선반입을 이용한 효율적인 버퍼 캐쉬 관리 알고리즘

(An Efficient Buffer Cache Management Algorithm based on Prefetching)

전 흥 석 † 노 삼 혁 ††
(H. Seok Jeon) (Sam H. Noh)

요 약 본 논문은 선반입에 기반한 디스크 버퍼 관리 알고리즘인 W^2R 알고리즘을 제안한다. W^2R 알고리즘은 어떤 블록을, 언제 선반입할 것인가를 결정하기 위한 복잡한 선반입 정책 대신, LRU-OBL 알고리즘의 접근 방법을 따라 현재 참조되는 블록의 논리적 다음 블록을 선반입한다. LRU-OBL 알고리즘과의 기본적인 차이점은 W^2R 알고리즘은 버퍼를 논리적으로 두개의 공간, 즉, Weighing Room과 Waiting Room으로 분할한다는 것이다. 참조되는 블록은 Weighing Room에 반입되고 선반입되는 논리적 다음 블록은 Waiting Room에 저장된다. 이렇게 함으로써, 무조건으로 참조되는 블록의 논리적 다음 블록을 선반입하는 LRU-OBL 정책의 단점을 해결한다. 구체적으로, 선반입되었으나 결코 참조되지 않을, 혹은 실제로 참조된다고 할지라도 교체될 블록보다 더 나중에 참조될 블록들을 위해 재 참조될 가능성이 있는 블록들을 교체하는 문제점들을 해결한다. W^2R 알고리즘은 트레이스 기반 시뮬레이션을 통해 버퍼 캐쉬 적중률을 측정한 결과 2Q 알고리즘에 비해서는 최고 23.19 %, LRU-OBL 알고리즘에 비해서는 최고 10.25 %의 성능향상을 나타낸다.

Abstract This paper proposes a prefetch-based disk buffer management algorithm, which we call W^2R (Weighing/Waiting Room). Instead of using elaborate prefetching schemes to decide which block to prefetch and when, we simply follow the LRU-OBL (One Block Lookahead) approach and prefetch *the logical next block along with the block that is being referenced. The basic difference is that the* W^2R algorithm logically partitions the buffer into two rooms, namely, the Weighing Room and the Waiting Room. The referenced, hence fetched block is placed in the Weighing Room, while the prefetched logical next block is placed in the Waiting Room. By so doing, we alleviate some inherent deficiencies of blindly prefetching the logical next block of a referenced block. Specifically, a prefetched block that is never used may replace a possibly valuable block and a prefetched block, though referenced in the future, may replace a block that is used earlier than itself. We show through trace driven simulation that for the workloads and the environments considered the W^2R algorithm improves the hit rate by a maximum of 23.19 percentage points compared to the 2Q algorithm and a maximum of 10.25 percentage points compared to the LRU-OBL algorithm.

1. 서 론

디스크 버퍼의 성능을 향상시키기 위해 많은 알고리즘들이 제안되어져 왔다. 그들 중 대표적으로 LRU, LRU-K [1], 2Q [2], 그리고 FBR [3] 알고리즘 등이 있다. 구체적으로 살펴보면, LRU 알고리즘은 교체 블록에 대한 결정을 블록의 가장 최근 참조 정보에 근거를 두는 반면, LRU-K 알고리즘은 여러 번, 즉, K번의 참조 정보를 고려한다. 이것은 교체 정책으로 하여금 각 블록의 중요성을 단 한번의 참조가 아닌 긴 시간에 걸친 관찰을 통해서 블록의 중요성을 판단하게 한다. 2Q

· 본 연구는 한국과학재단 특정 기초 연구 과제 (과제번호: 98-0102-09-01-3)의 지원에 의해 수행되었음.
† 학생회원 : 홍익대학교 전자계산학과
hsjeon@cs.hongik.ac.kr
†† 종신회원 : 홍익대학교 컴퓨터공학과 교수
noh@cs.hongik.ac.kr
논문접수 : 1999년 5월 26일
심사완료 : 2000년 3월 13일

알고리즘은 블록이 최소한 두 번 이상 참조된 후에야 캐쉬에 장시간 보전할 가치가 있다고 판단한다. 첫 번째 참조 시에 블록은 A1 큐라고 불리는 임시 공간에 저장된다. 그리고 두 번째 참조 시에야 비로소 Am 큐라고 불리는 캐쉬의 주요 부분으로 옮겨진다. 즉, 각 블록의 중요도가 증명된 후에야 비로소 캐쉬의 주요부분으로 저장되어 장시간 보존하는 것이다.

LRU-K 와 2Q 알고리즘들은 교체에 대한 결정을 기본적으로 블록 참조의 최근성에 두는 반면 FBR 알고리즘은 블록 참조의 최빈성에 의해 교체할 블록을 결정한다. 또한 Robinson과 Devarakonda는 연관 참조의 고려가 알고리즘의 효율성에 있어서 중요한 요소임을 보여준다.

LRU-K, 2Q, 그리고 FBR 알고리즘은 어떠한 형태의 선반입도 사용하지 않는다. 그런데 선반입을 교체 정책과 통합하여 사용한 연구 결과에 따르면 특정 작업 부하에 대해 버퍼 관리의 성능에 있어서 선반입이 상당한 성능향상을 가져옴을 보여준다 [4, 5].

선반입의 통합은 주로 시스템 분야에서 이루어져 왔으며 어떤 블록을 언제 선반입하느냐에 따라 크게 세 그룹으로 분류되어진다. 첫 번째 그룹은 응용 프로그램들의 과거 참조 형태를 유지한다 [6, 7]. 그러나 이러한 접근 방법은 부정확한 선반입과 과거 정보 유지를 위한 오버헤드로 인해 오히려 성능 감소를 가져 올 수 있다는 단점을 갖고 있다.

두 번째 그룹은 실행 이전에 응용 프로그램들로부터 힌트를 얻는다 [8, 9, 10, 11, 12]. 그러나 이러한 접근 방법은 정확한 선반입을 제공하긴 하지만 일반적인 작업 부하 환경에서는 아직 검증된 바가 없다. 물론 어떤 특정한 환경에서는 특정한 응용 프로그램에 대해 힌트를 얻을 수가 있지만 이러한 방법이 일반적인 버퍼 관리에 어떻게 적용될 수 있을까 는 아직 해결되지 않은 문제이다.

