적응력있는 블록 교체 기법을 위한 효율적인 버퍼 할당 정책

(Efficient Buffer Allocation Policy for the Adaptive Block Replacement Scheme)

최 종 무 [†] 조 성 제 [††] 노 삼 혁 [†††] 민 상 렬 [††††] 조 유 근 [††††]

(Jongmoo Choi) (Seongje Cho) (Sam Hyuk Noh) (Sang Lyul Min) (Yookun Cho)

요 약 본 논문에서는 디스크 입출력 시스템의 성능을 향상시키기 위한 효율적인 버퍼 관리 기법을 제시한다. 본 기법은 사용자 수준의 정보 없이 블록의 속성과 미래 참조 거리간의 관계를 기반으로 각 응용의 블록 참조 패턴을 자동으로 발견하고, 발견된 참조 패턴에 적합한 최적 블록 교체 기법을 적용한다. 또한, 응용이 참조하는 블록이 버퍼 캐쉬에 없어 새로운 버퍼 블록이 요구될 때, 응용별로 블록 참조 패턴에 따라 버퍼 예상 적중률을 분석하여 이를 기반으로 전체 버퍼 캐쉬의 적중률이 극대화되도록 해 주는 버퍼 할당 기법을 제안한다. 이러한 모든 과정은 시스템 수준에서 자동으로 그리고 온라인으로 수행된다. 제시한 기법의 성능을 평가하기 위해 블록 참조 트레이스를 이용해 모의 실험을 수행하였다. 실험 결과 제시한 기법은 적은 오버헤드로 기존의 블록 교체 기법들보다 캐쉬 블록의 적중률을 크게 향상시켜 주었다.

Abstract The paper proposes an efficient buffer management scheme to enhance performance of the disk I/O system. Without any user level information, the proposed scheme automatically detects the block reference patterns of applications by associating block attributes with forward distance of a block. Based on the detected patterns, the scheme applies an appropriate replacement policy to each application. We also present a new block allocation scheme to improve the performance of buffer cache when kernel needs to allocate a cache block due to a cache miss. The allocation scheme analyzes the cache hit ratio of each application based on block reference patterns and allocates a cache block to maximize cache hit ratios of system. These all procedures are performed on-line, as well as automatically at system level. We evaluate the scheme by trace-driven simulation. Experimental results show that our scheme leads to significant improvements in hit ratios of cache blocks compared to the traditional schemes and requires low overhead.

1. 서 론

최근 VLSI 기술의 급속한 발달로 처리기와 입출력 시스템의 속도 차이가 점점 커지고 있으며 입출력 시스템의 성능이 전체 시스템의 성능을 좌우하게 되었다[1, 2]. 따라서 입출력 서비스 시간을 줄여 응용의 응답 시간을 단축하려는 연구가 활발하게 진행되고 있는데, 그 중에 하나가 파일 캐싱에 대한 연구이다. 파일 캐싱은 자주 참조되는 디스크 블록들을 버퍼 캐쉬에 유지하여 디스크 블록에 대한 입출력 요구를 주 기억장치에서 처리함으로써 시스템의 성능을 향상시켜 준다.

버퍼 캐쉬를 효과적으로 관리하기 위한 블록 관리 기법들이 기존의 연구에서 많이 제안되었다. 먼저, 효율적인 캐싱 정책을 개발하기 위해 응용들의 블록 참조 패턴에 대한 모델과 특성을 분석한 연구가 수행되었다[3,

† 비 회 원 : 서울대학교 컴퓨터공학과
choijm@ssrnet.snu.ac.kr
†† 비 회 원 : 단국대학교 전산통계학과 교수
sjcho@dankook.ac.kr
††† 종신회원 : 홍익대학교 정보컴퓨터공학부 교수
noh@cs.hongik.ac.kr
†††† 종신회원 : 서울대학교 컴퓨터공학과 교수
symin@dandelion.snu.ac.kr
cho@ssrnet.snu.ac.kr
논문접수 : 1999년 5월 19일
심사완료 : 2000년 1월 13일

4]. 이러한 연구들은 어떤 블록이 더 가까운 미래에 참조될 것인가에 대한 정보를 제공하였으며, 이를 기반으로 블록 교체 및 캐쉬 관리 정책들이 제안되었다[5, 6, 7, 8]. 예를 들어, FBR(Frequency Based Replacement)[5]와 LRU-K[6] 등의 블록 교체 정책은 각 블록들이 서로 독립적인 참조 확률을 갖는다는 독립 참조 모델(Independent Reference Model)[3]을 기반으로 교체될 블록을 결정한다.

응용 수행 시 블록 참조 패턴을 동적으로 발견하고 이를 캐쉬 관리 정책에 이용하는 연구들도 수행되었다[9, 10, 11]. 이러한 연구들로는 응용의 파일 참조 패턴을 분석하고 그 결과를 다음 수행 때 이용하여 캐쉬를 관리하는 기법[9], 응용의 작업 집합(working set) 전이 과정을 분석하여 블록 교체 정책에 반영하는 기법[10], 가상 메모리 시스템에서 순차 참조 패턴이 발견될 시에 MRU 페이지 교체 정책을 적용하여 응용의 실행 시간을 감소시키는 기법[11] 등이 있다. 시스템 수준이 아닌 데이타베이스나 컴파일러 수준에서 블록 참조 패턴을 자동으로 발견하는 기법도 제안되었다. 데이타베이스의 경우 질의어를 분석하여 파일의 참조 패턴을 예측하고 이를 기반으로 최대 이익이 되도록 캐쉬를 관리하는 기법[14]이 제시되었으며, 컴파일러 수준에서 응용의 파일 참조 특성을 분석하여 선반입과 블록 교체에 이용하는 기법[15]도 제시되었다.

한편, 응용의 블록 참조 패턴에 대한 사용자 수준 정보를 이용한 캐쉬 관리 기법들도 제안되었다[12, 13]. 이들 연구는 사용자 수준 정보에 기반한 캐쉬 관리 기법이 각 응용에게 최적에 가까운 캐쉬 성능을 제공한다는 것과, 블록 참조 패턴에 대한 정확한 정보가 캐쉬 관리의 효율을 높일 수 있다는 것을 보여 준다. 하지만 응용의 블록 참조 패턴에 대한 정보를 얻어내기 위해서 사용자는 각 응용의 블록 참조 특성을 이해하여야 하는데, 이러한 요구는 사용자에게 큰 부담이 되며 다양한 종류의 응용들에게 적응력있는 블록 교체 기법을 제공하지 못하는 문제점이 있다.

본 논문에서는 시스템 수준에서 온라인으로 응용 별로 블록 참조 패턴을 발견할 뿐만 아니라 각 응용의 버퍼 캐쉬에 대한 예상 적중률을 분석하고, 이를 기반으로 시스템의 성능을 향상시키는 버퍼 할당 기법을 제시한다. 본 기법은, 어떤 응용이 참조한 블록이 캐쉬에 없어 새로운 버퍼 블록이 요구될 때, 응용별로 블록 참조 패턴에 따른 버퍼 예상 적중률을 계산하여 적중률 감소가 가장 적은 응용의 버퍼를 회수함으로써 시스템 전체 캐쉬의 적중률이 최대가 되도록 해 준다. 또한, 응용 수행 중 참조 패턴의 변화가 발생하면 이를 반영할 수 있으며, 사용자 수준의 정보를 필요로 하지 않는다.

본 논문의 구성은 다음과 같다. 2장에서는 기본적인 4가지 블록 참조 패턴의 특성을 분석하고, 이를 자동으로 발견하는 기법을 간단히 기술한다. 3장에서는 참조 패턴에 따른 버퍼 적중률 모델에 대해 설명하고, 4장에서는 버퍼 적중률에 기반한 블록 할당 정책을 제시한다. 5장에서는 생성한 블록 참조 트레이스와 실제 응용들의 블록 참조 트레이스를 이용해 블록 할당 정책의 성능 분석 결과를 설명한다. 마지막으로 6장에서 결론을 맺고 향후 연구 과제를 논의한다.

2. 적응력있는 블록 교체 기법

버퍼 할당 정책을 제시하기 전에, 기존에 본 연구진이 제안했던 적응력있는 블록 교체 기법[20], 즉 DEAR (DEtection-based Adaptive Replacement)에 대해 간단하게 기술한다. DEAR는 응용의 블록 참조 패턴을 자동으로 발견하여 블록을 교체하는 기법으로 자세한 내용은 [20]에 기술되어 있다. 본 논문의 주목적은 [20]에서 제안된 DEAR 기법 상에서 효율적인 버퍼 할당 정책을 개발하는 것이다.

