# 객체지향 데이타베이스에서의 비용기반 버퍼 교체 알고리즘

## (A Cost-Based Buffer Replacement Algorithm in Object-Oriented Database Systems)

박 종 목 † 한 욱 신 †† 황 규 영 †††
(Chong-Mok Park) (Wook-Shin Han) (Kyu-Young Whang)

**요 약** 많은 객체지향 데이타베이스 시스템들은 객체에 대한 빠른 액세스를 제공하기 위하여 객체 버퍼를 관리한다. 기존의 고정 길이의 페이지를 단위로 하는 교체 알고리즘들은 고정 크기의 페이지의 교체 비용이 일정하므로 버퍼에서 발생하는 비용이 단순히 버퍼 폴트 횟수에 비례한다고 가정하고 있다. 그러나, 객체 버퍼에서는 객체들의 크기와 교체 비용이 객체마다 다르므로 이러한 가정은 더이상 성립하지 않는다.

본 논문에서는 객체버퍼를 위한 비용기반 교체 알고리즘을 제안한다. 제안된 알고리즘은 객체들의 크기와 교체 비용을 포함하도록 기존의 페이지 기반 모델을 확장한 비용 모델을 기반으로 단위 시간 및 단위 공간당 비용이 최소가 되도록 하는 객체를 교체한다. 성능 평가 결과에 따르면 이 알고리즘은 기존의 LRU-2에 비해 거의 항상 우수하며 경우에 따라 2배 이상의 성능을 보인다. 비용기반 알고리즘은 기존의 방법들이 적용된 어떤 응용에도 쉽게 적용될 수 있으며, 특히 교체 비용이 일정하지 않은 객체지향 데이타베이스 시스템에서 효율적으로 활용될 수 있다.

***Abstract*** Many object-oriented database systems manage object buffers to provide fast access to objects. Traditional buffer replacement algorithms based on fixed-length pages simply assume that the cost incurred by operating a buffer is proportional to the number of buffer faults. However, this assumption no longer holds in an object buffer where objects are of variable-length and the cost of replacing an object varies for each object.

In this paper, we propose a cost-based replacement algorithm for object buffers. The proposed algorithm replaces the objects that have minimum costs per unit time and unit space. The cost model extends the previous page-based one to include the replacement costs and the sizes of objects. The performance tests show that the proposed algorithm is almost always superior to the LRU-2 algorithm and in some cases is more than twice as fast. The idea of cost-based replacement can be applied to any buffer management architectures that adopt earlier algorithms. It is especially useful in object-oriented database systems where there is significant variation in replacement costs.

## 1. 서 론

전통적인 DBMS에서는 버퍼를 물리적인 단위인 페이지의 단위로 관리한다. 이러한 DBMS에서는 교체(replacement)와 인출(fetch)이 모두 페이지 단위로 수행되며, 이와 같은 버퍼 관리 방식을 *페이지기반 버퍼링(page-based buffering)*[1]이라 한다. 이에 반하여, 대부분의 OODBMS에서는 버퍼를 *페이지 버퍼(page buffer)*와 *객체 버퍼(object buffer)*로 나누어서 관리한다[1,2]. 객체 버퍼는 논리적인 단위의 객체들을 관리하

· 본 연구는 첨단정보기술 연구센터를 통하여 과학재단의 지원을 받았음.

† 정 회 원 : 삼성전자소프트웨어센터
cmpark@swc.sec.samsung.co.kr
†† 학생회원 : 한국과학기술원 전산학과
wshan@mozart.kaist.ac.kr
††† 종신회원 : 한국과학기술원 전산학과 교수
kywhang@cs.kaist.ac.kr

논문접수 : 1999년 3월 19일
심사완료 : 1999년 10월 7일

며, 페이지 버퍼는 객체들이 디스크 저장 포맷으로 저장된 물리적인 페이지들을 관리한다. 객체 버퍼에 상주하는 객체들은 페이지 버퍼로부터 복사되고 교체시에는 다시 페이지 버퍼에 저장되며, 실제적인 디스크 액세스는 모두 페이지 버퍼에서 일어난다. 이러한 버퍼 관리 방식을 *이중 버퍼링(dual buffering)*[1,2]이라고 한다.