마지막 접근 방법은 응용 프로그램과 과거 참조에 대해서 어떠한 부가적인 정보도 요구하지 않는다. LRU-OBL 알고리즘은 [4, 5] 이러한 접근 방법을 사용하는 대표적인 알고리즘이다. 이 알고리즘은 단순히 참조된 블록의 논리적 다음 블록을 선반입한다. 이러한 단순한 알고리즘 임에도 불구하고 특정 작업 부하에 대해 80 %이상의 캐쉬 적중률 향상을 나타낸다 [5]. 이 알고리즘의 또 다른 장점은 구현하기가 쉽고, 선반입을 위하여 사용자나 시스템으로부터 어떠한 추가적인 오버헤드를 요구하지 않는다는 점이다.

이 논문에서 제안하는 W^2R 알고리즘은 LRU-OBL 알고리즘과 2Q 알고리즘을 확장하여 개선한 알고리즘이다. 이 정책은 어떤 블록을 언제 선반입해야 할 것인지에 관해 전혀 오버헤드가 없는 LRU-OBL 정책과 동일한 블록을 선반입한다. 그러나 이들 블록에 대한 관리는 전혀 다르다. W^2R 알고리즘은 버퍼를 크게 두개의 공간으로 분할한다. 하나는 참조된 블록들을 관리하기 위한 공간이고, 다른 하나는 선반입된 블록을 관리하기 위한 공간이다. 이 논문에서는 이러한 버퍼분할을 통하여 LRU-OBL 정책의 문제점을 해결하여 성능을 개선함을 보여준다. 트레이스에 기반한 시뮬레이션을 통해 캐쉬 적중률을 측정한 결과, W^2R 알고리즘은 2Q 알고리즘에 비해서는 최고 23.19 %, 그리고 LRU-OBL 알고리즘에 비해서는 최고 10.25 %의 성능 향상을 보여준다.

W^2R 알고리즘의 LRU-OBL 정책에 비한 또 다른 장점은 그 모듈성에 있다. LRU-OBL 알고리즘은 기본적으로 블록 교체와 선반입을 하나의 알고리즘으로 관리하지만, 제안된 W^2R 알고리즘의 구조는 이들 두 개념을 명확하게 구분하여 더 큰 성능 개선을 가져오는 교체 정책을 쉽게 적용할 수 있게 한다.

본 논문의 나머지 부분은 다음과 같이 구성되어 있다. 2 절에서는 본 논문에서 제안하는 W^2R 알고리즘에 대해 설명하며 3 절에서는 W^2R 알고리즘의 성능을 측정하기 위한 실험 환경 및 실험 결과를 서술한다. 4절에서는 W^2R 알고리즘의 모듈성에 관하여 서술한다. 5 절에서 본 연구의 결론을 맺고 향후 연구 과제를 제시한다.

2. 관련 연구

2.1 교체 정책

지금까지 연구되어온 버퍼 캐쉬 교체 정책들은 버퍼 캐쉬내의 블록들 중에서 교체할 블록을 선정하는 배경에 따라 크게 최근성 (Recency)을 근거로 하는 정책과 최빈성 (Frequency)을 근거로 하는 정책들로 분류할 수 있다. 최근성을 근거로 하는 정책에는 Least Recently Used (LRU) 교체 정책, LRU-K 교체 정책, Segmented LRU (SLRU) 교체 정책, 그리고 2Q 교체 정책 등이 있으며 최빈성을 근거로 하는 정책에는 대표적으로 Least Frequently Used (LFU) 정책과 Frequency Based Replacement (FBR) 정책 등이 있다. 이중에서 먼저 최근성을 근거로 하는 정책들을 살펴보면 다음과 같다.

LRU 교체 정책은 최근성을 바탕으로 하는 가장 널리 사용되는 정책으로서 버퍼 캐쉬 안의 블록들 중에서 가

장 오래 전에 사용되었던 블록을 교체한다. 이는 파일 참조의 시간적 지역성으로 인해 가장 오래 전에 사용되었던 블록이 재 참조될 확률이 가장 적다는 전제 때문이다. LRU정책은 매우 단순하여 쉽게 구현될 수 있는 장점을 가지고 있다. 그러나 LRU 정책은 버퍼 캐쉬 안에 유지되는 블록들의 참조된 순서 이외에는 어떠한 정보도 유지하지 않기 때문에 빈번하게 참조되는 블록들과 그렇지 못한 블록들을 구별하지 못하게 된다. 따라서 LRU 정책은 빈번하게 참조되지 않는 블록들도 버퍼 캐쉬 안에 장시간 유지하는 비효율성을 가지고 있다.

LRU-K 교체 정책 [1] 은 LRU 교체 정책이 빈번하게 참조되는 블록과 빈번하게 참조되지 않는 블록을 구별하지 못하는 문제점을 해결하기 위해 제안된 정책이다. LRU-K 교체 정책은 최근 K 번의 참조로부터 추정된 참조 조밀성에 근거하여 교체 블록을 결정한다. 즉, 버퍼 캐쉬 안의 블록들 중에서 최근 K 번째의 참조 시간이 가장 오래 전인 블록을 교체한다. 최근의 K 번의 참조 시간을 고려하기 때문에 LRU-K 교체 정책은 빈번하게 참조되는 블록들과 그렇지 못한 블록들을 구별하게 되어 참조 횟수가 적은 블록들을 버퍼 캐쉬에서 빨리 교체할 수 있게 된다. 그러나 LRU-K 교체 정책은 교체 블록을 결정하기 위하여 모든 참조된 블록들에 대한 최근 K 번의 참조 시간 정보를 유지해야 한다. 그러므로 많은 양의 메모리 공간과 처리 시간을 요구하게 되는 단점을 가지고 있다.

SLRU 교체 정책 [13] 에서는 캐쉬를 Probationary segment와 Protection segment로 분할하여 최초 참조되는 블록들을 Probationary segment에 반입하였다가 재 참조되면 Protection segment로 이동시켜 캐쉬에 장시간 유지한다. SLRU 교체 정책에서는 새로이 캐쉬에 반입되는 블록을 위해서 Probationary segment내에서 가장 오래 전에 사용되었던 블록을 교체한다. 캐쉬를 분할하는 이유는 순차적으로 오직 한 번만 참조된 블록들을 가려내기 위한 것이다.

