■ 2008년도 학생논문 경진대회 수상작

NAND 플래시메모리를 위한 가상메모리의 쓰기 참조 분석 및 페이지 교체 알고리즘 설계

(Analyzing Virtual Memory Write Characteristics and Designing Page Replacement Algorithms for NAND Flash Memory)

이 혜 정 † 반 효 경 ††
(Hyejeong Lee) (Hyokyung Bahn)

요 약 최근 NAND 플래시메모리를 모바일시스템의 파일저장용 뿐 아니라 가상메모리의 스왑장치용으로 사용하려는 시도가 늘고 있다. 가상메모리의 페이지 참조는 시간지역성이 지배적이어서 LRU 및 이를 근사시킨 CLOCK 알고리즘이 널리 사용된다. 한편, NAND 플래시메모리는 읽기 연산에 비해 쓰기 연산의 비용이 높아 이를 고려한 페이지 교체 알고리즘이 필요하다. 본 논문에서는 가상메모리의 읽기/쓰기 참조 패턴을 독립적으로 분석하여 시간지역성이 강한 읽기 참조와 달리 쓰기 참조의 경우 시간지역성의 순위 역전 현상이 발생함을 발견하였다. 이에 근거하여 본 논문은 쓰기의 재참조 성향 예측을 위해 시간지역성뿐 아니라 쓰기 연산의 빈도를 함께 고려하는 페이지 교체 알고리즘을 제안한다. 새로운 알고리즘은 연산별 I/O 비용을 고려해서 메모리 공간을 읽기 연산과 쓰기 연산에 독립적으로 할당하고 참조 패턴의 변화에 적응해 할당 공간을 동적으로 변화시킨다. 알고리즘의 시간 오버헤드가 매우 적어 가상메모리 시스템에서 사용될 최적의 조건을 갖추고 있으며 파라미터 설정이 필요 없음에도 CLOCK, CAR, CFLRU 알고리즘에 비해 20-66% 정도의 I/O 성능을 향상시킴을 보였다.

키워드 : NAND 플래시메모리, 페이지 교체 알고리즘, 가상메모리

Abstract Recently, NAND flash memory is being used as the swap device of virtual memory as well as the file storage of mobile systems. Since temporal locality is dominant in page references of virtual memory, LRU and its approximated CLOCK algorithms are widely used. However, cost of a write operation in flash memory is much larger than that of a read operation, and thus a page replacement algorithm should consider this factor. This paper analyzes virtual memory read/write reference patterns individually, and observes the ranking inversion problem of temporal locality in write references which is not observed in read references. With this observation, we present a new page replacement algorithm considering write frequency as well as temporal locality in estimating write reference behaviors. This new algorithm dynamically allocates memory space to read/write operations based on their reference patterns and I/O costs. Though the algorithm has no external parameter to tune, it supports optimized implementations for virtual memory systems, and also performs 20-66% better than CLOCK, CAR, and CFLRU algorithms.

Key words : NAND Flash Memory, Page Replacement Algorithm, Virtual Memory

· 본 논문은 2007년 교육과학기술부의 한국학술진흥재단 지원(KRF-2007-331-D00364)과 삼성전자 산학과제 지원을 받아 수행된 연구임

† 비 회 원 : 이화여자대학교 컴퓨터공학과
huizh@ewhain.net
†† 종신회원 : 이화여자대학교 컴퓨터공학과 교수
bahn@ewha.ac.kr
(Corresponding author임)

논문접수 : 2009년 6월 8일
심사완료 : 2009년 7월 30일

정보과학회논문지 : 시스템 및 이론 제36권 제6호(2009.12)

1. 서 론

최근 PMP(Portable Media Player, 휴대용 미디어 플레이어), 스마트폰(Smartphone) 등 모바일기기의 기능이 다원화되고 멀티태스킹이 일반화되면서 이들 시스템에서 가상메모리 기능의 중요성이 증가하고 있다. 한편, 하드디스크를 사용하지 않는 모바일기기에서는 일반적으로 NAND 플래시메모리를 스왑 영역으로 사용하므로 이에 적합한 가상메모리 관리가 필요하다. NAND 플래시메모리는 그 물리적 특성이 하드디스크와 상이하여 플래시주소변환계층(FTL, Flash Translation Layer), 플래시전용 파일시스템 등의 연구가 널리 이루어졌으나 가상메모리를 위한 연구는 아직 초기 수준에 머물러 있다[1-3]. 본 논문에서는 NAND 플래시메모리를 스왑장치로 사용하는 시스템을 위한 가상메모리의 페이지 참조 특성을 분석하고 이에 근거한 새로운 페이지 교체 알고리즘을 연구한다.

가상메모리의 페이지 참조는 전통적으로 시간지역성(temporal locality)이 강한 것으로 알려져 있어 LRU (Least Recently Used) 및 이를 근사시킨 클럭 알고리즘(clock algorithm)이 널리 사용되고 있다[4]. 한편, NAND 플래시메모리는 읽기 연산에 비해 쓰기 연산의 비용이 높아 이를 고려한 페이지 교체 알고리즘의 설계가 필요하다. 본 논문은 가상메모리의 페이지 참조를 읽기 참조와 쓰기 참조로 나누어 각각의 참조 패턴을 시간지역성 측면에서 분석한다. 그 결과 시간지역성이 강한 읽기 참조와 달리 쓰기 참조는 일정 범위를 넘어설 경우 시간지역성의 순위 역전 현상이 발생하여 페이지의 참조 가능성 예측에 한계가 있음을 발견하였다. 이에 근거하여 본 논문은 쓰기의 재참조 성향을 예측하기 위해 시간지역성뿐 아니라 페이지의 쓰기 빈도를 함께 고려하는 새로운 교체 알고리즘을 제안한다. 새로운 알고리즘은 읽기/쓰기 연산의 패턴 분석을 연산의 특성에 맞게 독립적으로 수행하여 연산별로 페이지의 재참조 가능성을 정교하게 예측한다. 또한, 각 연산별 I/O 비용을 고려해서 메모리 공간을 읽기 연산과 쓰기 연산에 독립적으로 할당하고 참조 패턴의 변화에 적응해 할당 공간을 동적으로 변화시킨다. 실제 프로세스의 가상 메모리 접근 트레이스를 추출한 시뮬레이션 실험을 통해 제안하는 방식이 기존 알고리즘들에 비해 플래시메모리의 전체 I/O 시간 면에서 향상된 성능을 보임을 확인하였다. 특히, 가상메모리 시스템에서 가장 널리 사용되는 클럭 알고리즘에 비해 평균 23.9%, 최대 66.5%의 성능 향상을 나타내었다. 그럼에도 제안한 알고리즘의 시간 오버헤드가 매우 적고 LRU처럼 페이지 접근 시마다 리스트 조작이 이루어지는 알고리즘과 달리 메모리에서 적중된 페이지에 대해서는 하드웨어에 의한 비트 세팅만 일어날 뿐 아무런 부가 연산이 이루어지지 않아 가상메모리 시스템에서 사용될 수 있는 최적의 조건을 갖추고 있다. 또한 파라미터 튜닝이 자동적으로 이루어져 파라미터 설정에 따라 성능 차이를 보이는 연구들과 차별화된다. 본 논문의 주요 연구 내용을 요약하면 다음과 같다.

- 비용이 높은 플래시메모리의 쓰기 연산 특성을 규명하기 위해 가상메모리 페이지의 읽기 참조와 쓰기 참조의 시간지역성을 독립적으로 분석하고 이를 읽기와 쓰기의 재참조 가능성 예측에 각각 활용한다.
- 읽기 참조의 경우 시간지역성이 강하여 읽기의 재참조 가능성 예측에 있어 시간 지역성에 근거한 LRU 및 CLOCK 알고리즘이 적합함을 다시 한번 확인한다.
- 쓰기 참조의 경우 시간지역성의 역전현상이 일부 구간에서 발생하여 시간지역성만으로는 재참조 가능성 예측에 제약이 있음을 발견하여 페이지의 쓰기 빈도를 함께 고려한 새로운 방식으로 쓰기 참조 가능성을 예측한다.
- 연산별 I/O 비용 절감 효과에 기반해 읽기와 쓰기를 위한 메모리 공간을 별도로 할당하고 각 연산의 비율, 재참조 가능성 등에 근거해 동적으로 공간 할당치를 조절하는 적응적 메커니즘을 사용한다.

본 논문의 구성은 다음과 같다. 2장에서는 가상메모리의 페이지 참조 특성과 플래시메모리를 위한 페이지 교체 알고리즘에 대한 관련 연구를 살펴본다. 3장에서는 가상 메모리의 페이지 참조 패턴을 읽기 참조와 쓰기 참조로 나누어 관찰하고 이를 분석한 결과를 기술한다. 4장에서는 본 논문이 제안하는 NAND 플래시메모리를 위한 가상메모리의 페이지 교체 알고리즘을 상세히 설명한다. 5장에서는 기존의 페이지 교체 알고리즘과 본 논문이 제안한 알고리즘의 실험 결과를 시뮬레이션을 통해 비교 분석한다. 끝으로 6장에서는 본 논문의 결론을 제시한다.

2. 관련 연구

본 장에서는 가상메모리의 페이지 참조와 플래시메모리를 위한 기존의 페이지 교체 알고리즘에 대해 간단히 살펴본다.