많은 응용들은 수행되는 동안 매우 규칙적인 블록 참조 패턴을 보이며, 각 특성에 따라 서로 다른 블록 참조 패턴을 나타낸다[4, 6, 13, 16, 17, 18, 19]. 즉, 많은 과학 계산 응용은 블록들을 일정한 간격으로 반복 참조하는 패턴을 보이며[16], 데이타베이스 응용은 블록들을 서로 다른 확률로 참조하는 비 균등(non-uniform) 참조 패턴을 보인다[6, 17]. 반면에 많은 UNIX 응용들은 블록들을 순차적으로 참조하거나 시간적인 지역성을 가지는 참조 패턴을 보이며[4, 13], Web 응용은 시간적 지역성이 있는 비 균등 참조 패턴을 보인다[18]. 그리고 멀티미디어 응용은 순차적인 또는 반복적인 참조 패턴을 보인다[19]. 이러한 기존의 연구를 기반으로 본 논문에서는 각 응용의 일반적인 블록 참조 패턴을 블록들이 한번 참조된 후 다시 재 참조되지 않는 순차 참조(sequential reference) 패턴, 모든 블록들이 일정한 간격으로 재 참조되는 반복 참조(looping reference) 패턴, 최근에 참조된 블록일수록 더 가까운 미래에 참조되는 시간적인 지역성을 보이는 참조(temporally-clustered reference) 패턴, 각 블록이 고유한 확률로 참조되는 확률 참조(probabilistic reference) 패턴 등으로 분류하였다.

DEAR 기법에서는 블록 참조 패턴을 자동으로 발견하기 위해 블록의 속성(block attribute)과 미래 참조 거

리(forward distance)를 이용한다. 블록의 속성이란 블록의 과거 참조로부터 얻어지는 정보로, 과거 참조 거리(backward distance)와 참조 횟수(frequency)라는 두개의 속성을 이용한다. 블록의 과거 참조 거리는 현재 시간[1]과 해당 블록이 최근에 참조된 시간간의 차이이며, 참조 횟수는 블록이 현재 시간까지 참조된 횟수이다. 그리고 블록의 미래 참조 거리는 현재 시간과 그 블록이 미래에 다시 참조될 시간간의 차이이다. 일반적으로 블록의 미래 참조 거리는 캐쉬에서 블록의 가치를 결정하며, 블록 교체 시 미래 참조 거리가 가장 큰 것을 교체하는 것이 최적이다[3].

하지만 블록의 미래 참조 거리는 온라인으로 알 수 없는 정보이다. 온라인으로 블록 참조 패턴을 발견하기 위해서, 본 논문에서는 주기적인 블록 참조 패턴 발견 방식을 제안한다. 이 방식은 한 단계 이전 시점을 기준으로 참조 패턴을 분석하는 2 단계 파이프라인(two-stage pipeline with one-level look behind) 방법을 사용한다. 이는 블록의 미래 참조 거리 정보를 제공할 뿐 아니라 응용의 블록 참조 패턴 변화를 인식할 수 있다[20].

블록 참조 패턴의 발견은 주기적으로 호출되는 발견 모니터(detection monitor) 루틴에 의해 수행된다. 발견 모니터가 i-번째 시점(이후 m_i로 표현)에 호출되면, m_{i-1} 시점을 기준으로 m_{i-1}과 m_i 기간동안 참조된 블록들에 대해 미래 참조 거리를 계산한다. 또한, 그 블록들에 대해 과거 참조 거리와 참조 횟수를 수집하여 2개의 순서 리스트(ordered list)를 구성한다. 순서 리스트는 블록 속성마다 하나씩 구성되며 블록의 속성에 따라 오름차순으로 정렬된다. 이 순서 리스트는 같은 수의 블록들로 구성된 하위 리스트(sublist)들로 다시 분할된다. 그 다음, 각 하위 리스트에 속한 블록들의 평균 미래 참조 거리와 블록 속성간의 관계를 기반으로 블록 참조 패턴을 발견한다. 발견이 끝나면 m_{i-1}과 m_i 기간동안 참조된 블록의 속성을 갱신한다. 갱신된 블록 속성은 m_{i+1} 시점에 발견 모니터가 다시 호출되면 이용된다.

예를 들어 발견 주기가 10일 때, m_{i-1} = 40과 m_i = 50 사이에 b_4, b_2, b_6, b_{12}, b_4, b_8, b_{11}, b_6, b_4, b_6의 순서로 10개의 블록이 참조되었다고 가정하자(그림 1의 (b) 참조). 또한 m_{i-1} 시점에서 각 블록 b_4, b_2, b_6, b_{12}, b_8, b_{11}의 과거 참조 거리가 각각 15, 12, 25, 4, 20, 9이고, 참조 횟수가 각각 6, 4, 5, 2, 1, 1 이라고 가정하자(그림 1의 (a) 참조). m_i 시점에 호출된 발견 모니터는 우선 m_{i-1}과 m_i 기간동안 참조된 블록들의 미래 참조 거리를 m_{i-1}시점을 기준으로 계산한다. 따라서, b_4, b_2, b_6, b_{12}, b_8, b_{11}의 미래 참조 거리는 각각 1, 2, 3, 4, 6, 7이다. 그리고 발견 모니터는 2개의 순서 리스트를 만든다(그림 1의 (c) 참조). 하나는 과거 참조 거리를 기반으로 정렬된 리스트이며, 다른 하나는 참조 횟수를 기반으로 정렬된 리스트이다. 각 순서 리스트는 2개 블록을 갖는 3개의 하위 리스트로 분할된다. 그림에서 $sublist_i^{bd}$와 $sublist_i^{fr}$는 각각 과거 참조 거리와 참조 횟수를 기반으로 순서 리스트를 만들었을 때 i-번째 하위 리스트이다. 그림 1을 보면 참조 횟수가 큰 블록들의 하위 리스트가 더 작은 미래참조 거리를 가지며, 따라서 다음 단락에서 설명할 발견 규칙에 따라 확률 참조 패턴으로 발견할 수 있다. 발견이 완료되면 m_{i-1}과 m_i 동안 참조된 블록의 정보는 각 블록의 속성에 반영된다. 즉, b_4, b_2, b_6, b_{12}, b_8, b_{11}의 과거 참조 거리가 각각 2, 9, 1, 7, 5, 4로, 참조 횟수는 각각 9, 5, 8, 3, 2, 2로 갱신된다. 갱신된 속성은 m_{i+1}시점에서 이용된다.

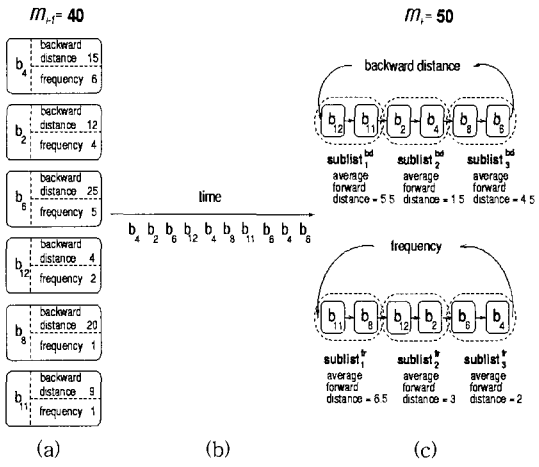


그림 1 블록 참조 패턴의 발견 예

본 논문에서 분류한 각 블록 참조 패턴은 블록 속성과 미래 참조 거리간에 서로 다른 관계를 보인다. 따라서 이러한 관계를 순서 리스트에서 적용하여 다음과 같은 참조 패턴을 발견할 수 있다 (아래에서 $Avg_fd(sublist)$는 $sublist$에 속한 블록들의 평균 미래 참조 거리이다).

- **순차 참조**: 모든 블록의 미래 참조 거리는 ∞가 된

1) 여기서 시간은 가상 시간으로 응용이 블록을 참조할 때의 시간을 의미하는데, 처음에는 0이며 블록이 참조될 때마다 시간이 한 단위씩 증가한다.