2Q 교체 정책 [2] 은 SLRU 교체 정책과 유사한 정책이다. 2Q 정책에서는 버퍼 캐쉬를 A1큐와 Am 큐로 분할한다. 최초 참조되는 블록들을 A1큐에 반입하였다가 디스크 블록에 대한 포인터만 저장한 채 방출한 후 재 참조되었을 때에야 비로소 Am 큐로 반입하여 버퍼 캐쉬에 장시간 유지된다. A1 큐 내에서 일정 기간 내에 재 참조되지 않을 경우에는 빈번하게 참조되지 않는 블록으로 간주되어 조기에 교체되어 진다. 또한 2Q 정책에서는 연관 참조의 문제를 해결하기 위하여 A1 큐를 A1in 큐와 A1out 큐의 두 영역으로 분할한다. MRU 블록을 포함하는 A1in 큐에서 재 참조되는 블록들은 연관 참조되는 블록으로 간주되어 Am 큐로 이동하지 않으며 A1out 큐에서 재 참조될 경우에만 Am큐로 이동된다. 2Q 교체 정책은 LRU-K 정책과 비슷한 동기에서 제안된 정책이나 LRU-K 교체 정책의 시간 복잡도가 O(log n)(n은 버퍼 캐쉬 안의 블록의 수)임에 비해 2Q 정책의 시간 복잡도는 O(1)이라는 장점이 있다.

최빈성을 근거로 하는 교체 정책에는 대표적으로 LFU 교체 정책이 있다. LFU 정책에서는 버퍼 캐쉬 안의 모든 블록들의 참조 횟수에 관한 정보를 유지한다. 이를 바탕으로 새로이 반입되는 블록을 위해 버퍼 캐쉬 안의 모든 블록들 중에서 참조 횟수가 가장 적은 블록을 교체한다.

FBR 정책 [3] 은 LFU 정책을 좀더 보완한 정책이다. FBR 정책에서는 버퍼 캐쉬를 논리적으로 new section, middle section, 그리고 old section 등의 세 개의 영역으로 분할한다. New section에는 최근에 참조되는 블록들을 유지하게 되는데 new section에서 재 참조될 경우에는 연관 참조로 간주되어 참조 횟수를 증가하지 않는다. Old section에서 재 참조되거나, new section 과 old section의 중간에 위치한 middle section에서 재 참조될 경우에만 참조 횟수를 증가시킨다. 재 참조된 블록들은 다시 new section으로 옮겨진다. 교체 블록은 old section에서 가장 적은 참조 횟수의 블록을 선택한다. middle section은 상대적으로 빈번하게 참조되는 블록들이 old section에서 교체되지 않도록 참조 횟수를 증가시키는 역할을 수행한다.

지금 까지 언급한 여러 교체 정책들은 공통적으로 버퍼 캐쉬로의 반입의 대상을 참조된 블록에 한정시키는 단점을 가지고 있다. 이것은 모든 블록의 최초 참조 시 항상 버퍼 캐쉬 비적중이 되어 디스크 지연을 발생시키게 된다는 것이다. 이러한 점을 보완하기 위해 선반입 기법이 도입된다. 다음절에서 선반입을 고려한 기존의 버퍼캐쉬 관리 정책들을 살펴본다.

2.2 선반입 정책

지금까지 연구되어진 선반입 정책들은 선반입할 블록을 결정하는 방법에 따라 미래의 참조 정보를 기반으로 하는 정책과 과거의 참조 정보를 바탕으로 하는 정책, 그리고 외부로부터의 어떠한 미래 참조에 관한 힌트도 없고 과거의 참조 정보도 유지하지 않는 정책 등으로 분류되어질 수 있다.

Cao et al. [8, 9] 과 Patterson et al. [12] 등은 미래의 참조 정보를 기반으로 하는 선반입 정책을 제안하였다. 이들 정책들에서는 프로그래머가 파일 시스템에

응용 프로그램의 미래 참조에 관한 정보를 제공하면 파일 시스템이 이를 바탕으로 선 반입을 한다. 이들 정책들은 참조될 블록을 예측하는 것이 아니라 실제로 참조될 블록을 선반입하기 때문에 매우 뛰어난 성능을 보이게 된다. 그러나 응용 프로그램으로부터 힌트를 제공받아 처리할 수 있도록 기존의 운영체제에 응용 프로그램과 파일 시스템간의 인터페이스를 첨가하여 수정해야 한다. 또한 파일 시스템에 미래 참조에 대한 힌트를 제공하는 응용 프로그램을 작성해야 하는 프로그래머에게는 큰 부담을 주게 된다. 이와 관련한 연구로서 프로그래머의 부담을 없애는 방식으로 컴파일러가 운영체제에 참조 정보를 제공하는 방법이 가상 메모리 영역에서 제안된 바가 있다 [14]. 그러나 이러한 접근 방법은 컴파일러의 특성상 정적인 정보에 제한되는 한계를 가지고 있으며 아직 일반적인 시스템 환경에서는 검증된 바가 없다.

이와는 반대로 과거의 참조 정보를 바탕으로 미래 참조를 예측하여 선반입하는 정책들 [6, 7] 이 있다. 이는 과거 일정 기간의 참조 형태를 분석한 후 이를 바탕으로 미래에 참조될 것으로 예상되는 블록들을 선반입하는 것이다. 이러한 정책에서는 예측 정확도를 높이기 위하여 많은 양의 과거 참조 정보를 유지해야 하므로 많은 기억 장소와 처리 시간을 요구하게 되어 이것이 오히려 성능 저하의 요인이 될 수 있다는 단점이 있다.

위의 정책들과는 달리 Smith가 제안한 LRU-OBL 정책은 [4, 5] 1절에서 설명하였듯이 외부로부터의 어떠한 미래 참조에 관한 힌트가 없고 과거의 참조 정보도 유지하지 않는 정책이다. LRU-OBL 정책은 일괄 파일들과 임시 파일들에 대한 캐쉬 적중률을 80%정도 증가시키는 성능 향상을 나타내었다 [5]. 이러한 성능 향상은 선반입으로 인해 교체되는 블록들에 비해 LRU-OBL 정책에 의해 선반입되는 블록의 가치가 더 높음을 암시하고 있다.

그러나 LRU-OBL 정책은 다음과 같은 몇 가지 단점들을 가지고 있다.

- 첫째, LRU-OBL 정책은 선반입된 블록과 참조된 블록의 가치를 동일하게 평가하는 문제점을 가지고 있다. 이는 선반입하다 블록들 중에서 일정 기간 내에 실제로 참조되지 않을 블록들을 위해 재 참조될 수 있는 블록을 교체하게 되며, 선반입하다 블록이 참조된다고 할지라도 더 먼저 참조될 수 있는 버퍼 캐쉬 안의 블록을 교체하게 된다.
- 둘째, LRU-OBL 정책에서는 순차성을 고려하여 선반입을 하지만 순차적으로 한번 만 참조될 블록을 위해 빈번하게 참조되는 버퍼 캐쉬 안의 많은 블록을 교체하는 오류를 범하고 있다.
- 셋째, LRU-OBL 정책은 빈번하게 참조되는 블록들과 비교적 참조 횟수가 적은 블록들을 구별하지 못하는 LRU 정책의 비효율성을 여전히 가지고 있다.
- 넷째, LRU-OBL 정책에서는 짧은 기간 내에 빈번히 참조되나 장기적으로 재참조 되지 않는 단기적 지역성을 가진 블록들을 구별하지 못한다.