2.1 가상메모리의 페이지 참조와 LRU 알고리즘

프로그램의 주소공간을 구성하는 가상메모리 페이지들은 '최근에 참조된 페이지가 가까운 미래에 다시 참조될 가능성이 매우 높은 특성'을 가지고 있다. 이와 같은 시간지역성을 가진 참조 패턴에 대해서는 LRU 알고리즘이 최적의 교체 알고리즘으로 알려져 있다[5]. LRU 알고리즘은 메모리 상의 페이지들을 마지막 참조 시점

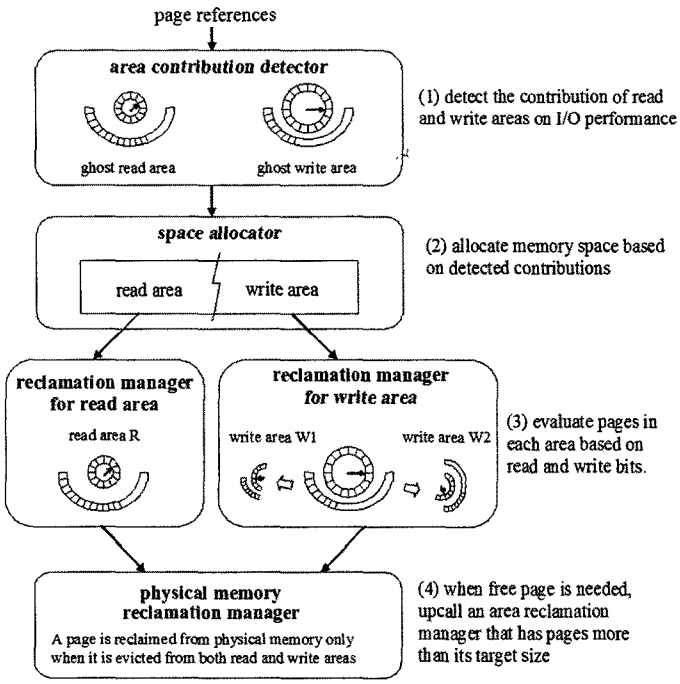


그림 1 본 논문이 제안하는 알고리즘의 기본 흐름

에 따라 우선 순위 리스트로 유지하고, 메모리 공간이 부족한 경우 가장 오래 전에 참조되었던 페이지를 교체한다. LRU 알고리즘은 구현 방식이 단순하지만 시간지역성을 가지는 다양한 환경에서 좋은 성능을 나타내어 가상메모리 환경뿐 아니라 버퍼 캐쉬, 웹 캐쉬 등 여러 영역에서 폭넓게 사용되고 있다.

한편, 순수한 LRU 알고리즘이 가상메모리의 페이지 교체 알고리즘으로 사용되기에는 현실적인 문제점이 있다. LRU 알고리즘은 페이지 참조가 일어날 때마다 참조된 페이지를 리스트의 한쪽 끝(MRU 포지션)으로 이동시켜야 한다. 이는 운영체제 커널 단에서 수행해야 하는 업무인데 사용자 프로세스가 CPU에서 실행 중일 때에는 페이지 참조가 이루어지더라도 커널에 의한 리스트 조작이 불가능하다. 페이지 폴트가 발생하여 CPU의 제어권이 커널로 넘어온 경우에는 리스트의 조작이 가능해지지만 페이지 폴트가 발생하지 않아 메모리에서 직접 참조된 페이지들에 대해서는 참조된 시간 순서를 정확히 알아내는 것이 현실적으로 불가능하다. 따라서, 대부분의 시스템에서는 사용자 프로세스가 수행 중일 때에 참조된 페이지들에 대해서는 정확한 참조 시각을 파악하는 대신 페이지가 참조될 때마다 하드웨어적으로 세팅되는 접근 비트(access bit) 또는 참조 비트(reference bit)에 근거해서 최근에 참조되었는지의 여부만을 파악하고 이에 근거한 LRU의 근사적인 운영 방식을 사용한다. 그 대표적인 예가 클럭 알고리즘(clock algorithm)이다. 클럭 알고리즘은 페이지 교체가 필요할 경우 메모리 내의 페이지들을 순차적으로 스캔하여 접근 비트가 세팅된 페이지는 비트를 클리어시키고 그렇지 않은 페이지는 교체시킨다. 이렇게 하면 모든 페이지들을 한번 스캔하는 동안 재참조가 이루어지지 않을 경우 그 페이지는 교체 대상 페이지가 된다. 클럭 알고리즘은 LRU 알고리즘과 달리 가장 오래 전에 참조된 페이지를 교체하지는 않지만 적어도 최근에 참조되지 않은 페이지를 교체하기 때문에 시간지역성을 고려하는 알고리즘이라 할 수 있다.

2.2 플래시메모리의 연산 비용과 CFLRU 알고리즘

리눅스를 비롯한 대부분의 운영체제는 하드디스크를 저장장치로 사용하는 환경에 최적화되어 있다. 따라서, 가상메모리의 페이지 교체 알고리즘은 메모리 상의 페이지들 중에서 가장 재사용될 가능성이 낮은 페이지를 교체함으로써 메모리 적중률(hit rate)을 높이는 것에 초점을 맞추고 있다. 그러나 NAND 플래시메모리는 읽기 연산과 쓰기 연산의 비용이 균일하지 않는 등 하드디스크와는 다른 고유한 물리적 특성을 지니고 있다. 표 1에서 보는 것과 같이 NAND 플래시메모리에서는 쓰기 연산이 읽기 연산에 비해 동일한 크기의 데이터에 대해 8배 정도의 I/O 시간이 소요된다. 또한, 동일 위치에 쓰기 연산을 수행하기 위해서는 삭제 연산이라는 많은 시

표 1 NAND 플래시메모리의 연산별 접근 시간[6]

연산(operation)	접근 시간(access time)
읽기(read)	25μs(2KB)
쓰기(write)	200μs(2KB)
삭제(erase)	1.5ms(128KB)

간이 소요되는 부가적인 연산이 필요하다. 따라서 대부분의 NAND 플래시메모리는 플래시주소변환계층(Flash Translation Layer)을 두어 동일한 위치가 아닌 다른 곳에 쓰기(out-place update)를 하여 삭제 연산의 직접적인 비용을 숨기게 된다. 이와 같은 특징 때문에 메모리 적중률만을 추구하는 기존의 페이지 교체 알고리즘으로는 NAND 플래시메모리 시스템에서 최적의 성능을 기대하기 어려우며, 성능 척도 역시 메모리 적중률이 아닌 전체 I/O 시간(total I/O time)으로 측정하는 것이 적절하다.

CFLRU(Clean-First LRU) 알고리즘은 메모리 적중률 뿐 아니라, 읽기/쓰기 연산의 비용이 이질적인 NAND 플래시메모리의 물리적 특성을 고려한 페이지 교체 알고리즘이다[3]. CFLRU는 페이지 교체 시 메모리에서 삭제만 하면 되는 클린 페이지와 스왑 영역에 써야 하는 더티 페이지의 가치를 달리 취급하는 방식으로, 메모리 적중률을 크게 저하시키지 않는 범위에서 더티 페이지의 교체를 최대한 지연시킴으로써 고비용인 NAND 플래시메모리의 쓰기 연산 횟수를 줄이는 것을 목표로 한다.

CFLRU는 기본적으로 LRU 리스트에 의해 페이지들을 관리한다. CFLRU의 리스트는 그림 2와 같이 워킹 영역(working region)과 클린-우선 영역(clean-first region)으로 구분된다. 워킹 영역은 LRU 리스트 상에서 상대적으로 최근에 참조된 페이지들을 담고 있는 공간으로, 이 영역에 속한 페이지들은 LRU 알고리즘에 의해 교체 대상을 선정한다. 클린-우선 영역에 속한 페이지들은 LRU 리스트에서 상대적으로 오래 전에 참조된 페이지들로, 교체 알고리즘 동작 시 워킹 영역보다 우선해서 교체된다. 클린-우선 영역에서 CFLRU 알고리즘은 페이지의 클린/더티 여부를 구분한다. 클린 페이지는 메모리에 올라온 이후 페이지의 내용이 변하지 않은 페이지이고, 더티 페이지는 메모리에 올라온 이후 페이지의 내용이 변경된 페이지이다. 따라서 클린 페이지가 교체 대상으로 선정된 경우에는 플래시메모리에 추가 연산 없이, 메모리에서 해당 페이지를 지우기만 하면 된다. 반면 더티 페이지가 교체 대상으로 선정된 경우에는 수정된 내용을 플래시메모리에 쓴 후에 해당 페이지를 메모리에서 지워야 한다. CFLRU는 클린-우선 영역 내에서 클린 페이지를 우선적으로 교체한다. 이는 더티 페이지를 교체하는 경우 발생하는 고비용의 쓰기 연산을 줄이려는 목적이다. CFLRU에서는 클린-우선 영역을 윈도우(window)라고 부른다.

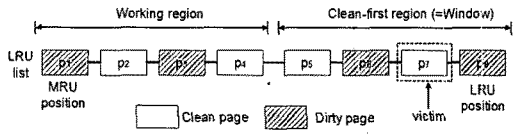


그림 2 CFLRU 알고리즘의 페이지 교체 예

그림 2는 CFLRU 알고리즘에서의 페이지 교체 예를 보여주고 있다. 리스트 내에서 가장 오래전에 참조된 페이지는 p8이지만, 메모리 부족 시 윈도우 내에서 가장 오래전에 참조된 클린 페이지인 p7이 교체 대상이 된다. 윈도우 내에 클린 페이지가 존재하지 않는 경우에만 더티 페이지가 교체 대상이 되기 때문에, 윈도우 내의 페이지들 중 교체 대상이 되는 페이지의 순서는 p7 → p5 → p8 → p6이다.