다. 결국 각 하위 리스트간에 $Avg\_fd(sublist_1^{bd}) = Avg\_fd(sublist_2^{bd}) = \ldots = Avg\_fd(sublist_1^{fr}) = Avg\_fd(sublist_2^{fr}) = \ldots = \infty$ 의 식이 성립하면 순차 참조이다.

o 반복 참조: 과거 참조 거리가 작은 블록일수록 더 큰 미래 참조 거리를 갖는다. 결국 과거 참조 거리를 기반으로 구성된 순서 리스트에서 모든 i, j에 대해 $i < j$ 일 때 $Avg\_fd(sublist_i^{bd}) > Avg\_fd(sublist_j^{bd})$ 가 성립하면 반복 참조이다.

o 시간적 지역성을 보이는 참조: 과거 참조 거리가 작은 블록일수록 더 작은 미래 참조 거리를 갖는다. 모든 i, j에 대해 $i < j$ 일 때 $Avg_fd(sublist_i^{bd}) < Avg_fd(sublist_j^{bd})$가 성립하면 시간적 지역성을 보이는 참조이다.

o 확률 참조: 이 참조 패턴은 독립 참조 모델[3]로 모델링 될 수 있으며, 각 블록(b_i)은 서로 독립적인 참조 확률(p_i)을 가진다. 확률 이론에 따라 블록 b_i의 평균 미래 참조 거리는 $1/p_i$에 비례하며, 안정 상태에서 블록의 p_i는 블록의 참조 횟수와 양의 상관 관계를 갖는다. 따라서 참조 횟수가 많은 블록일수록 더 작은 미래 참조 거리를 가진다. 결국 참조 횟수를 기반으로 구성된 순서 리스트에서 모든 i, j에 대해 $i < j$ 일 때 $Avg_fd(sublist_i^{fr}) > Avg_fd(sublist_j^{fr})$이 성립하면 참조 확률이다.

블록 참조 패턴이 발견되면, DEAR 기법은 발견된 블록 참조 패턴에 적합한 블록 교체 정책을 각 응용에게 적용한다. 순차 참조나 반복 참조 패턴이 발견되면 MRU 교체 정책을 적용하는 것이 최적이다[14]. 시간적 지역성을 보이는 참조에 대해서는 LRU 정책을, 확률 참조에 대해서는 LFU 정책을 적용하는 것이 최적이다[3]. 참조 패턴이 발견되기 전까지는 LRU 정책을 적용하며, 수행 중에 새로운 참조 패턴이 발견되면 그에 적합한 교체 정책을 적용한다.

3. 버퍼 적중률 모델

시스템은 모든 응용의 버퍼 적중률의 합이 최대가 되도록 버퍼 캐쉬를 관리해야 한다. 따라서 블록 할당 정책의 성능 측정 단위는 다음의 식으로 표현될 수 있으며, 이를 극대화시키는 것이 버퍼 할당의 목표이다. 다음 식에서 HIT는 적중률, B는 시스템 전체의 버퍼 캐쉬 크기, m은 현재 수행중인 응용의 수, B_j는 j번째 응용에게 할당된 버퍼 캐쉬 크기를 나타낸다.

$$\sum_{j=0}^{m} HIT_j(B_j), \quad \text{단 } \sum_{j=0}^{m} B_j = B$$

블록 참조 패턴에 기반한 적응력있는 블록 교체 기법에서, 각 참조 패턴의 버퍼 적중률은 다음과 같이 계산될 수 있다.

o 순차 참조: $HIT_{sequential}(B_j) = 0$

o 반복 참조: 반복 참조의 크기가 L이고 블록들에 대한 전체 참조 횟수가 R(따라서 반복 횟수는 R/L이 된다) 인 응용 j에 대하여 MRU 교체 정책을 적용하였을 때 적중률은 다음과 같다.

$$HIT_{looping}(B_j) = \frac{\min[L, B_j] * (R/L)}{R} - \frac{\min[L, B_j]}{R} = \frac{\min[L, B_j]}{L} - \frac{\min[L, B_j]}{R}$$

이때 $\min[L, B_j]/R$는 블록들이 처음 참조될 때 발생하는 비적중률(cold miss ratio)이다.

o 시간적 지역성을 보이는 참조: 이 참조 패턴은 LRU 스택 모델[3]로 모델링 될 수 있으며, 각 블록 b_i은 최근에 참조된 시간에 따라 LRU 스택 상의 한 위치에 존재하게 되며 스택의 각 위치에 따라 독립적인 참조 확률 a_i을 갖는다(i가 작을수록 최근에 참조된 블록으로 스택의 상단에 위치하며 $a_i \geq a_{i+1}$임을 가정). 따라서 시간적 지역성을 보이는 참조 패턴을 갖는 응용 j에 대하여 LRU 교체 정책을 적용하였을 때 적중률은 다음과 같다.

$$HIT_{temporal-clustered}(B_j) = \sum_{i=1}^{B_j} a_i - \frac{\min[N, B_j]}{R}$$

이때 N는 참조된 블록의 수이며 R은 전체 참조 횟수이다. 그리고 $\min[N, B_j]/R$는 블록들이 처음 참조될 때 발생하는 비적중률이다.

여기서 a_i는 여러 가지 방법으로 설정될 수 있으며, Belady의 lifetime function이 시간적 지역성을 보이는 특성을 보이는 실제 응용들의 블록 참조 행위를 잘 근사시키는 것으로 알려져 있다[21, 22]. Belady의 lifetime function은 응용이 N개의 서로 다른 블록을 참조할 때, 다음의 식으로 a_i를 결정한다.

$$A_i = a_1 + a_2 + \ldots + a_i = 1 - c * i^{-k}, \quad \text{단 } 1 \leq i \leq N$$

이 식은 $a_i \geq a_{i+1}$이라는 LRU 스택 모델의 가정을 만족한다. 이때 c와 k는 시간적 지역성을 보이는 정도(degree of locality)를 제어하는 인수이며 k가 클수록

지역성이 크다.

o 확률 참조: 이 참조 패턴은 독립 참조 모델[3]로 모델링 될 수 있으며, 각 블록 b_i는 고유한 참조 확률 p_i을 갖는다(i가 작을수록 참조 횟수가 큰 블록이며 $p_i \geq p_{i+1}$임을 가정). 따라서 응용 j에 대하여 LFU 교체 정책을 적용하였을 때 적중률은 다음과 같다.

$$HIT_{probabilistic}(B_j) = \sum_{i=1}^{B_j} p_i - \frac{\min[N, B_j]}{R}$$

이때, $\min[N,B_j]/R$는 블록들이 처음 참조될 때 발생하는 비적중률이다.

p_i는 Zipfian 확률 분포를 이용해 잘 근사 될 수 있는 것으로 알려져 있으며[6], 이 분포는 다음 식으로 각 p_i를 결정한다.

$$P_i = p_1 + p_2 + \ldots + p_i = (i/N)^{\log \alpha / \log \beta}, \quad \text{단 } 1 \leq i \leq N$$

이때 α와 β는 0~1 사이 값을 가지며, 참조 확률의 비대칭(skewness) 정도를 제어하는 변수이다. 예를 들어 α=0.7, β=0.2라면 전체 참조 요청(R)의 70%가 전체 블록(N)의 20%에 대해서 이루어진다는 의미이다.

4. 버퍼 할당 정책

적응력있는 블록 교체 기법은 각 응용마다 그 응용에 적합한 서로 다른 블록 교체 기법을 적용한다. 이를 위해 그림 2에 나타나 있는 것처럼, 기존의 버퍼 관리 부분을 ACM (Application Cache Manager)과 SCM(System Cache Manager)이라는 두 부분으로 나누었다. ACM은 각 응용별로 존재하며 블록 참조 패턴을 발견하고 그에 적합한 교체 정책을 적용한다. SCM은 시스템에 하나 존재하며 버퍼 할당, 캐쉬 블록에 대한 정보 유지 등을 담당한다. 블록 교체는 ACM과 SCM이 상호 협력하여 수행한다. 응용이 요청한 블록이 버퍼 캐쉬에 없어 새로운 캐쉬 버퍼가 필요할 경우, 그 응용의 ACM은 SCM에게 새로운 블록을 요청한다. 이때 자유 버퍼가 없다면 SCM은 블록 교체 요청을 어떤 ACM에게 보낼 것인가를 결정해야 한다(그림 2의 단계 2).

이는 새로운 블록을 위해 어떤 버퍼를 할당할 것인가 하는 자원 분배의 문제로, 본 논문에서는 블록 전체의 적중률을 극대화시킬 수 있는 적중률 분석 기반 블록 할당(Block Allocation based on Analytic Prediction of Hit Ratio) 기법을 제안한다. 본 기법에서 각 ACM은 버퍼 적중률 계산식을 기반으로, 자신에 할당된 버퍼가 하나 줄어들 경우에 버퍼 적중률의 감소 예상치를 평가하여 SCM에 제공한다. SCM은 적중률의 감소 예상치가 가장 적은 ACM을 선택하여 블록 교체를 요청한다. 즉, 각 응용 j에 대하여 $\triangle HIT_j(B_j) = HIT_j(B_j) - HIT_j(B_j - 1)$를 계산하여 $\triangle HIT_j(B_j)$이 가장 적은 응용을 선택하여 블록 교체를 요청하는 것이다. 이러한 버퍼 할당 기법은 시스템 전체의 적중률을 극대화시킬 수 있다.