이러한 단점들로 인해 LRU-OBL 정책은 교체 정책과 선반입 정책이 통합되어 파일 시스템의 성능을 크게 향상시키긴 하였지만 많은 개선할 점들을 가지고 있다.

따라서 본 논문에서는 실용성을 위하여 미래의 참조에 관해 알 수 없는 현실적인 가정 하에 과거의 참조를 유지하지 않는 LRU-OBL 정책을 기본적인 모델로 설정하여 응용 프로그램들의 참조 형태에 대한 면밀한 관찰을 통해 LRU-OBL 정책의 문제점을 해결하는 W^2R 선반입/캐싱 통합 정책을 제안한다.

3. W^2R 정책

기존의 선반입 정책들은 적절한 선반입을 위하여 응용 프로그램의 미래 참조에 관해 미리 알고 있다는 비현실적인 가정을 하거나 과거의 참조 정보를 유지하는 오버헤드로 인해 실용적이지 못한 단점을 가지고 있다. 이것은 기존의 선반입 정책들이 어떤 블록을 선반입할 것인가에 초점이 맞추어져 있기 때문이다. 이러한 접근 방법과는 달리 LRU-OBL 정책은 공간적 지역성을 근거로 선 반입을 한다. LRU-OBL 정책은 매우 단순하여 최소한의 오버헤드를 가지고 있음에도 불구하고 파일 시스템의 성능을 크게 향상시켰다. 따라서 본 논문에서 제안하는 W^2R (Weighing and Waiting Room) 정책에서는 LRU-OBL 정책을 기본 모델로 설정하여 LRU-OBL 정책과 동일하게 참조된 블록의 논리적 다음 블록을 선반입하되 선반입된 블록의 효율적인 관리에 중점을 두고자 한다.

다음절에서 설명할 실험에 사용된 트레이스들에 대한 관찰 결과에 의하면 많은 블록들이 선반입된 후 실제로 참조되지 않는다. 표 1은 각 트레이스들에서 LRU-OBL 정책에 의해 선반입된 블록들 중 실제로 참조되지 않는 블록들의 비율을 보여준다. 각 트레이스들에 대해서는 다음절에서 자세히 설명할 것이다. 표 1의 (a)와 (b)의 데이타 베이스 환경의 트레이스들에서는 LRU-OBL 정책에 의해 선반입된 블록들 중에서 실제로 참조되지 않고 버퍼 캐쉬에서 방출된 블록들이 버퍼 캐쉬의 크기에

따라 전체 선 반입된 블록들의 45~70 %를 차지한다. 또한 매우 순차적인 성질을 갖는 것으로 알려진 표 1의 (c)와 (d)의 과학적 환경의 트레이스들에서도 5~58 %정도가 선반입되었으나 참조되지 않는다. 그러므로 선반입된 블록들 중에서 실제로 참조되지 않는 블록들은 잘못 선 반입된 것으로써 버퍼 캐쉬에 장시간 유지할 필요가 없는 블록들인 것이다. 이렇게 잘못 선 반입된 블록들은 다른 빈번하게 참조될 블록을 위하여 버퍼 캐쉬에서 조기에 방출되어야 한다.

표 1 LRU-OBL 정책에 의해 선반입된 블록들 중 실제로 참조되지 않는 블록의 비율 (%)

버퍼캐쉬크기	100	1000	2000	3000
선반입된블록수	339,979	200,484	172,773	156,418
참조않된블록	70.95 %	56.86 %	51.15 %	46.81 %

(a) DB2 트레이스

버퍼캐쉬크기	100	1000	2000	3000
선반입된블록수	864,605	688,834	610,019	560,201
참조않된블록	66.87 %	66.04 %	63.94 %	61.80 %

(b) OLTP 트레이스

버퍼캐쉬크기	100	1000	2000	3000
선반입된블록수	195,849	150,144	129,462	105,092
참조않된블록	58.43 %	39.96 %	28.42 %	18.84 %

(c) Sprite 트레이스 1

버퍼캐쉬크기	100	1000	2000	3000
선반입된블록수	92,094	84,321	79,556	75,007
참조않된블록	15.26 %	8.65 %	5.74 %	4.54 %

(d) Sprite 트레이스 2

W^2R 정책에서는 선반입된 블록과 참조된 블록의 가치를 달리 평가하기 위해 버퍼 캐쉬를 Weighing Room과 Waiting Room의 두 개의 논리적인 영역으로 분할한다. 그림 1은 이러한 내용을 반영하는 W^2R 정책의 논리적인 구성도 이다. Weighing Room은 적어도 한 번 이상 참조된 블록들을 관리하기 위한 공간이며 Waiting Room은 선반입되는 블록들을 관리하기 위한 공간이다. 선반입된 블록들은 일단 Waiting Room에 반입이 되었다가 실제로 참조되면 Weighing Room으로 이동한다. Waiting Room은 FIFO 방식으로 관리되기 때문에 만일 새로 선 반입되는 블록을 위한 공간이 부족할 때까지 참조되지 않을 경우에는 잘못 선반입된 블록으로 간주되어 방출되어진다. W^2R 정책에서는 이와 같은 버퍼 캐쉬 분할을 통해 잘못 선반입된 블록들을 버퍼 캐쉬에서 조기에 방출하게 되며 또한 선반입되는 블록에 의해 재 참조될 수 있는 블록의 교체를 방지할 수 있게 된다.

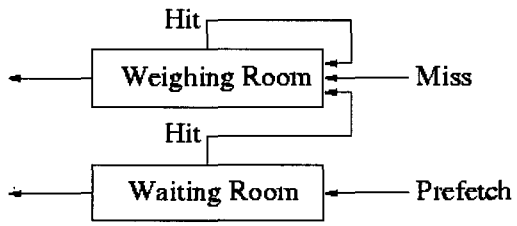


그림 1 W^2R 정책의 논리적 구성

Weighing Room은 한 번 이상 참조된 블록들을 대상으로 각 블록들의 무게를 재어 상대적으로 무게가 많이 나가는 블록들을 버퍼 캐쉬에 유지하는 역할을 수행한다. 여기서 무게라고 하는 것은 각 블록들의 재 참조에 대한 기대치를 의미한다. 즉, 상대적으로 재 참조될 확률이 높은 블록은 무게가 많이 나가는 블록이며, 재 참조될 확률이 적은 블록은 무게가 적게 나감을 의미한다. 각 블록들의 무게는 Weighing Room에 적용된 특정 교체 정책에 의해 결정된다.