2.3 기존 페이지 교체 알고리즘의 한계

CFLRU 알고리즘은 LRU 리스트의 특정 범위 내에서 클린 페이지를 우선적으로 삭제하는 방식으로, 기존 LRU 알고리즘을 NAND 플래시메모리에 적합하도록 변형시킨 최초의 시도라는 점에서 의미를 갖는다. 그러나 아래와 같은 한계점을 가지고 있다.

- 페이지에 쓰기가 발생했는지의 여부만 고려할 뿐, 쓰기 참조의 빈도(frequency)나 쓰기 참조의 최근성(recency)을 고려하지 않는다.
- 윈도우를 구분 짓는 하나의 정점을 기준으로 쓰기를 동반하는 페이지와 그렇지 않은 페이지의 우선순위를 극단적으로 다르게 부여한다.
- 클린 페이지를 찾기 위해 최악의 경우 윈도우 내의 페이지를 전부 탐색해야 하는 오버헤드가 발생할 수 있다.
- 읽기/쓰기 요청의 비율 및 패턴 등 워크로드의 특성 변화에 적응적으로 동작하지 못한다.
- 워크로드 특성에 따라 윈도우 사이즈의 조절(tuning)이 필요하다.
- 가상메모리 시스템에서 사용되기에는 리스트 조작의 오버헤드가 크다.

3. 가상메모리의 페이지 참조 특성 분석

본 장에서는 가상메모리의 페이지 참조 특성을 시간지역성 측면에서 분석하고 이를 읽기 연산과 쓰기 연산 관점에서 각각 관찰한 결과를 기술한다. 쓰기 연산의 경우에는 시간지역성 뿐 아니라 쓰기 빈도 측면의 분석을 함께 수행하여 페이지의 재참조 가능성 예측을 효율적으로 수행할 수 있는 방안에 대해 논의한다. 사용된 응

용 프로그램은 리눅스의 Xwindow 상에서 수행된 음악 파일 재생(xmms), 이미지 편집(gqview), 문서 편집(gedit), 게임(freecell) 프로그램 등이다. 워크로드에 대한 자세한 설명은 5장 실험 부분에서 하도록 한다. 메모리 참조 요청의 종류는 명령 읽기(instruction read), 데이터 읽기(data read), 데이터 쓰기(data write)로 나누어 볼 수 있으며, 본 장에서는 전체 참조 패턴과 읽기 참조 패턴, 그리고 쓰기 참조 패턴으로 분류하여 각 요청 종류별로 페이지의 재참조 가능성에 미치는 영향을 살펴본다.

3.1 시간 지역성(temporal locality)

기존의 연구 결과와 마찬가지로 메모리 참조 패턴은 전체적으로 강한 시간 지역성(temporal locality)이 나타났다. 그림 3은 이러한 참조 성향을 나타낸 그래프로, x축은 LRU 리스트 상에서의 페이지의 순위(즉, LRU 스택 거리)를 의미한다. 그림 3에서 x축의 값이 1인 경우는 LRU 리스트의 MRU 포지션, 즉 가장 최근에 참조된 페이지를 의미하고, x 값이 커질수록 더 오래전에 참조된 페이지를 의미한다. y축은 해당 페이지 순위에서 페이지 참조가 일어난 횟수를 의미한다. 그림 3에서 보는 바와 같이 x축의 값이 작을수록 y축 값이 크게 나타나는 것은 최근에 참조된 페이지가 가까운 미래에 다시 참조되는 경우가 많음을 의미하는 것으로, 이러한 참조 패턴 하에서는 LRU 알고리즘을 사용할 경우 높은 메모리 적중률을 얻을 수 있다[7].

그림 3은 읽기 참조와 쓰기 참조를 포함한 모든 페이지 참조에 대한 시간지역성 그래프이다. 그림 4와 5는 이를 읽기 참조와 쓰기 참조로 분류해서 각 연산별 페이지 참조 패턴을 살펴본 것이다. 예를 들어 그림 4의 x축은 읽기 연산의 최근성 순위(즉, 가장 최근에 읽기 연산이 일어난 페이지의 순위가 1)이고, y축은 해당 페이지 순위에서 읽기 참조가 다시 일어난 횟수를 의미한다. 그림 4에서 볼 수 있듯이, 읽기 참조에서는 시간지역성이 강하게 나타났다. 오히려 그림 3에서 나타났던 일부 돌출된 영역이 사라지고 평탄한 그래프를 나타내어 읽기 참조만의 시간지역성이 전체 참조에서보다 더 명확하게 나타나는 것을 확인할 수 있다.

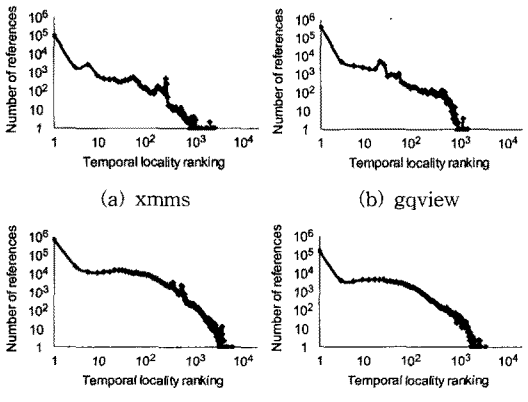


그림 3 시간지역성 순위에 따른 참조 횟수(읽기+쓰기)

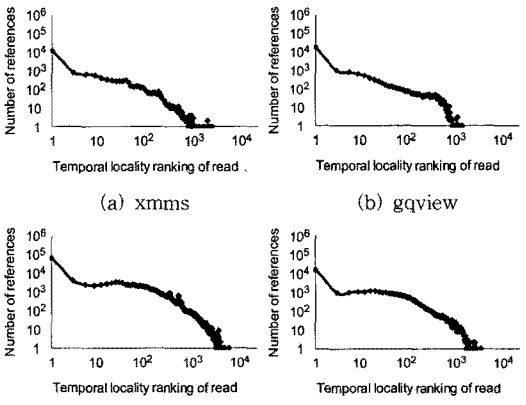


그림 4 읽기 참조의 시간지역성 순위에 따른 읽기 참조 발생 횟수

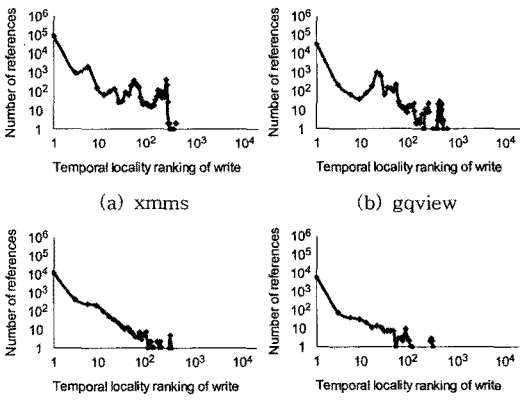


그림 5 쓰기 참조의 시간지역성 순위에 따른 쓰기 참조 발생 횟수

이에 비해 그림 5에서 볼 수 있듯이 쓰기 참조의 경우에는 시간지역성이 다소 불규칙적인 경향을 나타내는 것을 확인할 수 있다. 특히, (a), (b), (d) 등의 응용 프로그램에서는 순위에 따른 시간지역성의 역전 현상이 뚜렷이 나타나 시간지역성만으로는 쓰기 연산의 참조 가능성 예측이 정확하지 않음을 보여주고 있다. 이와 같은 현상이 나타나는 원인을 정확히 규명하기는 어렵지만 캐쉬메모리의 후방 쓰기(write-back)에 의한 영향인

것으로 추측할 수 있다. 메모리 참조 요청 중 일정 부분은 캐쉬메모리에 흡수되므로 본 논문에서 분석한 트레이스들의 페이지 참조는 캐쉬메모리에서 접근 실패하여 실제 메모리에 요청이 전달된 경우만을 그 대상으로 한다. 한편, 읽기 요청의 경우에는 캐쉬메모리에서 접근 실패한 읽기 요청이 그대로 메모리에 전달되므로 시간지역성이 다소 약해질 수는 있으나 그 성질이 크게 훼손되지는 않는다. 이에 비해 쓰기 요청의 경우에는 실제 쓰기 요청이 발생한 시점에는 캐쉬메모리에만 쓰므로 메모리에 요청이 전달되지 않다가 해당 데이터가 캐쉬메모리에서 쫓겨나는 시점에 메모리에 쓰기 요청이 전달되므로 요청이 실제로 일어난 시점과 메모리에 전달되는 시점이 일치하지 않는다. 따라서, 쓰기의 시간지역성이 상당 부분 훼손되는 것으로 추측된다.

3.2 쓰기 참조의 빈도(write reference frequency)

3.1절에서는 가상메모리 페이지의 쓰기 참조에 있어 시간지역성이 다소 약하거나 불규칙적인 성향을 보이는 것을 확인하였다. 본 절에서는 페이지의 쓰기 빈도의 순위에 따른 쓰기 연산의 재참조 성향을 살펴본다. 이를 위해 쓰기 참조 페이지들에 대해 매 시점 과거 쓰기 횟수에 따른 순위를 유지하고 각 순위의 페이지 중 쓰기 연산이 다시 일어난 빈도를 조사하였다. 그림 6에서 x축은 매 시점 쓰기 횟수에 기반한 페이지의 순위를 나타낸다. x축의 값이 1인 경우는 과거에 쓰기 연산의 횟수가 가장 많았던 페이지를 의미하고, x축의 값이 커질수록 쓰기 횟수가 적었던 페이지를 의미한다. y축은 해당 페이지 순위에서 쓰기 참조가 이루어진 횟수를 의미한다. 즉, 매 시점 페이지들의 쓰기 참조 순위를 유지하고 있다가 다시 참조된 순위에 대해 y값을 1씩 증가시키는 방식으로 그래프를 작성하였다. 그림 6에서 보는 바와 같이 쓰기 빈도의 순위값이 작을수록 쓰기 참조가 더 빈번히 발생한 것을 확인할 수 있다. 이는 과거 빈번히 쓰기 연산이 발생했던 페이지에 쓰기 참조 요청이 다시 일어날 가능성이 높은 것을 의미한다. 그림 6에 의하면 쓰기 참조의 빈도는 시간지역성과 달리 순위 역전 현상이 관찰되지 않으며 특정 순위까지는 시간지역성보다 높은 참조 횟수를 유지하고 있는 것을 확인할 수 있다.

