로 모두 삭제하면 된다.

② 만약 종료된 프로세스가 없다면 블록 사용률이 가장 작은 블록을 희생 블록으로 정한다.

3.3.2 희생 블록 내의 유효 슬롯의 복사

선택된 희생 블록이 프로세스에 할당된 마지막 블

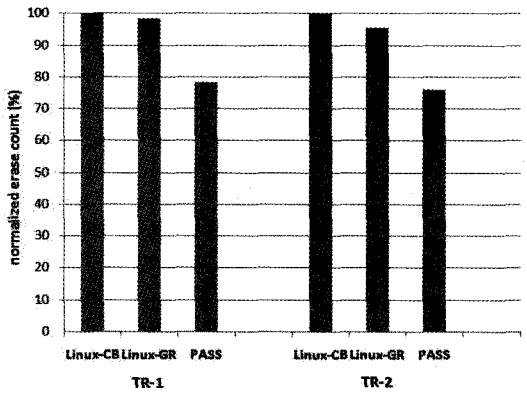
```
garbage_collection()
{
   if(종료된 프로세스가 존재함) {
        프로세스에 할당된 블록들을 삭제;
        p_struct와 프로세스에 속한 b_struct들을 제거;
        삭제된 블록에 대하여, block_map의 해당 데이터 초기화 (가용 상태로 만듬);
        return;
   }
   블록의 사용률이 최소인 블록을 찾음;
   if(희생 블록이 프로세스에 할당된 마지막 블록임) {
        if(end_link 블록들 중에서 충분한 슬롯이 있는 블록 D가 존재함) {
             희생 블록의 유효 슬롯을 블록 D에 복사;
             block_map에서 블록 D의 해당 슬롯 상태 변경, share_proc_count 증가;
        } else {
            새 블록을 할당하고 희생 블록의 유효 슬롯들을 복사;
            block_map에서 해당 슬롯의 상태 변경, share_proc_count = 1;
            b_struct를 생성하고 p_struct의 블록 리스트 및 end_link의 리스트에 연결;
            p_struct의 block_count 값 증가;
        }
   } else {
        희생 블록의 소유 프로세스의 마지막 블록 A에 희생 블록의 유효 슬롯을 복사;
        block_map에서 블록 A의 슬롯 상태 변경;
        if(복사할 유효 슬롯이 남아 있음) {
            if(end_link 블록들 중에서 충분한 슬롯이 있는 블록 D가 존재함) {
                희생 블록의 남은 유효 슬롯을 블록 D에 복사;
                block_map에서 블록 D의 슬롯 상태 변경, share_proc_count 증가;
            } else {
                새 블록을 할당하고 희생 블록의 유효 슬롯들을 복사;
                block_map에서 해당 슬롯의 상태 변경, share_proc_count = 1;
                b_struct를 생성하고 p_struct의 블록 리스트 및 end_link의 리스트에 연결;
                p_struct의 block_count 값 증가;
            }
        }
   }
   희생 블록을 삭제, b_struct를 제거;
   희생 블록의 소유 프로세스의 p_struct에서 block_count를 감소;
   희생 블록에 대하여, block_map의 해당 데이터 초기화 (가용 상태로 만듬);
}
```

그림 13. 가비지 수집 알고리즘

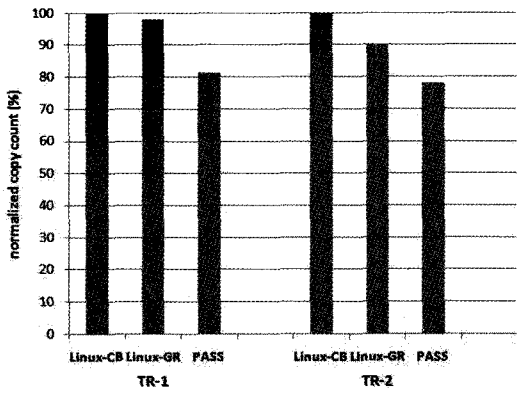
록인지 여부에 따라 복사 정책이 다르다.

① 희생 블록이 프로세스에 할당된 마지막 블록이라면, end_link 블록 리스트(각 프로세스의 마지막 블록들) 중에서 충분한 가용 슬롯이 있는 블록이 있는 지를 찾아 희생 블록의 유효 슬롯들을 복사한다. 복사할 블록을 찾지 못한 경우에는 새 블록을 할당하고, 희생 블록의 유효 슬롯들을 복사한다.

② 희생 블록이 프로세스에 할당된 마지막 블록이 아니라면, 그 프로세스에 할당된 마지막 블록에 유효 슬롯을 복사한다. 그래도 복사할 유효 슬롯이 남았다면, end_link 블록 리스트(각 프로세스의 마지막 블록들) 중에서 충분한 가용 슬롯이 있는 블록이 있는



(a) 삭제 비교



(b) 복사 비교

그림 14. 삭제 수와 복사 수의 비교

지를 찾아 희생 블록의 유효 슬롯들을 복사한다. 복사할 블록을 찾지 못한 경우에는 새 블록을 할당하고, 희생 블록의 유효 슬롯들을 복사한다.