예를 들어, LRU 정책을 사용할 경우에 버퍼 캐쉬내의 블록들의 무게를 재는 기준은 각 블록들이 참조된 순서이다. 가장 최근에 참조된 블록이 가장 무거운 블록으로 평가받아 Weighing Room의 MRU (Most Recently Used) 위치에 놓여지며 가장 오래 전에 참조된 블록이 가장 적은 무게로 평가되어 가장 먼저 교체되어 진다. 실제로 Weighing Room에는 LRU 정책 외에도 여러 다양한 교체 정책들이 적용될 수 있는데 본 논문에서는 LRU-OBL 정책과 동일한 블록을 선반입하면서도 선반입되는 블록과 참조되는 블록을 구별하는 것이 성능에 미치는 영향을 알아보기 위하여 Weighing Room에 LRU 정책을 적용한다.

특정 블록 i에 대한 요청이 들어 왔을 때 Weighing Room에 LRU 교체 정책을 적용한 W^2R 정책의 알고리즘은 그림 2와 같다.

```
 1 Algorithm W²R
 2 input : requested block number i
 3 output: requested block
 4 if i is in Weighing Room then
 5 *     put i at head of Weighing Room
 6 /* Weighing Room is managed as an LRU queue */
 7       if i+1 is not in either Room then
 8           prefetch i+1 from disk and
 9           put at head of Waiting Room
10 /* Waiting Room is managed as a FIFO queue */
11       end if
12 else if i is in Waiting Room then
13       remove i from Waiting Room
14 **    put i at head of Weighing Room
15       if i+1 is not in either Room then
16           prefetch i+1 from disk and
17           put at head of Waiting Room
18       end if
19 else
20       fetch i from disk
21 **    put i at head of Weighing Room
22       if i+1 is not in either Room then
23           prefetch i+1 from disk and
24           put at head of Waiting Room
25       end if
26 end if
```

그림 2 Weighing Room에 LRU 정책을 적용한 W2R 정책의 알고리즘

알고리즘을 설명하면, 디스크 블록 i 요구 시 블록 i가 Weighing Room에 존재하면 Weighing Room을 관리하는 LRU 정책에 의해 MRU (Most-Recently-Used) 블록으로 유지하고 Waiting Room에 존재하면 Weighing Room으로 이동시킨다. 만일 요구된 블록이 버퍼 캐쉬에 존재하지 않으면 디스크에서 읽어와 Weighing Room에 반입한다. 이때 i 블록의 논리적 다음 블록인 i+1 블록을 함께 검색하여 만일 버퍼 캐쉬에 존재하지 않으면 디스크로부터 선반입하여 Waiting Room에 저장한다. 요구된 블록이 Waiting Room에서 적중이 된 경우에 Weighing Room으로 이동하는 이유는 선반입된 블록이 예상대로 실제로 참조되어 적절하게 선반입된 블록으로 인정을 받았기 때문이다. 만일 일정한 기간 내에 실제로 참조되지 않을 경우에는 Waiting Room에서 새로이 선반입되는 블록들에 의해 조기에 교체되어 진다. 위 알고리즘에서 '*'과 '**'로 표시한 열(5, 14, 그리고 21열)은 Weighing Room에 LRU 정책을 적용한 부분에 해당한다.

4. 성능 평가

이 절에서는 W²R 정책의 성능을 평가하기 위한 실험 환경과 실험 결과를 기존의 버퍼 캐쉬 관리 정책들과 비교 분석한다.

4.1 실험 환경

본 논문에서는 W²R 정책의 성능을 평가하기 위하여 4.3 BSD 파일 시스템을 참고하여 C++ 언어로 제작한 파일 시스템 시뮬레이터와 과학적 환경의 특성을 보여주는 트레이스와 데이타 베이스 환경의 특성을 나타내는 트레이스들을 사용한다.

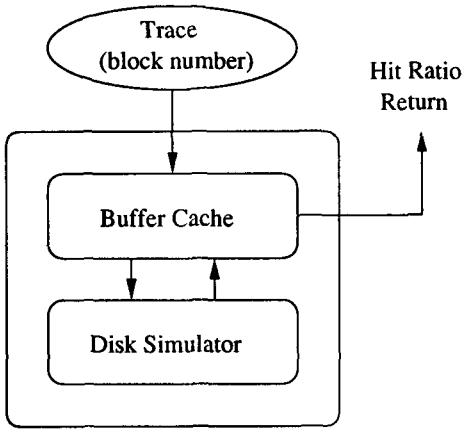


그림 3 파일 시스템 시뮬레이터의 구성

시뮬레이터는 그림 3과 같이 버퍼 캐쉬와 디스크 모듈의 두 부분으로 구성되어 있다. 시뮬레이터는 응용 프로그램으로부터 파일 시스템에 요청되는 디스크 블록에 대한 전체 요구들 중 버퍼 캐쉬에서 만족되는 비율인 버퍼 캐쉬 적중률을 측정한다. 버퍼 캐쉬 모듈은 트레이스에서 블록 번호를 넘겨받아 요청된 블록이 버퍼 캐쉬에 있으면 적중으로 표시를 해주고 없으면 디스크 모듈로 필요한 블록을 요청하게 된다. 시뮬레이터는 블록의 크기와 전체 버퍼 캐쉬의 크기, 그리고 버퍼 캐쉬 관리 정책을 인수로 사용한다.

과학적 환경의 특성을 보여주는 트레이스로는 Sprite 트레이스 [15] 를 사용한다. Sprite 트레이스는 버클리 대학에서 Sprite [16] 라고 하는 분산 파일 시스템을 사용하여 4대의 파일 서버와 40여대의 클라이언트에서 매일 사용하는 30여명의 사용자와 간헐적으로 사용하는 40여명의 사용자들에게서 24 시간 내지 48시간씩 여덟 개의 기간으로 나뉘어서 얻어진 트레이스이다. Sprite 트레이스가 과학적 환경의 특성을 나타내는 것은 이 기간 중 컴퓨터를 사용한 사람들이 운영체제 연구자, 컴퓨터 구조 연구자 그리고 VLSI 회로 설계 및 병렬 처리 연구자, 컴퓨터 그래픽 및 시스템 관리자 등이었기 때문이다. 본 논문에서는 실험을 위하여 Sprite 트레이스 중

에서 2번째 기간의 39번 클라이언트와 3번째 기간의 53번 클라이언트의 트레이스를 추출하여 사용한다. 2 번째 기간의 39번 클라이언트의 트레이스 (이하 Sprite 트레이스 1)는 49,277 개의 별개의 블록에 대해 239,748개의 요청을 하며, 3번째 기간의 53번 클라이언트의 트레이스 (이하 Sprite 트레이스 2)는 19,990개의 별개의 블록에 대해 141,233번의 요청을 한다. Sprite 트레이스 1은 과학적 환경의 일반적인 참조 형태를 포함하며 Sprite 트레이스 2는 최근 증가하고 있는 1 MB 이상의 매우 큰 파일에 대한 순차적인 참조를 포함하고 있다.