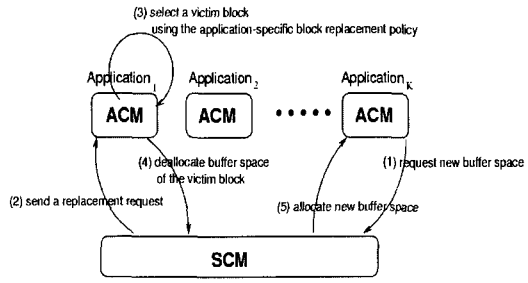


그림 2 캐쉬 블록 할당 기법

각 응용의 $\triangle HIT_j(B_j)$는 다음과 같은 방법으로 온라인으로 계산할 수 있다. 순차 참조의 경우 캐쉬 크기와 관계없이 적중률이 항상 0이므로 $\triangle HIT_{sequential}(B_j)$는 0이다. 반복 참조의 경우 $\triangle HIT_{looping}(B_j) = (\min[L,B_j] - \min[L,B_j-1])/L - (\min[L,B_j] - \min[L,B_j-1])/R$ 이다. 이때 $(\min[L,B_j] - \min[L,B_j-1])/R$은 초기 비적중률을 고려하기 위한 식으로, 만일 $L \geq B_j$라면 $1/R$이 되며, 그 외에는 0이 된다. 반복 참조의 적중률은 응용에게 할당된 캐쉬 크기 B_j과 반복 참조의 크기 L에 의해 큰 영향을 받으며, L은 블록 참조 패턴을 발견할 때 구할 수 있다.

시간적 지역성을 보이는 참조의 경우, $\triangle HIT_{temporally-clustered}(B_j) = \sum_{i=1}^{B_j} a_i - \sum_{i=1}^{B_j-1} a_i - [1/R \text{ 또는 } 0] = a_{B_j} - [1/R \text{ 또는 } 0]$ ($N \geq B_j$이면 $1/R$, 아니면 0)이다. 이때 a_i는 가상적인 LRU 스택을 이용해 온 라인으로 구할 수 있다. 이 방법은 블록 참조가 발생할 때마다 LRU 스택 상에서 참조된 위치를 실제 측정하는 것이다. 즉 응용이 블록을 R번 참조하였다면, $\hat{a}_i$=(i번째 스택 위치의 참조 횟수)/R을 측정하여 스택의 참조 확률 a_i를 평가하는 것이다. 2개 이상의 $\hat{A}_i$ ($= \hat{a}_1 + \hat{a}_2 + \ldots + \hat{a}_i$)를 측정하면 Belady의 lifetime function의 제어 변수인 c와 k를 근사할 수 있으며, 따라서 모

든 i에 대한 a_i를 평가할 수 있다.

확률 참조의 경우, $\triangle HIT_{probabilistic}(B_j) = \sum_{i=1}^{B_j} p_i - \sum_{i=1}^{B_j-1} p_i - [1/R \text{ 또는 } 0] = p_{B_j} - [1/R \text{ 또는 } 0]$이다. 이 경우에도 p_i는 가상적인 버퍼 리스트를 이용해 구할 수 있다. 버퍼 리스트에는 블록들이 LFU 순서대로 위치되는데, 블록 참조가 발생할 때마다 리스트 상에서 참조된 위치를 실제 측정한다. 즉, 응용이 R번의 참조 요청을 하였다면 $\hat{p}_i$=(i번째 블록의 참조 횟수)/ R을 측정하고 이를 기반으로 Zipfian 확률 분포의 제어 변수인 α, β를 구한다. 예를 들어 100개의 블록이 버퍼 리스트 상에 존재하고 $\hat{P}_{10} = p_1 + p_2 + \cdots + p_{10} = 0.7$이라면 $\hat{\alpha}$ =0.7, $\hat{\beta}$ =0.1로 유도된다. $\hat{\alpha}$, $\hat{\beta}$가 구해지면 모든 i에 대한 p_i를 평가할 수 있다.

제안한 버퍼 할당 정책의 설계 시, 추가로 고려해야 할 요소는 블록 참조 패턴이 발견되지 않은 응용들에 대한 예상 적중률 평가이다. 본 기법에서는 블록 참조 패턴이 발견되지 않은 응용들에 대해서는 LRU 기법을 적용하며, 따라서 블록 할당 기법도 시간적 지역성을 보이는 응용에 적용하는 적중률 계산식을 이용하여 예상 적중률을 평가한다.

5. 성능 분석

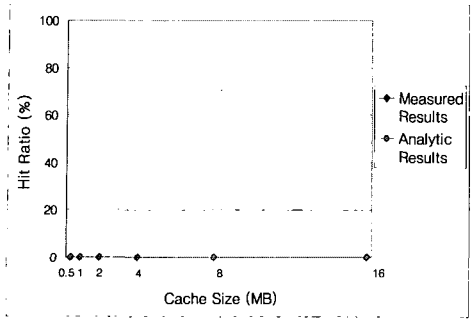
본 장에서는 우선 제안한 적중률 모델의 유효성을 검증하기 위해, 가상적인 블록 참조 패턴(synthetic workload)을 생성하여 각 블록 참조 패턴에 대한 적중률 계산에 의한 결과와 실제 적중률 측정 결과를 비교한다. 그리고 적중률 분석 기반 블록 할당 기법을 생성된 블록 참조 트레이스와 실제 응용 트레이스에 적용했을 때의 성능 분석 결과에 대해서도 기술한다.

5.1 버퍼 적중률 모델의 검증

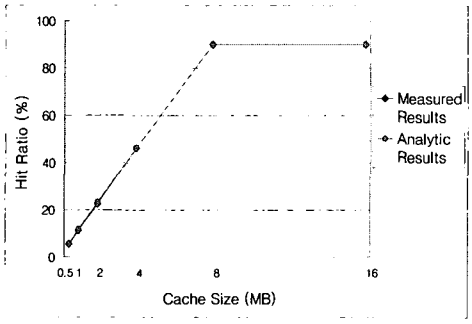
제안된 적중률 계산식들을 검증하기 위해, 먼저 synthetic workload generator를 작성하여 가상적인 블록 참조 트레이스 4개를 생성하였다. 각 트레이스는 10,000번의 블록 참조 요청으로 구성된다. 트레이스 1에서는 순차적으로 블록들이 참조되며 N = 10,000이다. 트레이스 2에서는 1,000개의 블록들이 일정한 간격으로 반복 참조되며 N=1,000, 반복 횟수는 10이다. 트레이스 3에서는 LRU 스택 모델에 따라 블록들이 참조되며 c =1, k=0.8 그리고 N = 2,000으로 설정되었다. 트레이스 4에서는 블록 b_i마다 고유한 확률 p_i에 따라 참조되도록 생성되었으며, α =0.7, β =0.3(이때 $\log\alpha/\log\beta$ =0.138이 된다), N=2,000으로 설정되었다.

생성된 4개의 트레이스에 대하여, "제시된 블록 교체 기법을 적용했을 때의 캐쉬 적중률"(Measured results)과 "제안된 적중률 계산식에 의해 계산된 결과"(Analytic results)가 그림 3에 나타나 있다. 모의 실험에서 블록 크기는 8KB로 가정하였다. 그리고 각 응용의 블록 참조 패턴의 발견은 주기적으로 수행되며, 본 논문에서 제시된 실험 결과는 발견 주기의 길이를 500으로 설정했을 때의 결과이다 (제안된 기법에서 발견 주기의 길이와 하위 리스트의 개수는 블록 참조 패턴의 발견 결과에 영향을 줄 수 있는 제어 변수이다. 실험 결과 발견 주기의 길이가 250~1000이고 하위 리스트의 개수가 3~7이면 각 응용의 블록 참조 패턴을 정확히 발견할 수 있는 것으로 분석되었다. 이에 대한 자세한 분석 내용은 [20, 23]에 기술되어 있다). 발견 결과 트레이스 1, 2, 3, 4에 대하여 각각 순차 참조, 반복 참조, 시간적 지역성을 보이는 참조, 그리고 확률 참조로 각각 발견하였다 (생성된 각 트레이스의 블록 참조 그래프와 발견 결과는 [23]에 분석되어 있다). 제안된 적응력있는 블록 교체 기법은 발견된 참조 패턴에 최적인 블록 교체 기법을 적용하며, 따라서 트레이스 1과 2에 대해서는 MRU 교체 정책을, 트레이스 3에 대해서는 LRU 교체 정책을, 트레이스 4에 대해서는 LFU 교체 정책을 적용한다. 또한 각 트레이스에 대해 참조 패턴을 기반으로 적중률을 계산에 필요한 제어 변수들을 온라인으로 구한다. 트레이스 2의 경우 반복 참조의 크기(L)가 1,000으로 탐지되며, 트레이스 3에 대해서는 $\hat{c}$=1.2, $\hat{k}$=0.738로 계산되었고, 트레이스 4에 대해서는 $\log(\alpha)/\log(\beta)$=0.156으로 계산되었다.