4. 실험 결과

본 연구에서는 제안한 PASS의 성능을 검증하기 위하여 트레이스 기반의 PASS 시뮬레이터를 구현하였다. 시뮬레이터의 구조는 그림 6에서 보여준 PASS의 전체 구조와 같으며 동작 과정은 3장에서 설명한 스왑 아웃, 스왑 인 및 가비지 수집의 동작 과정과 같다. 성능 비교를 위하여 리눅스 스왑 시스템의 시뮬레이터를 구현하였으며 이때 기존의 가비지 수집 알고리즘인 Greedy 기법과 Cost-benefit 기법을 사용하였다. 각 시뮬레이터는 C 언어로 작성하였다.

실험을 위해 리눅스 커널 2.6.16에서 스왑 입출력의 트레이스를 수집하였다. 리눅스 커널에서 스왑 아웃 및 스왑 인을 수행하는 함수를 수정하였고, 블록 디바이스의 입출력 정보를 추출하는 도구인 blktrace를 활용하였다[17]. 트레이스를 수집할 때 사용한 컴퓨터의 사양은 CPU가 인텔 셀러론 1.3G, RAM은 256MB, 하드 디스크는 40GB이다. 스왑 영역은 RAM 메모리의 2배인 512MB 크기의 파티션으로 설정하였다. 두 개의 트레이스 셋을 수집하였다. 각 트레이스 셋은 일반 데스크탑 PC의 사용 환경을 1주일간 실행하였으며, 웹서핑, 파일 편집 및 저장, 인터넷에서 파일 다운로드, 프로그램 설치, 동영상 재생, 커널 컴파일 등을 수행하였다. 두 트레이스 셋을 각각 TR-1과 TR-2로 호칭하겠다.

그림 14는 제안한 PASS 기법과 리눅스 스왑 시스템에서 greedy 기법을 사용한 경우(Linux-GR), 리눅스 스왑 시스템에서 cost-benefit 기법을 사용한 경우(Linux-CB)의 성능 비교를 보여주고 있다. 모든 경우에 PASS가 성능이 우수하였고, Linux-CB가 성능이 제일 좋지 않았다. 그림 14에서 (a)는 가비지 수집 동안에 수행한 삭제 연산 수를 Linux-CB를 100%로 하여 정규화한 비율로 보여주고 있고, (b)는 가비지 수집 동안에 수행한 유효 슬롯의 복사 수를 Linux-CB를 100%로 하여 정규화한 비율로 보여주고 있다. 또한, 그림 15는 가비지 수집 동안에 수행한 삭제와 복사 연산을 가비지 수집 비용으로 환산한 결과를 보여주고 있다. 가비지 수집 비용은 식 (1)을 사용하였고 각 연산의 소요 시간은 표 1의 값을 사용

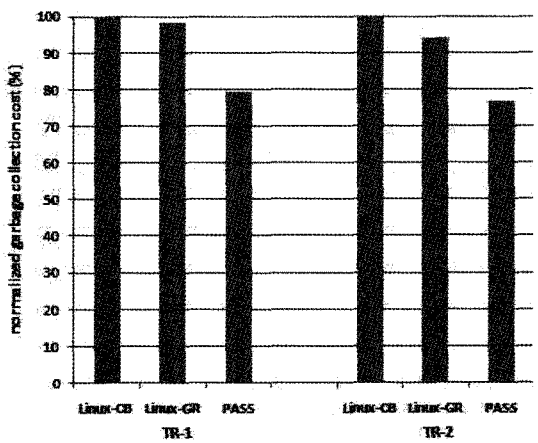


그림 15. 가비지 수집 비용의 비교

하였다. PASS의 가비지 수집 비용이 제일 적음을 알 수 있다.

이와 같이 프로세스 별로 블록을 할당하는 PASS 기법이 기존 리눅스 스왑 시스템 방식과 비교하여 가비지 수집 비용을 줄여 성능을 향상시킴을 알 수 있다. 실험에서는 프로세스의 종료가 없었으므로 프로세스의 종료를 감안한다면 성능이 더욱 향상될 것이다.

5. 결 론

플래시 메모리 저장 시스템의 성능은 변경 또는 삭제되는 데이터의 관리 및 가비지 수집으로 인해 제한되며, 플래시 메모리 기반의 스왑 시스템도 삭제되는 데이터를 효율적으로 관리하여 가비지 수집 비용을 최소화할 수 있도록 설계되어야 한다.