이와 같이, 읽기 참조에 대한 페이지의 참조 가능성은 시간지역성만으로도 잘 모델링되지만, 쓰기 참조의 경우 시간지역성 뿐 아니라 과거 쓰기 횟수에 따른 재참조 성향을 함께 사용하는 것이 적절함을 확인할 수 있다. 따라서 미래의 쓰기 참조 가능성을 정확히 예측하기 위해서는 쓰기 참조가 발생하는 페이지의 과거의 참조 시점뿐 아니라 쓰기 참조 빈도도 함께 고려하는 페이지 교체 알고리즘의 설계가 필요하다.

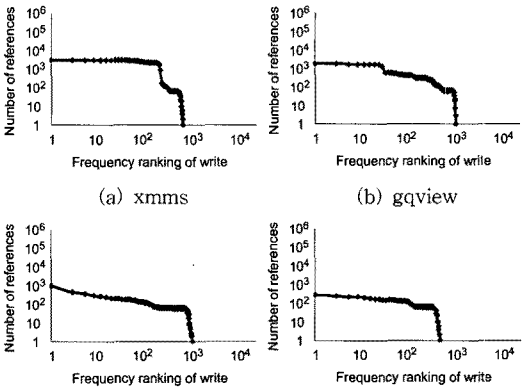


그림 6 쓰기 참조 빈도의 순위에 따른 쓰기 참조 발생 횟수

4. NAND 플래시메모리를 위한 페이지 교체 알고리즘

본 장에서는 NAND 플래시메모리를 스왑 장치로 사용하는 가상메모리 시스템을 위한 새로운 페이지 교체 알고리즘인 CRAW(Clock for Read And Write) 알고리즘을 설명한다. CRAW 알고리즘은 읽기 연산과 쓰기 연산을 위한 메모리 영역을 독립적으로 할당하고 두 영역이 입출력 수행 시간 절감에 어느 정도 효과가 있는지를 온라인으로 탐지하여 할당 영역의 크기를 동적으로 조절한다. 각각의 영역에서는 클럭 알고리즘과 유사한 효율적인 구현 방식을 채택하여 독립적인 페이지 교체가 이루어진다. 각 영역에서 교체 대상 페이지를 선정하기 위해서는 3장에서 설명한 가상메모리 페이지의 읽기 및 쓰기 참조의 특성을 반영한다. 즉, 읽기 참조에 대해서는 시간지역성을 고려하고 쓰기 참조에 대해서는 시간지역성과 쓰기 빈도를 함께 고려하여 페이지의 재참조 가능성을 예측한다. 이와 같이 연산별 특성에 맞게 메모리 영역을 독립적으로 관리하여 고비용의 쓰기 연산을 유발하는 더티 페이지를 메모리에 보호하면서도 읽기 연산이 매우 빈번히 일어나는 페이지 역시 메모리에 유지한다.

4.1 읽기 영역과 쓰기 영역의 크기 조절

본 논문이 제안하는 알고리즘은 읽기 영역과 쓰기 영역의 크기 조절을 위해 가상 페이지(ghost page)를 사용한다. 가상 페이지는 내용은 존재하지 않고 메모리에서 최근에 삭제된 페이지가 어떤 것인지에 대한 메타정보만을 일정 기간 유지하는 역할을 한다. 읽기 영역과 쓰기 영역 각각에 가상 페이지를 할당하고 가상 페이지에서 이루어지는 참조를 통해 해당 영역의 메모리 공간을 늘일 경우 어느 정도 성능 향상이 있을지 관찰하여

그 결과를 영역의 크기 조절에 반영한다. 즉, 읽기 영역의 가상 페이지가 자주 참조될 경우 읽기 영역의 크기를 증가시켜 페이지 폴트를 줄인다. 쓰기 영역의 경우에도 마찬 가지이다. 또한, 영역의 크기 조절에는 가상 페이지에서의 적중 뿐 아니라 읽기 연산과 쓰기 연산의 비용 차이를 고려한다. 쓰기 연산의 비용이 읽기 연산의 약 8배이므로 쓰기 가상 페이지가 한 번 적중될 때마다 쓰기 영역 크기를 하나씩 증가시킨다면 읽기 가상 페이지는 여덟 번 적중될 때마다 읽기 영역 크기를 하나씩 증가시킨다. 가상 페이지는 영역의 크기 조절 기능뿐 아니라 해당 페이지의 과거 참조 기록을 더 오랫동안 파악할 수 있어 개별 페이지의 참조 가능성 예측에도 기여한다.

메모리 내의 전체 페이지 프레임 수가 S개라고 하면 읽기 영역에 할당된 실제 페이지 수와 가상 페이지 수의 합이 S개가 되도록 유지한다. 즉, 읽기 영역의 실제 페이지 수를 하나 증가(감소)시킬 때에는 읽기 영역의 가상 페이지 수를 하나 감소(증가)시켜 총 개수를 S개로 유지한다. 쓰기 영역의 경우에도 마찬가지이다. 이는 각 영역별로 할당된 실제 페이지와 가상 페이지의 수를 합해서 S개만 유지하면 전체 메모리 크기만큼의 실제 페이지를 할당했을 경우의 메모리 적중률을 예측할 수 있기 때문이다. 그러면 가상 페이지는 그림 7과 같이 읽기 영역과 쓰기 영역을 통틀어 총 S개만 사용하면 된다. 이는 두 영역에 할당된 실제 페이지 수의 합이 S개이기 때문이다. 한편, 실제 페이지 수만큼 가상 페이지 수를 유지하는 것은 페이지 하나의 크기가 일반적으로 4KB임에 비해 가상 페이지 정보는 20byte(포인터와 페이지 주소 정보) 정도에 불과하여 메모리의 추가적인 오버헤드가 크지 않음이 알려져 있다[8-11].

그림 7은 읽기 영역과 쓰기 영역의 개념적인 모습을 보여주고 있다. 한편, 실제 메모리 운영에 있어서는 읽기 영역과 쓰기 영역에 동일한 페이지 정보가 중복해서 저장될 수 있다. 예를 들어 최근에 읽기와 쓰기가 동시

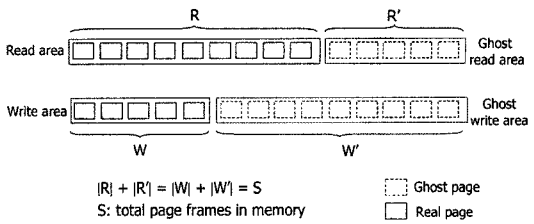


그림 7 가상 페이지를 통한 읽기 영역과 쓰기 영역의 크기 조절

에 발생한 페이지는 읽기 영역 R과 쓰기 영역 W에 그 페이지의 정보를 동시에 유지하고 있게 된다. 물론 메모리 상에 해당 페이지의 내용은 하나만 유지하고 있지만 읽기와 쓰기 영역에서는 그 페이지의 메타 정보를 각각 보유하고 있어야 두 영역이 독자적으로 자신에게 의미 있는 페이지를 선별할 수 있고 가상 영역 또한 정확한 정보를 제공할 수 있기 때문이다. 따라서, |R|+|W|의 값이 사실상은 페이지의 개수인 S개 이상이 될 수 있다. 이 경우 가상 페이지의 수는 더 적게 유지된다. 영역별 중복의 원리에 의해 읽기 영역과 쓰기 영역에서 모두 삭제된 경우에만 해당 페이지를 물리적인 메모리에서 교체하게 된다.

4.2 알고리즘의 상세 설명

그림 8은 CRAW 알고리즘의 대략적인 구조를 보여주고 있다. 메모리에 존재하는 페이지들은 읽기 영역 R과 쓰기 영역 W로 나뉘어 관리되며, 이들 두 영역에서 쫓겨난 페이지들은 각각 읽기 가상 영역 R'과 쓰기 가상 영역 W'에 일정 기간 동안 존재하게 된다. 읽기 영역에서 페이지 교체가 필요한 경우에는 클럭 알고리즘에 의해 헤드(head)가 가리키는 페이지들을 시계 방향으로 스캔하여 읽기 비트(read bit)가 1인 경우 0으로 클리어한 후 건너뛰고 0인 경우 교체한다. 쓰기 영역에서는 읽기 비트 대신 쓰기 비트(write bit)를 활용한다. 이들 비트는 페이지 참조 혹은 수정이 이루어질 때 대

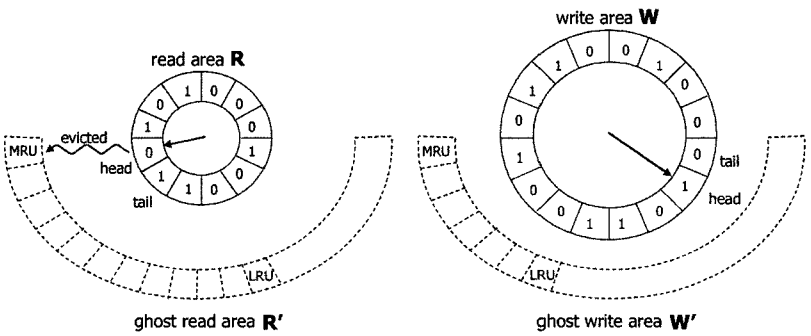


그림 8 CRAW 알고리즘의 대략적인 구조

부분의 시스템에서 하드웨어적으로 세팅되는 접근 비트(access bit) 및 수정 비트(dirty bit)와 유사한 의미를 갖는다.