데이타베이스 환경의 특성을 보여주는 DB2 트레이스와 OLTP 트레이스는 Johnson과 Shasha의 논문 [2]에서 사용된 트레이스와 동일한 것을 사용한다. DB2 트레이스는 DB2 Commercial Application으로부터 추출되었으며 75,514개의 별개의 페이지들에 대해 500,000번의 요청을 포함하고 있다. On-Line Transaction Processing System으로부터 얻은 OLTP 트레이스는 CODASYL 데이타베이스에 대한 한시간 동안의 블록 참조 내용을 포함하고 있다. OLTP 트레이스는 186,880개의 별개의 블록에 대해 914,145번의 요청을 한다.

4.2 실험 결과

이 절에서는 W^2R 알고리즘의 성능 평가를 위하여 LRU 알고리즘과 2Q 알고리즘, 그리고 LRU-OBL 알고리즘의 성능과 비교한다. 표 2는 W^2R 정책의 버퍼 캐쉬 적중률 실험 결과이다.

실험 결과를 분석하기 전에 언급할 것은 각각의 표에 나타난 2Q 정책의 버퍼 캐쉬 적중률이 2Q 정책을 제안했던 Johnson과 Shasha의 논문 [2] 에서 보다 낮게 나타나 있다는 것이다. Johnson과 Shasha에 의하면 버퍼 캐쉬의 분할에 있어서 A1 큐의 크기를 너무 작게 하면 빈번하게 참조되는 블록의 판단 기준이 너무 까다로워지게 되고 A1 큐의 크기를 너무 크게 하면 상대적으로 Am 큐의 크기가 줄어들어 빈번하게 참조되는 블록을 위한 공간이 줄어들어 적중률이 떨어질 수 있기 때문에 오히려 성능이 나빠지게 되는 문제를 발생시킨다. 이러한 문제를 해결하기 위하여 Johnson과 Shasha는 A1 큐에 디스크 블록 대신에 디스크 블록에 대한 포인터를 저장함으로써 적은 공간으로 디스크 블록에 대한 많은 양의 정보를 유지하고자 했다. 그러므로 A1 큐에서 재 참조되어 Am 큐로 이동할 경우에는 추가적인 디스크 접근이 요구된다. 따라서 본 논문에서는 다른 정책과의 공평한 비교를 위하여 A1 큐에 포인터가 아닌 실제 디스크 블록을 저장하도록 하였다.

표 2 W^2R 정책의 버퍼 캐쉬 적중률 실험 결과(%). 괄호 안은 전체 캐쉬에 대한 Waiting Room의 비율을 의미

캐쉬크기	1000	2000	3000
LRU	25.97	28.62	43.84
2Q	30.67	31.83	32.13
LRU-OBL	63.12	70.93	73.84
W^2R	67.03 (80%)	79.18 (90%)	79.34 (90%)

(a) Sprite 트레이스 1

캐쉬크기	1000	2000	3000
LRU	37.82	40.91	47.40
2Q	38.13	41.90	47.13
LRU-OBL	91.69	93.57	94.97
W^2R	92.45 (60%)	93.81 (30%)	95.33 (10%)

(b) Sprite 트레이스 2

캐쉬크기	1000	2000	3000
LRU	65.44	70.38	72.95
2Q	66.92	71.93	74.29
LRU-OBL	79.68	84.33	87.29
W^2R	81.85 (3%)	86.39 (1%)	88.87 (1%)

(c) DB2 트레이스

캐쉬크기	1000	2000	3000
LRU	32.83	42.47	47.10
2Q	37.70	43.95	47.81
LRU-OBL	51.62	60.43	65.48
W^2R	56.06 (3%)	64.63 (2%)	68.87 (1%)

(d) OLTP 트레이스

표 2 (a)의 Sprite 트레이스 1을 사용한 결과를 보면 W^2R 정책은 LRU-OBL 정책에 비해 4~8 %가 향상된 성능을 나타낸다. 이를 통해 선반입된 블록과 참조된 블록을 구별하는 것이 성능 향상을 가져온다는 것을 확인

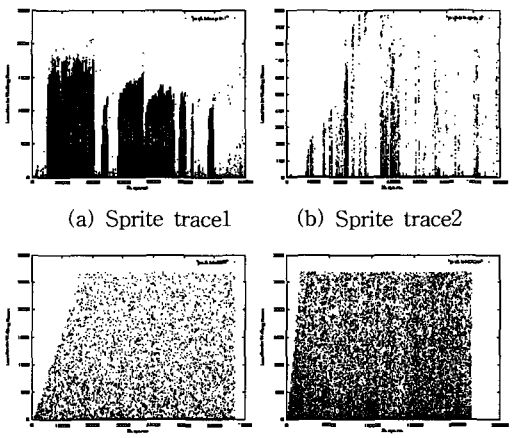
할 수 있다. 물론 Sprite 트레이스 2를 사용한 표 2 (b)의 결과에서는 W^2R 정책이 LRU-OBL 정책에 비해 1 % 내의 작은 향상을 나타내는데 이것은 Sprite 트레이스 2가 큰 파일에 대한 매우 순차적인 참조를 포함하고 있어 LRU-OBL 정책이 최대의 효과를 발휘하기 때문이다.

과학적 환경에 비해 순차성이 적은 데이타 베이스 환경에서도 W^2R 정책은 좋은 성능을 발휘한다. 표 2 (c)의 DB2 트레이스를 사용한 경우는 LRU-OBL 정책에 비해 1~2 %가 향상되었으며 표 2 (d)의 OLTP 트레이스에서도 3~5 %가 향상된 결과를 나타낸다.