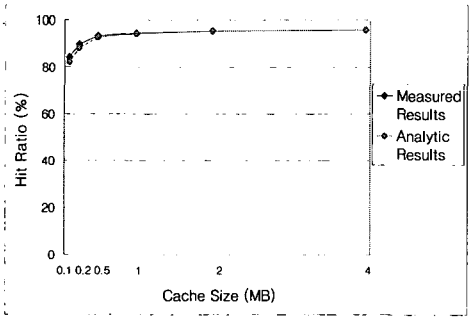
구체적으로 제어변수를 온라인으로 구하는 예는 다음과 같다. 우선 c와 k를 구하기 위해 4개의 가상 스택 버퍼를 사용한다. 이 버퍼들은 각각 최근에 참조된 블록들을 10, 20, 30, 50개씩 유지하면서(따라서 LRU 스택의 크기가 각각 10, 20, 30, 50이 된다), 새로운 블록이 참조되면 그 블록이 각 가상 버퍼에 있는지 여부를 조사하여, 있는 경우 적중 횟수를 증가시킨다. 발견 주기가 끝나면 각 버퍼의 적중 횟수를 이용해 $\hat{A}_{10}$, $\hat{A}_{20}$, $\hat{A}_{30}$, $\hat{A}_{50}$을 유도한다. 2개의 $\hat{A}_i$가 있으면 연립 방정식을 이용해 $\hat{c}$와 $\hat{k}$을 구할 수 있다. 이 예에서는 6개의 $\hat{c}$와 $\hat{k}$을 구할 수 있으며($_4C_2$), 그 평균을 lifetime function의 제어 변수로 설정하였다. 한편, α와 β도 참조 빈도가 가장 큰 블록들을 유지하는 4개의 가상 버퍼를 이용하여 $\hat{P}_{10}$, $\hat{P}_{20}$, $\hat{P}_{30}$, $\hat{P}_{50}$을 먼저 유도하여 구할 수 있다. c와 k, $\log(\alpha)/\log(\beta)$를 계산하기 위해 각각 4개의 가상



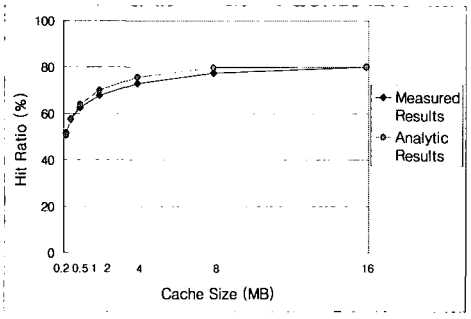
(a) 트레이스 1 (순차 참조)



(b) 트레이스 2 (반복 참조)



(c) 트레이스 3 (시간적 지역성을 보이는 참조)



(d) 트레이스 4 (확률적 참조)

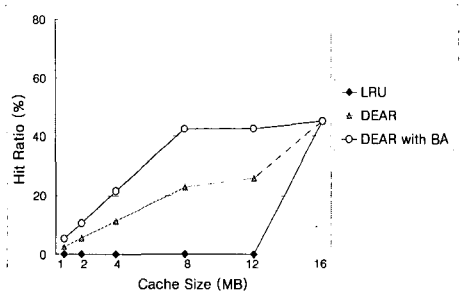
그림 3 각 블록 참조 패턴의 캐쉬 적중률 계산 결과와 실제 측정 결과 비교

버퍼를 사용하였는데, 가상 버퍼는 실제 캐쉬 블록을 갖는 것이 아니라 블록의 정보만 유지하므로 메모리 부하는 매우 적다. 실제 각 블록마다 vnode 번호, 블록 번호, 참조된 블록을 찾기 위한 해쉬 리스트, 블록들을 관리하기 위한 이중 연결 리스트만 있으며, 각 블록마다 20 바이트가 필요하고 전체 가상 버퍼에는 최대 20bytes×(10+20+30+50)×2=4,400 바이트가 필요하다. 이 크기는 캐쉬 블록 하나의 크기(8KB)보다 작다.

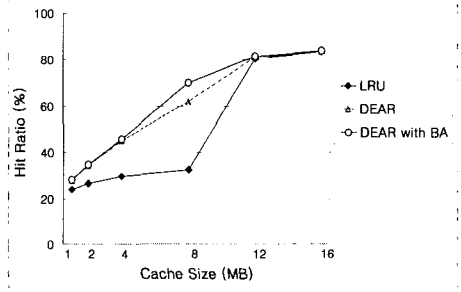
그림 3은 제안한 적중률 계산식이 실제 블록 교체 기법의 버퍼 적중률과 거의 일치함을 보여준다. 순차 참조와 반복 참조, 시간적인 지역성을 보이는 참조 패턴의 경우에는 계산 결과(Analytic results)와 실제 적중률(Measured results)이 거의 일치한다. 확률 참조의 경우에는 계산 결과가 실제 측정 결과보다 조금 크다. 이는 적중률 계산식은 항상 참조 확률이 가장 작은 블록을 교체하는 것을 가정하지만, 적응력있는 블록 교체 기법에서 LFU를 적용할 때 참조 횟수가 블록의 참조 확률을 정확히 반영하지 못하는 경우(불안정 상태로 응용의 수행 초기에 발생할 수 있다) LFU 정책이 참조 확률이 더 높은 블록을 교체하는 경우가 발생하기 때문이다.

그림 4는 생성된 트레이스들이 두 개 이상 동시에 수행되는 환경에서 LRU, DEAR, **DEAR-BA**(DEAR with Block Allocation based on Analytic Prediction of Hit Ratio: 적중률 분석 기반 블록 할당) 기법을 적용했을 때의 실험 결과이다. LRU 기법은 버퍼 캐쉬를 기존의 전역적인 LRU 방식으로 교체하는 것을 나타낸다. DEAR와 DEAR-BA 기법은 각 응용의 블록 참조 패턴을 발견하여 응용 별로 최적의 블록 교체 기법을 적용하는 면에서는 동일하다. 그러나, 특정 ACM에게 블록 교체 요청을 보낼 때 DEAR는 시스템 전체적으로 가장 오래 전에 참조된 블록을 소유한 ACM에게 교체를 요청하지만, DEAR-BA는 예상 적중률이 가장 적은 ACM에게 교체 요청을 보낸다는 점에서 다르다.

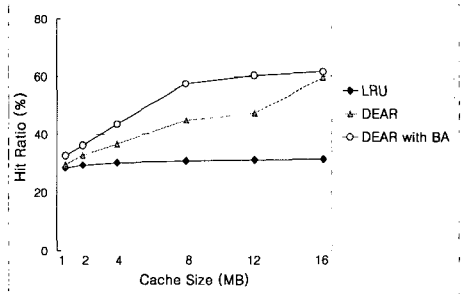
그림 4의 (a)는 트레이스 1과 트레이스 2가 동시에 수행되는 환경에서 각 기법의 성능 결과를 보여 준다. 순차 참조인 트레이스 1에 대해서는 세 기법 모두 적중률이 0이다. 반복 참조인 트레이스 2에 대해서, 작업 집합이 모두 캐쉬에 올라오기 전까지 LRU 기법은 적중률을 증가시키지 못한다. 반면에 DEAR 기법은 MRU 교체 정책을 적용하므로 캐쉬 크기가 증가함에 따라 적중률 증가를 얻을 수 있다. DEAR-BA 기법은 MRU 교체 정책을 적용할 뿐만 아니라 트레이스 1이 차지하던 캐쉬 블록을 우선적으로 교체하고 이 공간을 트레이스



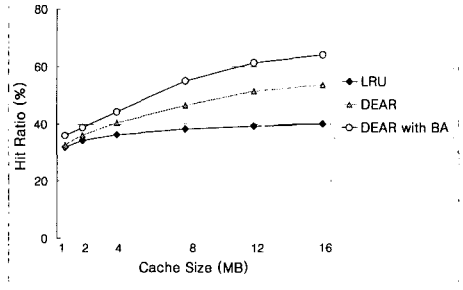
(a) 순차+반복 트레이스의 조합



(b) 반복+확률 트레이스의 조합



(c) 순차+반복+시간적 지역성을 보이는 참조 트레이스의 조합



(d) 순차+반복+확률+시간적 지역성을 보이는 참조 트레이스의 조합

그림 4 생성된 트레이스들 조합에서의 성능 비교

2에게 할당하므로 더욱 큰 적중률 증가를 보인다. 그림 4의 (b)와 같이 트레이스 2와 트레이스 4가 동시에 수행되는 환경에서, DEAR 기법은 각 트레이스에게 최적인 블록 교체 기법을 적용하기 때문에 LRU 기법에 비해 좋은 성능을 보인다. 한편 DEAR-BA 기법은 캐쉬 크기가 8MB일 때 버퍼 캐쉬를 트레이스 2에게 더욱 많이 할당하여 DEAR 기법보다 더 좋은 성능을 보인다. 이는 버퍼 크기가 클(4MB 이상) 때 추가로 버퍼를 더 할당할 경우, 확률 참조 트레이스보다 반복 참조 트레이스에서 더 큰 적중률 향상을 얻을 수 있기 때문이다(그림 3의 (b)와 (d) 참조). 트레이스 1, 2, 4가 동시에 수행되는 환경(그림 4의 (c))이나 트레이스 1, 2, 3, 4가 모두 수행되는 환경(그림 4의 (d))에서도 DEAR-BA 기법이 각 트레이스에게 최적의 블록 교체 기법을 적용함과 동시에 모든 트레이스의 적중률이 극대가 되도록 블록을 할당하기 때문에 가장 좋은 성능을 보인다.