읽기 영역 및 쓰기 영역에서 교체된 페이지는 각각 읽기 가상 영역과 쓰기 가상 영역으로 이동한다. 가상 영역에서 페이지 교체가 필요한 경우에는 LRU 순서에 의해 가장 오래 전에 참조된 페이지가 삭제된다.

한편, 쓰기 참조는 시간지역성 뿐 아니라 쓰기 연산의 빈도 역시 참조 가능성에 영향을 미치므로 쓰기 영역의 내부 구조는 그림 9와 같이 이원화된 관리가 이루어진다. 쓰기 영역 W는 쓰기 시간지역성 영역 W1과 쓰기 빈도 영역 W2로 나누어지며, 이들을 위한 가상 영역도 별도로 관리된다. 또한, W1과 W2의 크기 조절도 R과 W의 크기 조절과 마찬가지로 각 가상 영역에서의 참조 시 이루어진다. W1에는 쓰기 비트가 세팅된 것을 처음 발견한 페이지들을 보관하며, W2에는 쓰기 비트가 세팅된 것을 두 번 이상 발견한 페이지들을 보관한다. 한편, 쓰기 연산이 이루어질 때 그 페이지를 W1 또는 W2로 곧바로 이동시킬 수 있는 것은 아니다. (읽기 연산의 경우도 마찬가지이다.) 이는 W1 또는 W2에 존재하지 않는 페이지라 하더라도 그 페이지가 R에 이미 존재할 경우 쓰기 연산 시 페이지 폴트가 발생하지 않으므로 커널에 CPU 제어권이 곧바로 넘어오지 않아 페이지 이동이 즉시 이루어질 수 없기 때문이다. 이 경우 해당 페이지는 쓰기 비트가 세팅된 채로 읽기 영역에만 존재하게 된다. 이 페이지가 쓰기 영역에 포함되는 시점은 페이지 폴트가 발생하여 읽기 영역에서 페이지 교체 대상을 찾기 위해 페이지들을 스캔하다가 쓰기 비트가 세팅된 것이 발견되는 시점이다.

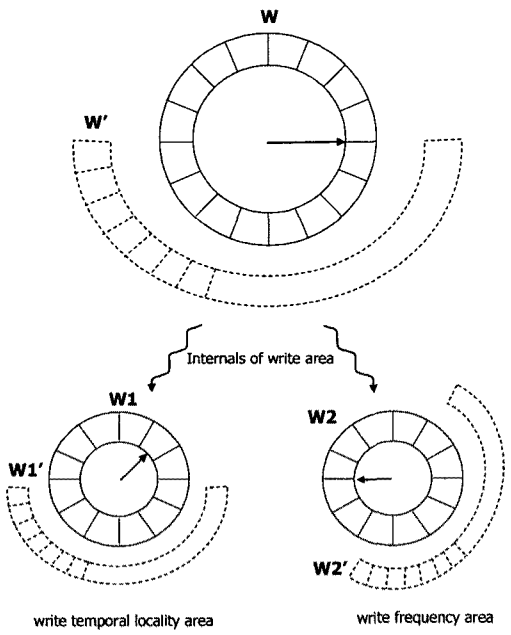


그림 9 시간지역성과 쓰기 빈도를 모두 고려하는 쓰기 영역의 내부 구조

이제 CRAW 알고리즘에 대해 조금 더 상세히 설명하겠다. 지금부터 설명하는 내용은 그림 10에 있는 CRAW의 의사코드(pseudocode)에 기반한 것이다. 페이지 요청 시 해당 페이지가 이미 메모리에 존재하는 경우, CRAW는 비트 세팅만 수행할 뿐 자료구조나 리스트 변경 등의 소프트웨어적인 업무는 수행하지 않는다. 읽기 참조일 경우 해당 페이지의 읽기 비트를, 쓰기 참조일 경우 쓰기 비트를 각각 세팅한다. 요청 페이지가 메모리에 존재하지 않아 페이지 폴트가 발생한 경우 CRAW는 우선 메모리 내에 빈 페이지 프레임이 존재하는지 확인한다. 빈 페이지 프레임이 존재하지 않는 경우 페이지 교체 모듈인 replace()를 호출하여 빈 페이지 프레임을 확보한 후 확보된 프레임에 요청된 페이지의 내용을 저장한다. 그런 다음 연산에 따라 페이지를 읽기 영역 또는 쓰기 영역에 연결하고 가상 영역에서의 적중 여부에 따라 필요할 경우 각 영역의 크기를 조절한다.

만약 페이지 폴트가 읽기 요청에 의해 발생했다면 페이지를 읽기 영역의 tail 부분에 연결한다. 이 때, 페이지의 메타 정보가 R'에 이미 존재했다면 이를 삭제하고 필요시 읽기 영역의 크기를 증가시킨다. 끝으로 읽기 영역의 크기가 증가했다면 읽기 가상 영역의 크기를 감소시켜 영역 간 크기의 균형을 유지한다.

만약 페이지 폴트가 쓰기 요청에 의해 발생했다면 크게 세 가지 경우로 나누어볼 수 있다. 첫째, 이 페이지의 정보가 W1'에 이미 존재할 경우 이를 삭제하고 페이지를 W2의 tail로 이동시킨다. 이 경우 쓰기 가상 영역 W1'에서 적중이 발생했으므로 쓰기 영역 W1의 크기를 증가시키고 필요할 경우 가상 영역을 포함한 다른 영역들의 크기를 조절한다. 둘째, 페이지의 정보가 W2'에 이미 존재할 경우 이를 삭제하고 페이지를 W2의 tail로 이동시킨다. 이 경우 쓰기 가상 영역 W2'에서 적중이 발생했으므로 쓰기 영역 W2의 크기를 증가시키고 필요할 경우 가상 영역을 포함한 다른 영역들의 크기를 조절한다. 셋째, 페이지의 정보가 쓰기 가상 영역 어디에도 존재하지 않을 경우 이 페이지의 정보를 W1의 tail에 연결한다.

이제 빈 공간이 필요해서 페이지를 교체하는 방법에 대해 설명하겠다. 물리적인 메모리에서 페이지를 교체하기 위해서는 그 페이지가 읽기 영역 R과 쓰기 영역 W1, W2에서 모두 교체된 경우에만 가능하다. 따라서 페이지 교체 모듈인 replace()를 한번 호출해서 빈 공간

이 마련되지 않을 수 있으며, 이 경우 빈 공간이 마련될 때까지 replace()를 반복 호출한다. replace()가 호출되면 먼저 R과 W1, W2 중 어느 영역에서 페이지를 교체할지 결정한다. 새로 요청된 페이지를 보관해야 할 영역에서 페이지를 교체하는 단순한 방법을 사용할 수 있지만, 영역 간 페이지의 중복 저장 등으로 좀 더 복잡한 교체 영역 결정 메커니즘이 필요하다. CRAW 알고리즘은 각 영역별로 현재 보관 중인 페이지의 수와 영역의 목표 크기의 비율을 구해 목표 크기에 비해 가장 많은 페이지를 보관 중인 영역을 페이지 교체 영역으로 결정한다. 교체 영역으로 결정된 영역에서는 클럭 알고리즘에 의해 헤드 위치를 시계 방향으로 스캔하며 교체 대상 페이지를 탐색한다. 이 과정에서 다른 연산의 접근 비트가 세팅된 페이지가 발견되면 이 페이지를 해당 연산의 영역으로 중복 저장시킨다. 예를 들어 교체 대상 영역이 읽기 영역 R인 경우 페이지의 쓰기 비트가 1로 세팅된 페이지가 발견되었는데 이 페이지가 쓰기 영역에 존재하지 않는 페이지이면 쓰기 비트를 0으로 클리어한 후 이 페이지를 쓰기 영역 W1의 tail에 연결시킨다. 물론, 이 과정에서 읽기 비트가 1인 페이지가 발견되면 이를 0으로 클리어한다. 읽기 비트가 0인 페이지가 발견되면 이를 읽기 가상 영역으로 강등시킨 후 교체한다. 교체 대상 영역이 W1, W2인 경우에도 같은 방법으로 동작한다. 이에 대한 상세한 알고리즘은 그림 10에 의사코드 형태로 기술하고 있다.

```
S is memory size and SR, SW1, SW2 are target sizes of R, W1 and W2, respectively;
Initially SR=S/8, SW1=(S-SR)/2, SW2=(S-SR)/2;

CRAW(page p, operation op)   /* p is requested page */
{   if(p is in memory)
    {   if(op is read) read_bit(p)=1;
        else write_bit(p)=1;
    }
    else /* page fault */
    {   while(no free page in memory) replace();
        memory_add(p, op);
        adjust_ghost_size();
    }
}

memory_add(page p, operation op)
{   if(op==read)
    {   if(p∈R')
        {   remove p from R';
            if (pages in R' has accessed eight times)
            {   SR = min(SR+1, S); /* increase the target size of R */
                SW1 = max(SW1-0.5, 0); /* decrease the target size of W1 */
                SW2 = max(SW2-0.5, 0); /* decrease the target size of W2 */
            }
        }
        insert p at the tail of R;
        read_bit(p)=0;
    }
    else  /* op should be write */
    {   if(p∈W1')
        {   remove p from W1' and insert p at the tail of W2;
            SW1 = min(SW1+1, S);  /* increase the target size of W1 */
            SR = max(SR-1, 0);  /* decrease the target size of R */
        }
        else if(p∈W2')
        {   remove p from W2' and insert p at the tail of W2;
            SW2 = min(SW2+1, S);  /* increase the target size of W2 */
            SR = max(SR-1, 0);  /* decrease the target size of R */
        }
        else  insert p at the tail of W1;
        write_bit(p)=0;
    }
}

adjust_ghost_size()
{   while(|R|+|R'|>S and |R'|>0)
        remove the LRU page in R';
    while(|W1|+|W1'|+|W2|+|W2'|>S and |W1'|+|W2'|>0)
        remove the LRU page in W1' or W2' in turns;
}
```