여기서 언급해야 할 것은 표에서 제시한 분할률이 최적은 아니라는 것이다. 본 논문에서는 데이타 베이스 트레이스를 이용한 실험에서 Waiting Room의 비율을 10 %까지는 실험의 용이함을 위하여 1 % 간격으로 변화하여 측정하였다. 그런데 데이타 베이스 환경에서는 Waiting Room에 아주 적은 양을 할당할 때 좋은 성능을 발휘하므로 이 1 %는 버퍼 캐쉬 크기가 커질수록 절대 크기가 증가하여 어느 시점에서는 선반입된 블록이 참조되기 전까지 기다리기 위한 가장 적절한 공간을 초과하게 된다. 따라서 버퍼 캐쉬가 커질수록 1 %보다 적은 양을 Waiting Room에 할당할 경우 W^2R 정책의 성능은 더욱 향상될 것이다. 또한 Waiting Room의 비율을 10 %에서 90 %까지는 10 % 간격으로 측정하였는데 Sprite 트레이스를 사용한 실험에서 이 또한 더 작은 단위로 세분하여 측정할 경우 더 많은 성능 향상을 가져오는 지점이 존재할 것으로 예상된다.

지금까지의 실험에서 살펴본 바와 같이 W^2R 정책에서 최고의 성능을 가져오는 Waiting Room의 비율은 사용하는 트레이스와 버퍼 캐쉬의 크기에 따라 달라진다. 이렇게 트레이스별로 최고의 성능을 가져오는 Waiting Room의 비율이 차이가 나는 것은 각 트레이스들의 참조 형태의 차이에서 발생하는 것이다. 즉, 선반입된 디스크 블록들은 실제 참조될 때까지 Waiting Room에서 기다려야 하는데, 각 트레이스 별로 선반입된 후 참조될 때까지의 시간이 다르기 때문에 각 트레이스 별로 요구되는 Waiting Room의 크기가 달라지게 되는 것이다.

그림 4는 이러한 사실을 확인해주는 관찰 결과이다. 그림 4는 버퍼 캐쉬 크기가 3000에서 Waiting Room에 90 % (2,700개)를 할당했을 때 선 반입된 블록이 Waiting Room의 어느 위치에서 참조가 되는지를 관찰한 것이다. 그림에서 X-축은 선반입된 블록들이 참조되는 순서를 의미하며 Y-축은 선반입된 블록이 참조 될 때 Waiting Room에서의 위치를 의미한다. 즉 가장 최근에 선 반입된 블록의 위치는 1이 되고 가장 오래 전에 선 반입된 블록의 위치는 2700이 되는 것이다. 관찰 결과에 의하면 과학적 환경과 데이타 베이스 환경에서의 차이가 확연하게 드러난다. 먼저 데이타베이스 환경의 DB2 와 OLTP 트레이스의 경우에는 대부분의 블록들이 그림에서는 정확한 값을 알기가 어렵지만 자세한 관찰에 의하면 30~50의 위치 내에서 집중적으로 참조되며 나머지 블록들은 전체적으로 고르게 분포되어있다. 그러므로 데이타베이스 환경에서는 Waiting Room에 30-50의 아주 적은 공간만을 할당해야 Waiting Room을 통한 이익을 극대화할 수 있음을 알 수 있다.



(a) Sprite trace1 (b) Sprite trace2

(c) DB2 trace (d) OLTP trace

그림 4 Waiting Room에 선반입된 후 참조되기까지의 시간

그러나 Sprite 트레이스의 경우에는 어느 시점에서 경계선을 그어야 할지 쉽게 알 수 없다. 순차적으로 참조되는 모습도 많지만 선반입되어 참조되는 위치가 집중되어 있지 않고 불규칙적으로 분포되어 있다. 따라서 이러한 참조 형태에 대해서는 Weighing Room과 Waiting Room에서 얻을 수 있는 상대적인 이익과 손실을 고려하여 이익의 합이 최대가 되는 시점을 최적의 분할률로 선택해야 하는 것이다.

5. W^2R 정책의 모듈성

W^2R 알고리즘의 또 다른 장점은 이 정책이 모듈성을 가지고 있다는 것이다. 즉, 지금까지 알려진 어떤 교체 정책도 Weighing Room에 적용되어질 수 있다. 이전에 서술한 것처럼, Weighing Room에서의 적중률은 작업

부하와 알고리즘에 의존한다. 버퍼 관리에 관한 최근의 연구 결과에 따르면, LRU 알고리즘이 최고의 버퍼 교체정책이 아닐 수 있다는 것이다[1, 2, 3]. 따라서 더 나은 성능을 가져오는 새로운 알고리즘이 개발되면 이러한 알고리즘들이 W^2R 알고리즘에 통합될 수 있다.

다른 알고리즘을 Weighing Room에 통합하기 위해서는 그림 2의 W^2R 알고리즘에서 줄 번호 5, 14, 그리고 21 ('*'으로 표시된 부분)만 수정하면 된다. 줄 번호 5는 블록이 버퍼 안에서 참조되었을 때의 작동 내용으로 대치되어진다. 예를 들어, 2Q 알고리즘에서는 블록이 버퍼 안에서 적중되면, 그것이 Am큐인지 A1 큐인지를 판단한다. 그리고 적절한 대응이 이루어진다. 따라서 Weighing Room에 2Q 알고리즘을 사용하기 위해서는 줄 번호 5가 다음의 코드로 대치되어야 한다.

```
if i is in Am queue then
        put i at head of Am queue
else if i is in A1out queue then
        remove i from A1out queue
        put i at head of Am queue
else
        /* Do Nothing  ! */
end if
```

반면에, 줄 번호 14와 21은 블록이 디스크로부터 반입되어 버퍼로 옮겨질 때의 작동 내용으로 대치되어진다. 2Q 알고리즘의 경우에는 단지 다음과 같은 내용으로 대치되어진다.

put *i* at head of A1in queue.

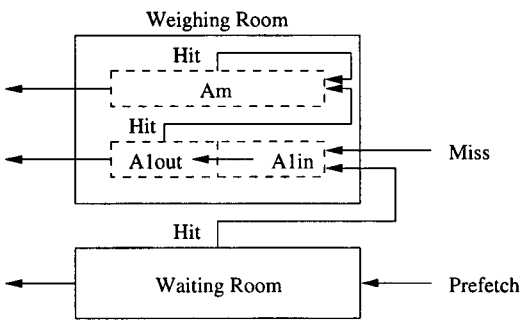
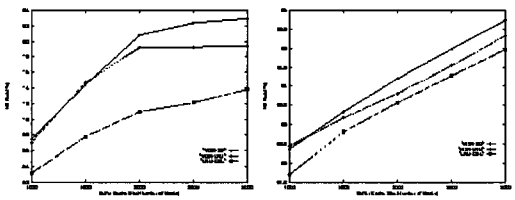


그림 5 W^2R-2Q 정책의 논리적 구성

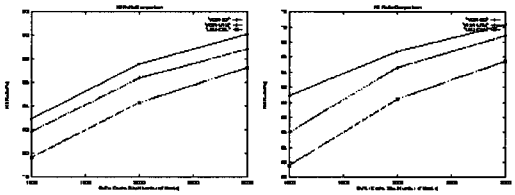
알려져 있는 버퍼 교체 알고리즘들 중에서 2Q 알고리즘이 좋은 버퍼 적중률을 보여주는 알고리즘들 중의 하나이므로 우리는 이 알고리즘을 W^2R 알고리즘에 적용하기로 결정하였다. 그림 5는 Weighing Room에 2Q 알고리즘을 적용한 W^2R-2Q 알고리즘의 구조 도를 보여준다.