모바일 컴퓨터를 위한 리눅스에서 플래시 메모리 기반 스왑 시스템의 필요성이 커져가고 있다. 기존 리눅스 스왑 시스템은 디스크의 탐색 시간을 최소화하기 위해 최적화되어 있으나 플래시 메모리에서는 탐색이 요구되지 않는다. 또한, 스왑 시스템은 프로세스의 실행 중에 사용되는 페이지들을 저장하기 때문에 프로세스 실행과 밀접한 관련이 있다. 이러한 성질을 이용하여 본 논문에서는 프로세스 별로 플래시 메모리 블록을 할당하는 PASS 기법을 제안하였고 트레이스 기반의 실험을 통해 기존 리눅스 스왑 시스템 방식보다 가비지 수집 성능이 우수함을 확인할 수 있었다.

참 고 문 헌

[1] D. P. Bovet and M. Cesati, Understanding the Linux Kernel, 3rd Edition, O'Reilly, 2005.

[2] Samsung Electronics. K9F2G08UXA 256M x 8bits NAND flash memory Data Sheet, http://www.samsungelectronics.com

[3] A. Kawaguchi, S. Nishioka, and H. Motoda, "Flash Memory Based File System," in Proceedings of USENIX95, pp. 155-164, 1995.

[4] M. Wu and W. Zwanepoel, "envy: A Non-volatile, Main Memory Storage System," in Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, 1994.

[5] M. Chiang and R. Chang, "Cleaning Policies in Mobile Computers Using Flash Memory," *Journal of Systems and Software,* Vol. 48, No. 3, pp. 213-231, 1999.

[6] M. Chiang, P. Lee, and R. Chang, "Using Data Clustering to Improve Cleaning Performance for Flash Memory," *Software Practice and Experience,* Vol. 29, No. 3, pp. 267-290, 1999.

[7] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient Identification of Hot Data for Flash Memory Storage Systems," *ACM Transaction on Storage,* Vol. 2, No. 1, 2006.

[8] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A Replacement Algorithm for Flash Memory," in Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems, pp. 234-241, 2006.

[9] Y. Yoo, H. Lee, Y. Ryu, and H. Bahn, "Page Replacement Algorithms for NAND Flash Memory Storages," *Lecture Notes in Computer Science,* Vol. 4705, Springer-Verlag, pp. 201-212, Aug. 2007.

[10] 유윤석, 류연승, "NAND 플래시 메모리 저장장치를 위한 요구 페이징 기법 연구", 멀티미디어학회 논문지 제10권, 제5호, pp. 583-593, 2007. 5.

[11] O. Kwon, Y. Yoo, K. Koh, and H. Bahn, "Replacement and Swapping Strategy to Improve Read Performance of Portable Consumer Devices Using Compressed File Systems," *IEEE Transactions on Consumer Electronics,* Vol. 54, No. 2, pp. 551-559, May 2008.

[12] 이혜정, 반효경, "Analyzing Virtual Memory Write Characteristics and Designing Page Replacement Algorithms for NAND Flash Memory," 정보과학회 논문지 36(6), pp. 543-556, 2009.

[13] J. Park, H. Lee, S. Hyun, K. Koh, and H. Bahn, "A Cost-aware Page Replacement Algorithm

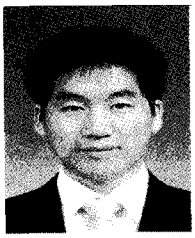
for NAND Flash-based Mobile Embedded Systems," in Proceedings of International Conference on Embedded Software, pp. 315-324, 2009.

[14] M. Saxena and M Swift, "FlashVM: Revisiting the Virtual Memory Hierarchy," in Proceedings of USENIX Workshop on Hot Topics in Operating Systems, 2009.

[15] D. Jung, J. Kim, S. Park, J. Kang, and J. Lee, "FASS: A Flash-Aware Swap System", in Proceedings of International Workshop on Software Support for Portable Storage, 2005.

[16] S. Koh, S. Jeon, and Y. Ryu,, "A New Linux Swap System for Flash Memory Storage Devices," in Proceedings of 3rd International Workshop on Data Storage Devices and Systems, pp. 151-156, Jun, 2008.

[17] blktrace, http://linux.die.net/man/8/blktrace



전 선 수

2002년 3월~2008년 2월 명지대학교 컴퓨터소프트웨어학과 학사

2008년 3월~2010년 2월 명지대학교 컴퓨터소프트웨어학과 석사

2010년 1월~현재 (주)팬택 중앙연구소 연구원, 안드로이드 시스템S/W 개발

관심분야: 파일 시스템, 가상 메모리 시스템, 차세대 메모리

류 연 승

1986년 3월~1990년 2월 서울대학교 계산통계학과 학사

1990년 3월~1992년 2월 서울대학교 전산과학과 석사

1992년 3월~1996년 8월 서울대학교 전산과학과 박사

1996년 9월~2000년 8월 삼성전자

2000년 9월~2003년 2월 한림대학교 조교수

2003년 3월~현재 명지대학교 컴퓨터공학과 부교수

관심분야: 운영체제, 네트워크시스템, 스토리지 등