그림 10 CRAW 알고리즘의 의사코드

```
replace()
{   r=|R|/SR ; w1=|W1|/SW1 ; w2=|W2|/SW2 ; /* ratio of current size and target size */
    if(r ≥ w1 and r ≥ w2)   /* replace from R */
    {   while()
        {   p = head of R;
            head of R points to the next page;
            if(write_bit(p)==1 and p∉W1∪W2)
            {   insert p at the tail of W1;   write_bit(p)=0;
            }
            if(read_bit(p)==1)
            {   read_bit(p)=0;  move p to the tail of R;
            }
            else    /* read_bit(p) should be 0 */
            {   replace p from R and insert p to MRU position of R';
                if(p∉W1∪W2)  return(p);   /* found free page */
                else return(NULL);
            }
        }
    }
    else if(w1 ≥ r and w1 ≥ w2)   /* replace from W1 */
    {   while()
        {   p = head of W1;
            head of W1 points to the next page;
            if(read_bit(p)==1 and p∉R)
            {   insert p at the tail of R;        read_bit(p)=0;
            }
            if(write_bit(p)==1)
            {   write_bit(p)=0;          move p to the tail of W2;
            }
            else    /*  write_bit(p) should be 0 */
            {   replace p from W1 and insert p to MRU position of W1';
                if(p∉R) return(p);   /* found free page */
                else return(NULL);
            }
        }
    }
    else if(w2 ≥ r and w2 ≥ w1) /* replace from W2 */
    {   while()
        {   p = head of W2;
            head of W2 points to the next page;
            if(read_bit(p)==1 and p∉R)
            {   insert p at the tail of R;        read_bit(p)=0;
            }
            if(write_bit(p)==1)
            {   write_bit(p)=0;          move p to the tail of W2;
            }
            else    /*  write_bit(p) should be 0 */
            {   replace p from W2 and insert p to MRU position of W2';
                if(p∉R) return(p);   /* found free page */
                else return NULL;
            }
        }
    }
}
```

그림 10 CRAW 알고리즘의 의사코드 (계속)

5. 실 험

본 장에서는 제안하는 알고리즘의 성능 평가를 위해 수행한 실험 및 결과에 대해 기술한다. 성능 검증을 위해 NAND 플래시메모리를 스왑 장치로 하는 가상메모리 시스템의 페이지 교체 모듈을 시뮬레이터로 구성하였다. 메모리의 페이지 프레임 크기는 리눅스를 비롯한 대부분의 운영체제에서 사용하고 있는 4KB로 하였으며, NAND 플래시메모리는 페이지 하나의 크기가 2KB이고 블록 하나가 64개의 페이지로 구성되는 대블록 플래시메모리를 가정하였다.

5.1 실험 환경

실험에 사용한 트레이스는 리눅스 상에서 프로세스를 실제로 수행시켰을 때 발생하는 가상메모리 참조를 Val

reference type	virtual address	access size (byte)
readi	0x04000BE0	2
write	0xBEFFFACC	4
readi	0x04000C30	1
write	0xBEFFFABC	4
readd	0x0401582C	4
:	:	:

그림 11 실험에 사용한 가상메모리 참조 트레이스의 형태

grind 3.2.3 버전의 cachegrind 툴을 수정하여 추출하였다[12,13]. 추출한 트레이스의 메모리 참조 요청 종류는 명령 읽기(instruction read), 데이터 읽기(data read), 데이터 쓰기(data write)로 나누어 볼 수 있으며 트레이스의 형태는 그림 11과 같다. 실험에 사용한 트레이스는 총 여섯 종류로 리눅스의 Xwindow 상에서 수행한 mp3 음악 파일 재생 프로그램인 xmms, 이미지 편집 프로그램인 gqview, 문서 작성 프로그램인 gedit, 게임 프로그램인 freecell, pdf 파일 뷰어 프로그램인 kghostview, 리눅스 커널 컴파일 및 빌드를 수행하는 kbuild 작업 시 각 프로세스의 가상메모리 참조 트레이스이다. 각 트레이스의 특징은 표 2와 같다.

CRAW 알고리즘과의 성능 비교를 위해 가상메모리 시스템에서 가장 널리 사용되는 CLOCK 알고리즘, 가상 페이지의 개념을 도입하여 CLOCK 알고리즘의 성능을 개선한 CAR 알고리즘[14], 그리고 NAND 플래시메모리의 특성을 고려한 CFLRU 알고리즘[3]을 함께 실험하였다. CFLRU 알고리즘의 경우 페이지 참조시 마다 LRU 리스트의 조작이 이루어져야 하는 등 가상메모리 시스템에 구현되기에 현실적인 어려움이 있어 이를 CLOCK 알고리즘 형태로 근사시킨 CFCLOCK 알고리즘으로 실험하였다. CFCLOCK은 페이지 교체가 필요할 때마다 CLOCK의 현재 위치에서 윈도우 크기만큼의 페이지를 스캔하면서 접근 비트와 수정 비트가 모두 0인 페이지가 존재하면 교체한다. 그런 페이지가 존재하지 않으면 윈도우 내에서 접근 비트가 0이면서 수정 비트가 1인 페이지를 교체한다. 그런 페이지가 존재하지 않으면 CLOCK의 다음 위치에서 접근 비트가 0인 페이지를 교체한다. 실험에서 CFCLOCK의 윈도우 크기는 CFLRU 논문의 가상메모리 시뮬레이션 부분에서 사용한 설정과 같은 비율인 전체 메모리 크기의 1/3로 설정하였다[3].

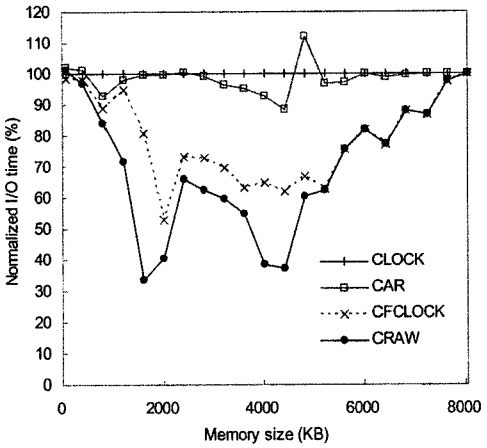
5.3 실험 결과

그림 12는 표 2에서 설명한 6개의 트레이스에 대해 메모리 용량을 각 트레이스별 최대 메모리 사용량의 1%에서 100%까지 변화시키며 알고리즘별 성능을 보여준 그래프이다. 100% 메모리 크기는 해당 프로그램의 메모리 사용량 전체를 한꺼번에 할당하는 경우로 페이지 교체가 필요하지 않아 모든 알고리즘의 성능이 동일한 값으로 수렴한다. 그래프에서 y축은 각 알고리즘별 총 I/O 시간을 CLOCK 알고리즘의 성능으로 정규화하여 상대적인 입출력 소요 시간을 나타낸 것이다. 그림 12에서 보는 것과 같이 CRAW 알고리즘은 다양한 응용 프로그램과 메모리 구간에서 기존 알고리즘들에 비해 향상된 성능을 나타내었다. 실제 시스템에서 널리 활용되고 있는 CLOCK 알고리즘에 비해서는 평균 23.9%, 최대 66.5%의 성능 향상을 보였으며, CAR 알고리즘에 비해서는 25-66%의 성능 향상을 나타내었다. 또한, NAND 플래시메모리의 물리적 특성을 고려하여 고비용의 쓰기 연산 횟수를 절약한다는 점에서 CRAW 알고리즘과 동일한 목표를 갖는 CFLRU 알고리즘에 비해서는 16-58%까지 입출력 수행 시간을 향상시켰다.