W^2R-2Q 알고리즘을 사용한 실험 결과가 그림 6에 보여진다. W^2R-2Q 알고리즘과 상대적으로 Weighing Room에 LRU 알고리즘을 사용한 결과를 W^2R-LRU로 표시한다. LRU-OBL과 W^2R-LRU 알고리즘들의 실험 결과는 표 2에서 나타낸 값과 동일한 값이다. W^2R-2Q 에서는 편의를 위하여 표 2와 동일하게 Waiting Room의 크기를 설정하였다.

W^2R-2Q 알고리즘을 사용할 때의 성능 향상의 원인은 주로 Weighing Room에서의 적중률 향상 덕분이다. 물론, 알고리즘이 변화하면 Weighing Room에 보관하는 블록의 내용이 변화하기 때문에 Waiting Room에 있는 블록들 또한 약간의 영향을 받게된다. 하지만 전체적으로 Weighing Room에 보다 나은 알고리즘을 사용하게 되면, 전체적인 성능향상을 가져온다는 것을 알 수 있다. 구체적으로 살펴보면, W^2R-2Q 알고리즘은 2Q 알고리즘에 비해서는 최고 23.19 %, LRU-OBL 알고리즘에 비해서는 최고 10.25%의 성능 향상을 보여준다.



(a) Sprite 트레이스1 (b) Sprite 트레이스2



(c) DB2 트레이스 (d) OLTP 트레이스

그림 6 W^2R-2Q 정책의 버퍼 캐쉬 적중률 실험 결과

6. 결론 및 향후 연구

본 논문에서는 외부로부터의 미래 참조에 관한 힌트가 없고, 과거의 참조 정보도 유지하지 않아 적은 부담으로 효율적인 입출력을 지원하는 선반입/캐싱 통합 정책인 W^2R 정책을 제안하였다. W^2R 정책에서는 버퍼

캐쉬를 논리적으로 두개의 영역, 즉 Weighing Room과 Waiting Room 으로 분할한다. Waiting Room으로의 선반입을 통해 선반입으로 인해 빈번하게 참조되는 블록의 피해를 방지하고 잘못 선반입된 블록을 버퍼 캐쉬에서 조기에 방출한다. 또한 Weighing Room을 통해 시간적 지역성을 가진 블록들을 여과한다. Weighing Room은 사용하는 응용 프로그램의 특성에 따라 LRU 교체 정책, 2Q 교체 정책 등 다양한 버퍼 캐쉬 교체 정책들로 관리되어질 수 있다.

W^2R 정책은 트레이스 기반 시뮬레이션을 통해 성능을 측정한 결과 기존의 대표적인 버퍼 캐쉬 관리 정책들인 LRU, 2Q, 그리고 LRU-OBL등의 정책들에 비해 가장 높은 버퍼 캐쉬 적중률을 나타냈다.

본 논문에서는 W^2R 정책의 Weighing Room과 Waiting Room으로의 분할률에 대해서는 트레이스별로 실험을 통하여 얻어진 고정된 비율을 제시하였다. 그런데 이러한 버퍼 캐쉬의 고정 분할은 W^2R 정책이 특정 참조 형태에 국한되는 제약점을 가지고 있다. 따라서 W^2R정책이 다양한 참조 형태에 대해 항상 좋은 성능을 발휘할 수 있도록 현재 우리는 버퍼 캐쉬의 최적의 분할률을 프로그램의 실행 시에 동적으로 결정하는 방안을 연구하고 있다.

또한 W^2R 정책이 단일 시스템 환경에서 설계된 정책이나 고속의 네트워크 속도를 기반으 로 한 분산 환경에서 W^2R 정책이 어떻게 적용되어야 할 지에 관해 연구할 예정이다.

참 고 문 헌

[1] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297--306, May 1993.

[2] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439--450, 1994.

[3] J. T. Robinson and N. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference*, pages 134--142, 1990.

[4] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 3(3):7--21, December 1978.

[5] Alan Jay Smith. Disk cache-miss ratio analysis and design considerations. *ACM Transactions on Computer Systems*, 3(3):161--203, August 1985.

[6] Thomas M. Kroeger and Darrell D.E. Long. Predicting File System Actions from Prior Events. In *1996 Annual USENIX Technical Conference*, October 1996.

[7] Vidyadhar Phalke and Bhaskarpillai Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '95/ Performance '95)*, pages 291--300, May 1995.

[8] Pei Cao, Edward Felten, and Kai Li. Application-Controlled File Caching Policies. In *Summer USENIX '94*, pages 171--182, JUNE 1994.

[9] Pei Cao and Edward W. Felton. Implementation and Performance of Integrated Application-Controlled File Caching, Prefetching, and Disk Scheduling. *ACM Transactions on Computer Systems*, 14(4):311--343, November 1996.

[10] F. Chang and G. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[11] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. An Implementation Study of a Detection Based Adaptive Block Replacement Scheme. In *Proceedings of the 1999 USENIX Technical Conference*, pages 239--252, 1999.

[12] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating Principles*, pages 79--95, December 1995.

[13] Ramakrishna Karedla and J.Spencer Love and Bradley G. Wherry. Caching Strategies to Improve Disk System Performance. *Computer*, 27(3):38--46, March 1994.

[14] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications. In *USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[15] Mary G. Baker, John H. Hartman, Michael D. Kupfer, KenW. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th Symposium on Operating System Principles*, pages 198--212, Pacific Grove, CA, October 1991.

[16] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23--36, February 1988.

전 홍 석

1996년 홍익대학교 컴퓨터공학과. 1998년 홍익대학교 전자계산학과 석사. 1997년 12월 ~ 1999년 8월 팅크웨어 시스템즈(주) 기술 연구소 선임연구원. 1998년 3월 ~ 현재 홍익대학교 전자계산학과 박사과정. 관심분야는 시스템소프트웨어, 분산시스템

노 삼 혁

1986년 서울대학교 컴퓨터공학과. 1993년 매릴랜드대학교 컴퓨터공학과 박사. 1994년 ~ 현재 홍익대학교 컴퓨터공학과 조교수. 관심분야는 시스템소프트웨어, 병렬처리 시스템