5.2 실제 응용에서 성능 측정

본 절에서는 실제 응용들을 이용한 적중률 분석 기반 블록 할당 기법의 성능 분석 내용을 기술한다. 이 실험에서는 wc, cscope, AB(Andrew file system Benchmark), 그리고 cpp 응용의 트레이스를 이용하였다. 고려한 응용들 중에서 wc는 입력 파일의 문자 수, 단어 수, 라인 수 등을 보여주는 응용이고, cscope는 대화형 C 소스 코드 탐색 도구로 C 언어에 대한 질의가 요청될 때마다 '*cscope.out*'이라는 파일을 읽어 질의에 대답하는 응용이다. 한편 cpp는 C 컴파일러 전 처리기이고, AB는 Andrew 파일 시스템을 개발할 때 사용한 벤치마크로 여러 파일에 대한 생성, 복사, 검색, 컴파일 등의 작업을 수행하는 응용이다.

본 논문에서 제안한 블록 참조 패턴 발견 기법은 wc, cscope, AB, cpp응용들에 대해 각각 순차 참조, 반복 참조, 시간적 지역성을 보이는 참조, 그리고 확률 참조 패턴을 갖는 것으로 발견하였다 [23]. 제안된 기법이 각 응용의 블록 참조 패턴을 얼마나 정확히 발견하였는가를 확인하기 위해, 본 논문에서는 각 트레이스의 블록 참조 그래프와 참조 패턴 발견 결과를 그림 5에 도시하였다. 그림 5의 각 그래프에서 x축은 가상 시간이며 y축은 그 시간에 참조된 블록의 논리적인 번호이다. 본 논문에서 가상 시간은 블록이 참조될 때마다 한 단위씩 증가한다고 가정하였으며, 이는 각 응용 (또는 프로세스) 마다 시간 정보를 유지하는 변수를 선언하고 각 응용이 블록에 대한 요청을 발생할 때마다 커널 수준에서 이를 파악하여 (UNIX의 버퍼 캐쉬 관리 함수 중 *bread*나 *breadn* 함수에서 이 요청에 대한 파악이 가능)

선언된 가상 시간 변수를 1씩 증가시킴으로써 실제 구현 가능하다.

제안된 기법의 블록 참조 패턴 발견 결과는 각 그래프의 상단에 도시되었다. 그림 5는 제안된 기법의 블록 참조 패턴 발견 결과와 각 트레이스의 블록 참조 그래프가 잘 일치함을 보여준다. Wc 응용의 경우 제안된 기법은 일관되게 순차 참조로 발견하며, 이는 wc 응용이 입력 파일의 문자 수, 단어 수, 라인 수를 셀 때 입력 파일을 순차적으로 읽는 특성이 있기 때문이다. Cscope 응용의 경우 제안된 기법은 가상 시간이 1600 이전일 때에는 순차 참조로 발견하며, 그 이후에는 반복 참조로 발견한다. 이것은 cscope 응용이 C 프로그램에 대한 질의가 발생할 때마다 '*cscope.out*' 이라는 파일을 순차적으로 탐색하여 질의에 응답하는 특성에 기인하며, 결국 제안된 기법이 cscope 응용의 블록 참조 패턴을 정확히 발견하였음을 의미한다. AB 응용의 경우 제안된 기법의 참조 패턴 발견 결과는 발견 시간에 따라 다르다. 전체적으로 시간적 지역성을 보이는 패턴을 주된 참조 패턴으로 발견하지만, 가상 시간 3000에는 반복 참조로 발견하며 가상 시간 4000에는 확률 참조로 발견한다. 실제 AB 응용의 블록 참조 그래프를 분석하면, 전체적으로 참조되는 블록들이 시간이 지남에 따라 계속 변하며 (즉 아주 먼 과거에 참조된 블록은 거의 다시 참조되지 않으며), 한번 참조된 블록들은 짧은 시간 간격 동안 다시 참조되는 경향이 있다 (특히 가상 시간 5000 이후에는 이 경향이 뚜렷하다). 한편 가상 시간 3000에

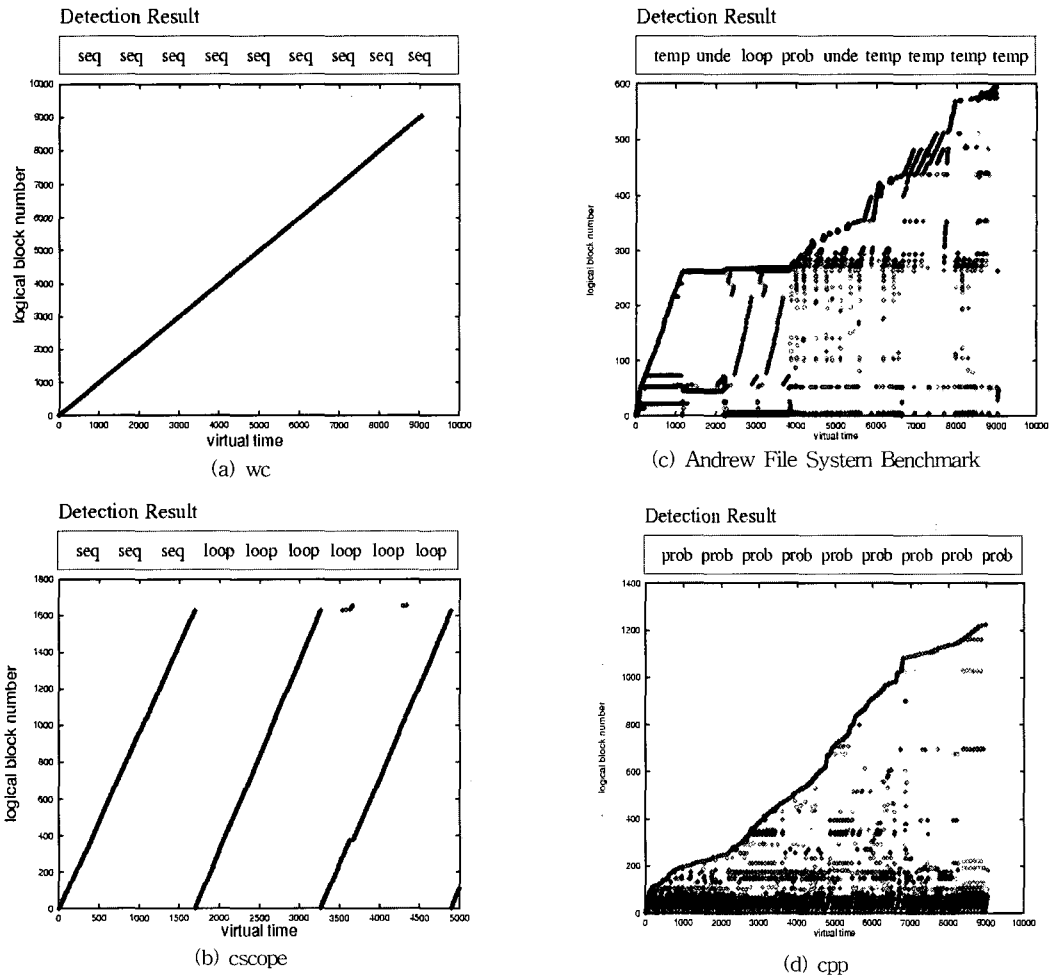
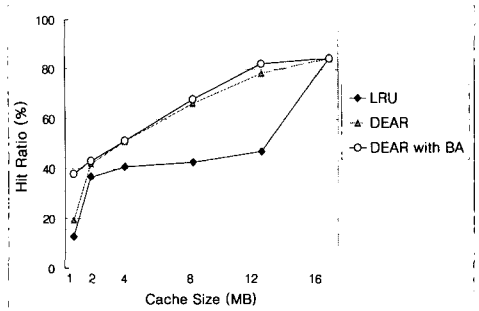
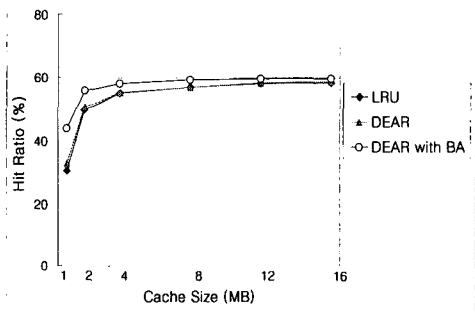