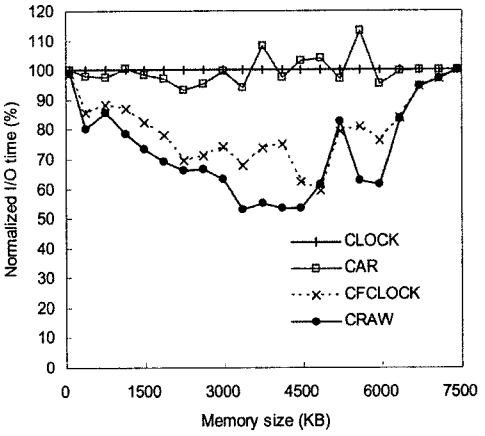
CFCLOCK의 경우 더티 페이지의 교체를 최대한 지연시켜 고비용의 쓰기 연산을 줄이지만 freecell, gedit 등 일부 읽기 중심 트레이스의 소용량 메모리 구간에서

표 2 트레이스별 메모리 사용량 및 참조 횟수

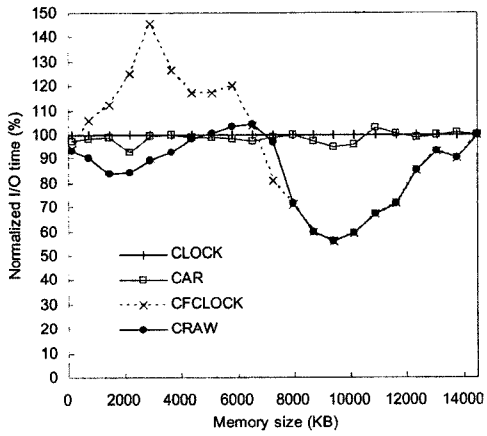
워크로드	메모리 사용량 (KB)	메모리 참조 횟수			
		전체 횟수	명령 읽기	데이터 읽기	데이터 쓰기
xmms	8,050	1,168,939	65,048	125,649	978,242
			읽기 : 쓰기 = 1 : 5.13		
gqview	7,430	610,685	93,242	172,044	345,399
			읽기 : 쓰기 = 1 : 1.30		
gedit	14,460	1,733,763	649,500	951,441	132,822
			읽기 : 쓰기 = 12.05 : 1		
freecell	10,080	490,175	114,233	315,902	60,040
			읽기 : 쓰기 = 7.16 : 1		
kghostview	17,390	1,546,135	380,609	1,061,986	103,540
			읽기 : 쓰기 = 13.93 : 1		
kbuild	1,232	13,435	2,253	2,393	8,789
			읽기 : 쓰기 = 1 : 1.89		



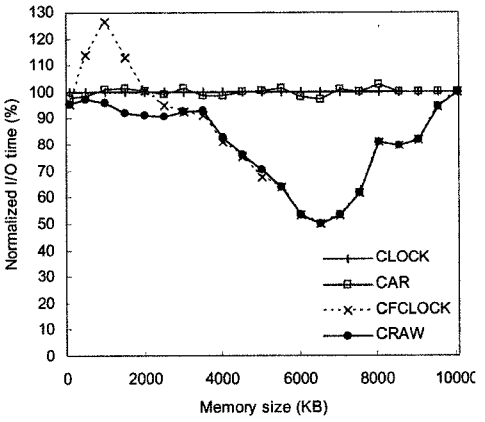
(a) xmms



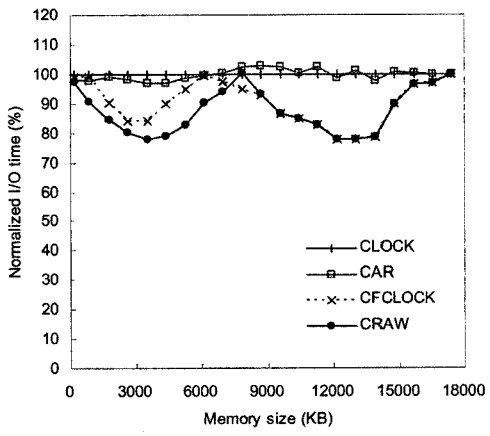
(b) gqview



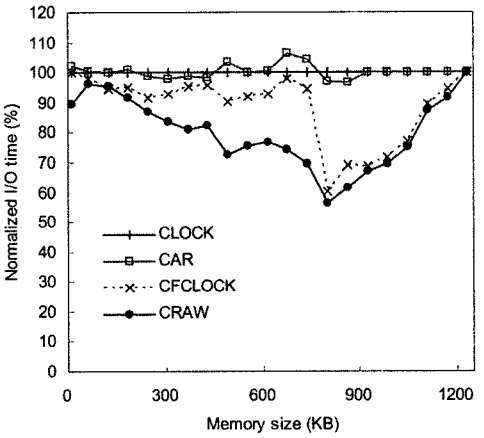
(c) gedit



(d) freecell

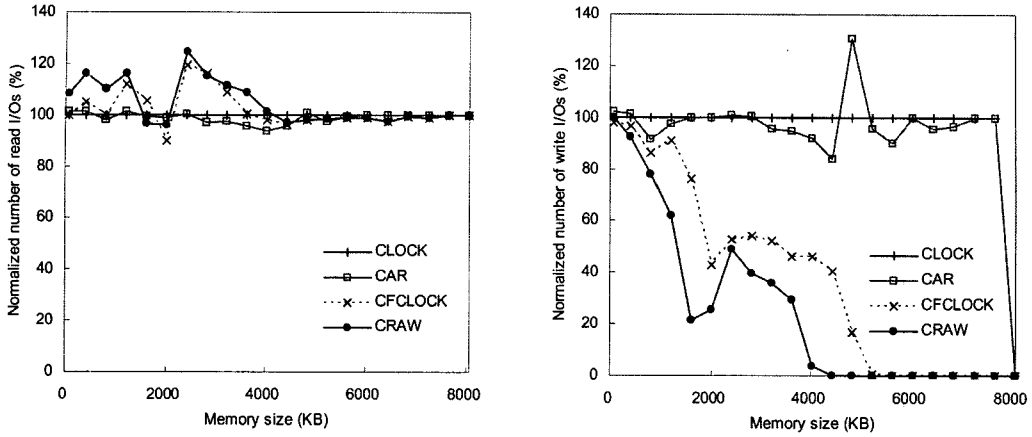


(e) kghostview

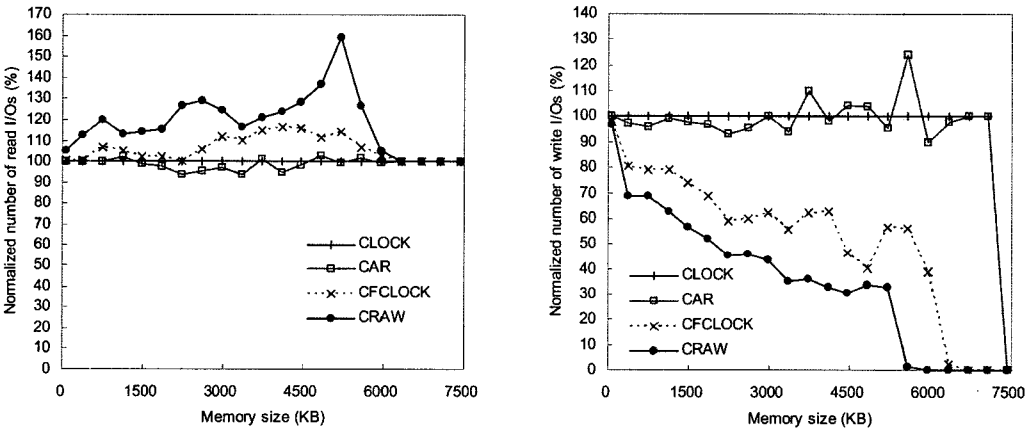


(f) kbuild

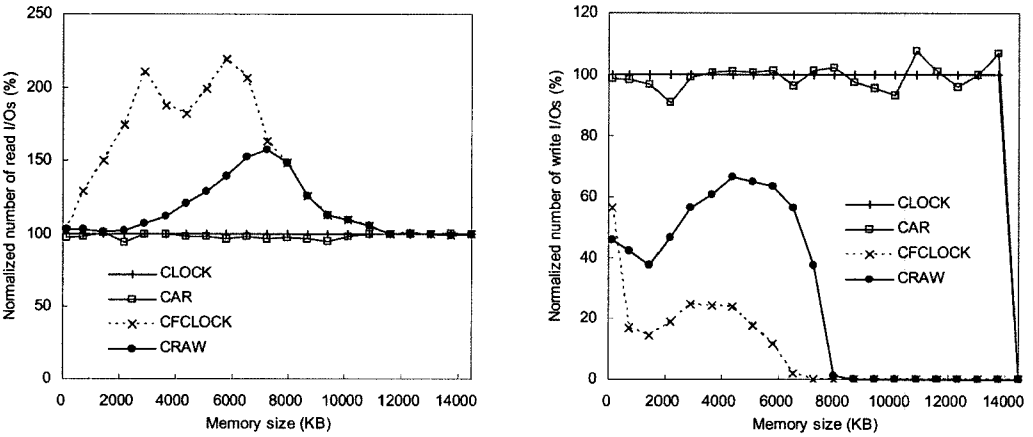
그림 12 NAND 플래시메모리의 전체 입출력 연산 소요 시간 비교



(a) xmms



(b) gqview



(c) gedit

그림 13 NAND 플래시메모리의 읽기 연산 및 쓰기 연산 횟수

CLOCK과 CAR에 비해 낮은 성능을 나타내었다. 이는 CFCLOCK이 워크로드 패턴 변화에 적응적으로 동작하지 못하여 읽기 연산만 발생하는 대용량의 클린 페이지들을 메모리에 보관하지 못했기 때문으로 분석된다. 읽기 참조는 시간지역성이 강하게 나타나는 요청으로, 이들을 위한 메모리 공간이 어느 정도 확보가 된다면 CLOCK 방식으로 동작했을 때 높은 메모리적중률을 보이게 된다. 그런데, CFCLOCK에서는 과도한 더티 페이지의 보호로 인해 이러한 다수의 읽기 참조 페이지를 위한 메모리 공간을 확보하지 못하게 된다. 이로 인해 고비용의 쓰기 연산 횟수를 줄였음에도 전체 입출력 시간 면에서는 낮은 성능을 나타내었다. 반면, CRAW 알고리즘은 워크로드의 패턴 변화 및 메모리 용량에 적응적으로 동작하여 읽기/쓰기 참조의 비율 변화 및 메모리 용량의 변화 등과 관계없이 일관성 있게 좋은 성능을 나타내었다.