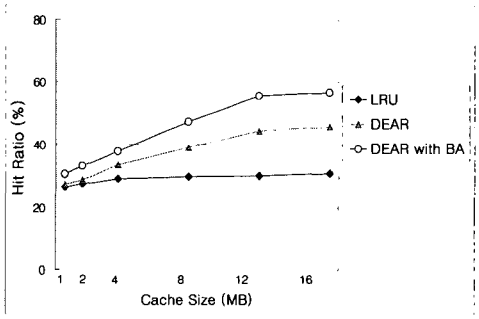
그림 5 응용들의 블록 참조 그래프와 발견된 참조 패턴



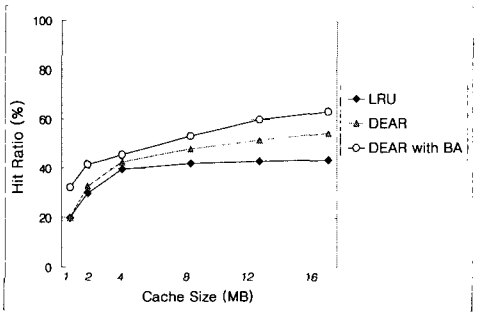
(a) cscope+cpp



(b) wc+AB+cpp



(c) wc+cscope+AB



(d) wc+cscope+AB+cpp

그림 6 실제 응용에서의 성능 비교

는 일부 블록들이 일정한 간격으로 재 참조된다. 결국 그림 5의 (c)는 블록 참조 패턴 발견 결과와 응용의 블록 참조 그래프가 잘 일치함을 보여주며, 이는 제안된 기법이 응용의 참조 패턴 변화를 인식할 수 있고 그 변화를 정확히 발견함을 의미한다. Cpp 응용의 경우 제안된 기법은 일관되게 확률 참조로 발견하며, 실제로 cpp 응용은 C 헤더 파일들을 C 코드 파일들에 비해 더 높은 확률로 참조한다.

그림 6의 (a)는 cscope와 cpp를 동시에 수행시켰을 때 각 기법의 성능을 보여준다. DEAR 기법은 cscope와 cpp 응용에게 최적인 MRU 기법과 LFU 기법을 적용하므로 LRU 기법에 비해 더 좋은 성능을 보인다. DEAR-BA 기법은 최적의 블록 교체 기법을 적용할 뿐만 아니라 전체 응용의 적중률이 극대가 되도록 캐쉬 블록을 할당하기 때문에 가장 좋은 성능을 보인다. DEAR-BA 기법을 자세히 분석한 결과 캐쉬 크기가 적을 때(1~2MB일 때) cpp에게 더 많은 캐쉬 버퍼를 할당하며, 캐쉬 크기가 클 때(8~12MB일 때) cscope에게 더 많은 캐쉬 버퍼를 할당하여 전체적인 캐쉬 적중률을 높임을 알 수 있었다. 이는 더 큰 적중률 향상을 얻기 위해서 캐쉬 크기가 작을 때는 확률 참조에 블록을 더 많이 할당하는 것이 좋고, 캐쉬 크기가 클 때는 반복 참조에 블록을 더 많이 할당하는 것이 좋기 때문이다(그림 3 참조).

그림 6의 (b)에서와 같이 wc, AB, cpp가 동시에 수행되는 환경에서 캐쉬 크기가 클 경우, DEAR 기법은 LRU 기법에 비해 성능 향상이 그렇게 크지 않다. 이것은 DEAR 기법이 cpp에게 최적 블록 교체 기법인 LFU를 적용하지만 캐쉬 크기가 클 경우 그 이익은 크지 않으며, AB와 wc에 대해서는 LRU 기법과 같은 적중률을 얻기 때문이다(wc는 순차 참조이므로 적중률 이득이 없고 AB에 대해 LRU 정책을 적용한다). DEAR-BA 기법은 wc가 차지하는 캐쉬 공간을 우선적으로 교체하기 때문에 cpp와 AB의 적중률을 더 높일 수 있으므로 최고의 성능을 보인다. Wc, AB, cscope가 동시에 수행되는 환경(그림 6의 (c))과 4개의 응용이 동시에 수행되는 환경(그림 6의 (d))에서도 DEAR-BA 기법이 각 응용들에게 최적의 블록 교체 기법을 적용함과 동시에 모든 응용의 적중률이 극대가 되도록 블록을 할당하기 때문에 가장 좋은 성능을 보인다.

본 논문에서 제안된 버퍼 할당 정책을 구현할 때 부가되는 계산 부하(computational overhead)에는 크게 다음과 같은 3가지가 있다. 첫째는 주기적으로 응용의 블록 참조 패턴을 발견하기 위한 부하이며, 둘째는 참조

패턴을 발견할 때 사용하는 블록의 속성 정보(과거 참조 거리와 참조 횟수)와 미래 참조 거리 정보를 블록이 참조 될 때마다 수집하는 부하이다. 그리고 세번째 부하는 버퍼 할당 시에 각 응용의 적중률 감소를 계산하기 위한 계산 부하이다. 본 연구진은 이미 [20]의 구현 연구를 통해 첫째와 둘째 부하는 그리 크기 않음을 정량적으로 분석하였다. 세번째 부하인 적중률 감소 계산을 위해 예상되는 부하는 다음과 같다.

제안된 버퍼 할당 정책은 버퍼 할당 시에 각 응용의 예상 버퍼 적중률을 계산하며, 이 계산은 발견된 응용의 블록 참조 패턴을 기반으로 수행된다. 예를 들어 반복 참조의 경우 반복 참조의 크기(L)와 전체 참조 횟수(R), 그리고 현재 응용에게 할당된 버퍼의 크기(B_j)에 의해 버퍼의 예상 적중률을 계산한다($HIT_{looping}(B_j) = \min[L, B_j]/L - \min[L, B_j]/R$). 이때 R은 변수를 하나 선언하고 응용으로부터 참조가 발생할 때마다 증가시키면 온라인으로 얻을 수 있는 정보이며, B_j도 변수를 선언하고 응용에게 버퍼 블록이 할당될 때마다 (회수될 때마다) 증가시키면 (감소시키면) 얻을 수 있는 정보이다. 또한 L은 블록 참조 패턴이 반복 참조로 발견될 때 얻을 수 있는 정보이다 (L을 구하기 위한 계산 부하는 버퍼 할당 시에 매번 부가되는 것이 아니라, 발견 주기마다 - 본 논문에서는 500 가상 시간마다 - 부가된다). 따라서 버퍼 할당 시 매번 부가되는 계산 부하는 적중률의 감소를 계산하기 위한 부하인 $(\min[L, B_j] - \min[L, B_j - 1])/L - (\min[L, B_j] - \min[L, B_j - 1])/R$ 이다.

다른 참조 패턴의 경우에도 반복 참조의 경우처럼 적중률의 감소를 계산하기 위한 부하가 매번 버퍼 할당 시에 부가된다. 시간적 지역성을 보이는 참조의 경우 $a_{B_j} - [1/R \text{or } 0]$의 계산 부하가 필요하며, 이때 a_{B_j}를 계산하기 위한 Belady's lifetime function의 제어 변수들의 (c와 k) 값은 응용의 참조 패턴이 시간적 지역성을 보이는 참조 패턴으로 발견될 때 계산된다 (이 제어 변수들의 값도 발견 주기마다 계산되며 버퍼 할당 시에 매번 계산되는 것은 아님에 주의). 또한 c와 k값을 계산하기 위해 가상적인 LRU 스택이 이용되는데, [12]의 연구에 따르면 가상적인 LRU 스택을 이용한 적중률 계산은 큰 계산 부하 없이 가능함을 보여주고 있다. 확률 참조의 경우는 시간적 지역성을 보이는 참조와 유사한 방법으로 적중률 감소가 계산되며, 순차 참조의 경우 예상 적중률이 항상 0이므로 계산을 위한 부가적인 부하는 없다.