그림 13은 xmms, gqview, gedit 워크로드가 수행되는 동안 플래시메모리에서 일어난 읽기 및 쓰기 연산의 횟수를 나타낸 그래프이다. 이를 통해, 워크로드의 특성 및 주어진 메모리 공간의 크기에 따라 CRAW 알고리즘이 읽기 연산과 쓰기 연산의 비용을 어떤 식으로 조절하여, 그림 12와 같은 전체 입출력 소요 시간 절약을 가져왔는지를 확인할 수 있다. 그림 13의 xmms와 gqview는 쓰기 참조의 비율이 읽기 참조에 비해 높게 나타나는 트레이스로, 많은 양의 쓰기 참조로 인해 더티 페이지가 다량 발생하는 특징을 가지고 있다. CRAW는 이와 같은 환경에서 쓰기 영역의 크기를 확장하여 더티 페이지의 교체를 적극적으로 지연함으로써, 쓰기 입출력 연산의 횟수를 크게 줄여 전체 입출력 시간을 절약하였다. 한편, gedit 트레이스는 읽기 참조의 비율이 쓰기 참조에 비해서 높은 트레이스이다. 이러한 트레이스 하에서 CRAW는 고비용의 쓰기 연산 횟수를 줄이는 동시에 읽기 연산이 빈번히 발생하는 페이지를 메모리에 유지해서 전체 입출력 시간 향상을 가져왔다. CFCLOCK은 쓰기 연산을 크게 줄였지만 메모리 크기가 작은 경우 CLOCK이나 CAR에 비해 읽기 연산의 성능이 좋지 않게 나타났다. 특히, 그림 13(c)를 보면 CFCLOCK의 읽기 입출력 횟수가 CLOCK의 2배에 이름을 확인할 수 있다.

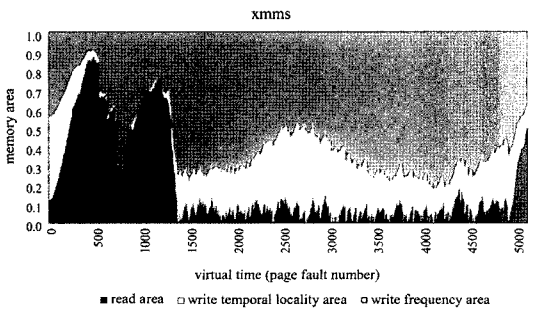


그림 14 시간에 따른 CRAW 알고리즘의 영역별 페이지 할당량 변화

그림 14는 시간에 따른 CRAW 알고리즘의 영역별 크기 변화를 보여주고 있다. 매 페이지 폴트 발생 시점에 읽기 영역과 둘로 구분된 쓰기 영역 각각의 목표 크기(target size)를 조사하였으며, 변화하는 읽기/쓰기 참조 패턴에 맞추어 동적으로 메모리 공간이 할당되고 있음을 확인할 수 있다.

6. 결 론

본 논문에서는 비용이 높은 플래시메모리의 쓰기 연산 특성을 규명하기 위해 가상메모리의 읽기/쓰기 참조 패턴을 독립적으로 분석하고, 시간지역성이 강한 읽기 참조와 달리 쓰기 참조의 경우 시간지역성의 순위 역전 현상이 발생함을 발견하였다. 이에 근거하여 본 논문은 쓰기의 재참조 성향 예측을 위해 시간지역성뿐 아니라 쓰기 연산의 빈도를 함께 고려하는 새로운 페이지 교체 알고리즘인 CRAW 알고리즘을 제안하였다. CRAW는 연산별 I/O 비용을 고려해서 메모리 공간을 읽기 연산과 쓰기 연산에 독립적으로 할당하고 참조 패턴의 변화에 적응해 할당 공간을 동적으로 변화시킨다. 이를 통해 NAND 플래시메모리를 스왑 장치로 사용하는 가상메모리 시스템의 전체 입출력 수행 시간을 기존 알고리즘들에 비해 크게 향상시켰다. 특히, CRAW 알고리즘은 CLOCK, CAR, CFLRU 등 대표적인 페이지 교체 알고리즘들에 비해 20-66%의 꾸준한 성능 향상을 나타내었다. CRAW 알고리즘은 시간 오버헤드가 매우 적고 LRU처럼 페이지 접근 시마다 리스트 조작이 이루어지는 알고리즘과 달리 메모리에서 적중된 페이지에 대해서는 하드웨어에 의한 비트 세팅만 일어날 뿐 아무런 부가 연산이 이루어지지 않아 가상메모리 시스템에서 사용될 수 있는 최적의 조건을 갖추고 있다. 또한 파라미터 튜닝이 자동적으로 이루어져 파라미터 설정에 따라 성능 차이를 보이는 연구들과 차별화된다.

CRAW 알고리즘의 구현은 읽기 비트와 쓰기 비트를 필요로 한다. ARM 아키텍처에서는 접근 비트나 수정 비트가 제공되지 않아 비트 셋팅을 고의적인 페이지 폴트 발생에 의해 소프트웨어적으로 처리하기 때문에 CRAW 알고리즘도 읽기 비트 및 쓰기 비트를 소프트웨어적인 방법으로 구현할 수 있다. 접근 비트와 수정 비트를 하드웨어적으로 지원하는 아키텍처에서는 읽기 비트와 쓰기 비트의 구현이 쉽지 않을 수 있다. 접근 비트는 읽기 연산이나 쓰기 연산 중 어느 연산이 발생하더라도 셋팅되므로 접근 비트를 통해 정확한 읽기 연산

정보를 추출해 내는 것은 쉬운 일이 아니다. 이 경우 읽기 영역과 쓰기 영역을 사용하는 대신에 읽기/쓰기를 모두 포함하는 총 참조 영역과 쓰기 영역을 두어 각각 접근 비트와 수정 비트로 관리하는 CRAW의 근사적인 구현을 생각해 볼 수 있다. CRAW 알고리즘이 읽기 영역과 쓰기 영역 간 페이지의 중복 저장을 허용하기 때문에 이와 같은 근사적인 구현 방식도 CRAW 알고리즘 자체의 동작 방식과 크게 다르지 않아 별 무리없이 동작할 수 있을 것으로 기대된다. 이는 2장에서 읽기 연산만의 시간지역성 그래프가 읽기와 쓰기 연산을 모두 포함한 총 참조에 의한 시간지역성 그래프와 매우 유사하다는 점과도 일관성 있는 근사 방식이라 할 수 있다.

참 고 문 헌

[1] C. Park, J. Seo, S. Bae, H. Kim, S. Kim, B. Kim, "A low-cost memory architecture with NAND XIP for mobile embedded systems," *Proceedings of CODES+ISSS*, 2003.

[2] C. Park, J. Kang, S. Park, J. Kim, "Energy-Aware Demand Paging on NAND Flash-based Embedded Storages," *Proceedings of International Symposium on Low Power Electronics and Design*, 2004.

[3] S. Park, D. Jung, J. Kang, J. Kim, J. Lee, "CFLRU: replacement algorithm for flash memory," *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, 2006.

[4] R. van Riel, "Page replacement in Linux 2.4 memory management," *Proceedings of the 2001 USENIX Annual Technical conference*, 2001.

[5] E. G. Coffman, P. J. Denning, Operating Systems Theory, Prentice-Hall, Ch.6, pp.241-283, 1973.

[6] http://www.samsung.com/Products/Semiconductor/NANDFlash/SLC_LargeBlock/4Gbit/K9F4G08U0A/ds_k9xxg08uxa_rev10.pdf

[7] D. Bovet, M. Cesati, "Understanding the Linux Kernel," O'Reilly, 2002.

[8] T. Johnson, D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," *Proceedings of the VLDB conference*, 1994.

[9] Y. Zhou, J. F. Philbin, "The multi-queue replacement algorithm for second level buffer caches," *Proceedings of the 2001 USENIX Annual Technical conference*, 2001.

[10] S. Jiang, X. Zhang, "LIRS: An efficient low interreference recency set replacement policy to improve buffer cache performance," *Proceedings of the ACM SIGMETRICS conference*, 2002.

[11] N. Megiddo, DS. Modha, "ARC: A Self-tuning, Low Overhead Replacement Cache," *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003.

[12] http://valgrind.org/

[13] N. Nethercote, J. Seward, "Valgrind: A Program Supervision Framework," *Electronic Notes in Theoretical Computer Science*, 2003.

[14] S. Bansal, DS. Modha, "CAR: Clock with Adaptive Replacement," *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

이 혜 정

2006년 이화여자대학교 컴퓨터학과 학사 2008년 이화여자대학교 컴퓨터정보통신공학과 석사. 2008년~현재 LG전자 전자기술원 연구원. 관심분야는 운영체제, 스토리지 시스템, 임베디드 시스템 등

반 효 경

1997년 서울대학교 계산통계학과 학사. 1999년 서울대학교 전산과학과 석사. 2002년 서울대학교 컴퓨터공학부 박사. 2002년~현재 이화여자대학교 컴퓨터공학과 교수 관심분야는 운영체제, 스토리지 시스템, 임베디드 시스템, 저전력 시스템, 캐싱 기법 등