이러한 분석 내용을 고려할 때 본 연구진은 버퍼 할당 시에 부가되는 계산 부하가 그리 크지 않을 것으로 예상하며 이후 실제 시스템에 제안된 기법을 구현하여 계산 부하를 정량적으로 평가할 계획이다.

6. 결 론

최근 여러 연구에서 응용의 블록 참조 패턴에 대한 사용자 수준 정보를 이용하는 적응력있는 블록 교체 기법이 제안되었으며, 이러한 기법이 최적에 가까운 블록 교체 성능을 제공할 수 있음을 보였다. 하지만 사용자 수준의 정보를 얻고 이를 시스템에 알려주는 것은 쉬운 일이 아니며, 이는 다양한 종류의 많은 응용들에게 적응력있는 블록 교체 기법을 제공하지 못하게 하고 있다. 본 논문에서는 먼저, 시스템 수준에서 블록의 속성과 미래 참조 거리 간의 관계를 기반으로 각 응용의 블록 참조 패턴을 자동으로 발견하고, 발견된 참조 패턴에 적합한 블록 교체 정책을 적용하는 기법을 제안하였다. 제안된 기법은 순차 참조, 반복 참조, 시간적인 지역성을 보이는 참조, 확률 참조 패턴을 발견할 수 있으며, 응용 수행 중 참조 패턴의 변화가 발생하면 이를 인식할 수 있다. 한편, 어떤 응용이 접근하고자하는 블록이 캐쉬에도 없고 자유 블록도 없다면 블록을 할당하기 위해 커널은 어떤 캐쉬 버퍼를 교체해야 한다. 이러한 경우 어떤 블록을 교체하여 할당할 것인가에 관한 기법도 제시하였다. 새로운 블록 할당이 요구될 때마다, 시스템은 각 응용 별로 자신에 할당된 버퍼가 하나 줄어들 경우에 버퍼 적중률의 감소 예상치를 분석하여, 적중률의 감소 예상치가 가장 적은 응용을 선택하여 블록 교체를 요청한다. 이러한 방식은 시스템 전체의 버퍼 블록의 적중률을 극대화시켜 준다. 각 응용의 버퍼 적중률은 그 응용의 블록 참조 패턴에 따라 온라인으로 계산된다. 제안된 기법을 여러 트레이스들에 대해 적용하여 실험한 결과, 기존의 다른 블록 할당 기법들보다 좋은 성능을 보여 주었다.

실제 시스템 환경에서 본 기법에 의해 도움을 받지 못하는 응용들이 존재할 수 있다. 단지 몇 개의 블록만을 참조하는 응용들이나 임의 블록 참조 패턴을 보이는 응용들이 대표적인 예이다. 현재 그러한 응용들에 대해서는 기존의 LRU 정책을 채택하고 있으며, 블록 참조 패턴을 오프라인으로 발견하여 응용이 시작될 때부터 적응력있는 블록 교체를 적용하는 방안에 대해 연구하고 있다. 또한, 발견된 참조 패턴 정보를 이용한 선반입 기법에 대해서도 연구할 것이다. 블록 참조 패턴 정보는 어떤 블록들이 가까운 미래에 참조될 것인가에 대한 정보를 제공할 수 있으며 어떤 블록을 선 반입해야 하는

지를 결정할 수 있다. 블록 할당 정책과 선반입 정책을 적절하게 결합하면 시스템의 성능을 더욱 높일 수 있을 것이다.

참 고 문 헌

[1] A. J. Smith, "Disk cache-miss ratio analysis and design considerations," ACM Trans. on Computer Systems, 3(3), pp 161-203, August 1985.

[2] P. M. Chen, et al., "Raid: High-Performance, Reliable Secondary Storage," ACM Computing Surveys, 26(2), pp 145-182, June 1994

[3] E. G. Coffman and P. J. Denning, Operating Systems Theory, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a Distributed File System," Proceedings of the 13th Symposium on Operating System Principles, pp 198-212, 1991.

[5] J. T. Robinson and M. V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," Proc. of the 1990 ACM SIGMETRICS Conf., pp 134-142, 1990

[6] E. J. O'Neil, P. E. O'Neil and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," Proc. of the 1993 ACM SIGMOD Conf., pp 297-306, 1993

[7] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," Proc. of the 1995 ACM SIGMETRICS Conference, pp 291-300, 1995

[8] D. Lee, J. Choi, H. Choe, S. H. Noh, S. L. Min, and Y. Cho, "Implementation and Performance Evaluation of the LRFU Replacement Policy," Proceedings of the 23th Euromicro Conference, pp 106-111, 1997.

[9] K. Korner, "Intelligent Caching for Remote File System," Proceedings of the 10th international Conference on Distributed Computing Systems, pp 220-226, May 1990.

[10] C. D. Tait and D. Duchamp, "Detection and Exploitation of File Working Sets," Proceedings of the 11th International Conference on Distributed Computing Systems, pp 2.-9, May 1991.

[11] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," Proceedings of the 1997 ACM SIGMETRICS Conference, pp 115-126, June 1997.

[12] Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed Prefetching and Caching," Proceedings of the 15th Symposium on Operating System Principles, pp 1-16, 1995.

[13] P. Cao, E. W. Felten, and K. Li, "Implementation and Performance of Application Controlled File Caching," Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation, pp 165-178, 1994.

[14] C. Faloutsos, R. Ng, and T. Sellis, "Flexible and Adaptable Buffer Management Techniques for Database Management Systems," IEEE Transactions on Computers, 44(4):546-560, April 1995.

[15] T. C. Mowry, A. K. Demke, and O. Krieger, "Automatic Compiler-Inserted I/O Prefetching for Out-of-Core Applications," Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation, pp 3-17, 1996.

[16] B. K. Pasquale and G. C. Polyzos, "A Static Analysis of I/O Characteristics of Scientific Applications in a Production Workload," Proceedings of Supercomputing '93, pp 388-397, 1993.

[17] A. Dan, P. S. Yu, and J. Y. Chung, "Characterization of Database Access Pattern for Analytic Prediction of Buffer Hit Probability," VLDB Journal, 4(1):127-154, January 1995.

[18] M. F. Arlitt and C. L. Williamson, "Web Server Workload Characterization : The Search for Invariants," Proceedings of the 1996 ACM SIGMETRICS Conference, pp 126-137, 1996.

[19] P. J. Shenoy, P. Goyal, S. S. Rao, and H. M. Vin, "Symphony: An Integrated Multimedia File System," Proceedings of Multimedia Computing and Networking (MMCN) Conference, pp 124-138, 1998.

[20] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "An Implementation Study of a Detection-Based Adaptive Block Replacement Scheme," Proceedings of the 1999 USENIX Annual Technical Conference, pp 239-252, 1999.

[21] J. R. Spirn, "Distance String Models for Program Behavior," Computer, vol. 9, pp 14-20, Nov, 1976.

[22] J. R. Spirn, Program Behavior: Models and Measurements, Elsevier-North Holland, N.Y., 1977.

[23] J. Choi, "Block Reference Behavior and Detection Results by the DEAR Scheme," Technical Memo, http://ssrnet.snu.ac.kr/~choijm/DEAR/TM2.ps

최 종 무

1993년 서울대학교 해양학과 이학사. 1995년 서울대학교 컴퓨터공학과 공학석사. 1995년 ~ 현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 운영체제, 입출력 시스템, 데이타 마이닝

조 성 제

1989년 서울대학교 컴퓨터공학과 공학사. 1991년 서울대학교 컴퓨터공학과 공학석사. 1996년 서울대학교 컴퓨터공학과 공학박사. 1997년 ~ 현재 단국대학교 이과대학 전산통계학과 조교수. 관심분야는 시스템 소프트웨어, 분산 및 병렬 처리, 시스템 보안.

노 삼 혁

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

민 상 렬

1983년 서울대학교 전자계산기공학과, 공학사. 1985년 서울대학교 전자계산기공학과 공학석사. 1989년 워싱턴대학 전산학과 박사. 1989년 ~ 1990년 IBM Watson 연구소 객원 연구원. 1990년 ~ 1992년 부산대학교 컴퓨터공학과 조교수. 1992년 ~ 현재 서울대학교 컴퓨터공학과 조교수. 관심분야는 컴퓨터구조, 병렬처리 시스템, 캐쉬 메모리 시스템 등

조 유 근

1971년 서울대학교 공대 졸업. 1978년 미네소타대학교 전산학과 박사. 1979년 ~ 현재 서울대학교 컴퓨터공학과 교수. 관심분야는 알고리즘, 운영체제, 데이타 구조 등