현재까지 페이지 버퍼를 위한 많은 교체 알고리즘들이 소개된 바가 있다. 대표적인 알고리즘들로서 LRU[3], CLOCK[4], LRU-k[5] 등이 있다. *교체(replacement)*란 새로운 페이지를 위한 공간을 확보하기 위하여 버퍼로부터 제거될 페이지를 선정하고 그 페이지를 디스크로 다시 쓰는 과정을 말한다[3]. 교체 알고리즘들의 공통적인 전략은 참조가 예상되는 시점이 가장 오래된 페이지, 즉 참조될 가능성이 가장 낮은 페이지를 교체하는 것이다. LRU는 가장 마지막으로 참조된 시점을 사용하여 앞으로 참조될 시점을 예측한다[3]. LRU-k는 LRU를 일반화시킨 알고리즘으로서 가장 마지막 k개의 참조된 시점들을 사용하며[5], CLOCK은 LRU를 단순화한 알고리즘으로서 참조 비트[4]를 활용한다.

이에 반하여, 객체 버퍼를 위한 알고리즘에 관한 연구는 매우 미흡한 실정이다. 현재 OODBMS들에서는 객체 버퍼의 관리를 위해서 간단한 알고리즘에 의존하거나 페이지 교체 알고리즘들을 적용해서 사용한다. ORION[2]과 UniSQL[6]에서는 현재 사용되고 있지 않은 객체들을 모두 교체시키는 간단한 불필요 정보 수집(garbage collection) 알고리즘에 의존한다. 그러나, 불필요 정보 수집은 앞으로 사용될 가능성이 높은 객체들도 교체시킬 수 있으므로 효율적이지 못하다. Versant[7]와 GemStone[8]의 경우 기존의 페이지 교체 알고리즘인 LRU를 사용한다.

페이지 버퍼에서는 페이지가 고정 길이이고 페이지를 교체하는데 발생하는 비용이 일정한 반면, 객체 버퍼에서는 객체들의 크기가 가변적이고 객체를 교체할 때에 발생하는 비용이 객체마다 다르다. 객체 버퍼의 이러한 특징들은 기존의 페이지 교체 알고리즘들이 간과하고 있는 것들로서 버퍼 운용 성능에 큰 영향을 미친다. 본 논문에서는 교체를 수행할 때 발생하는 비용을 *교체 비용(replacement cost)*이라 정의한다. 새로운 객체를 인출하는데 드는 비용은 교체 비용에 포함시키지 않고 별도로 취급하기로 한다. 또한, CPU 계산 시간은 디스크 액세스 시간에 비해 무시할 수 있다고 가정하여 모든 비용을 디스크 액세스 횟수로 정의한다.

교체비용이 객체마다 다르게 나타나는 이유는 객체들의 가변길이 특성으로 인한 것와 인덱스 갱신이 수반되는 경우의 두 가지가 있다. 객체들의 크기가 가변적이므로, 어떤 객체가 하나 이상의 페이지에 저장되는 경우에는 그 객체를 교체할 때 여러 개의 페이지를 갱신해야 한다. 또한, 객체의 변경된 애트리뷰트에 대해서 인덱스가 정의되어 있으면 인덱스를 갱신하는 비용이 교체 비용에 포함될 수 있다. 관계형 DBMS에서는 인덱스가 정의된 애트리뷰트를 갱신할 때마다 인덱스도 함께 갱신해야 했는데, OODBMS에서는 응용 프로그램에서 객체의 애트리뷰트를 갱신할 때 객체 버퍼에 복사된 객체의 내용만 갱신되며 그 객체가 교체될 때까지는 변경된 내용이 바로 페이지 버퍼에 기록되지 않으므로 인덱스의 갱신을 객체가 교체되는 시점에서 수행할 수 있다.

본 논문은 객체 버퍼의 특징들을 고려한 비용기반 교체 알고리즘을 제안한다. 제안된 비용 모델은 기존의 페이지 교체 알고리즘들인 LRU나 LRU-k의 모델[9]을 객체의 가변길이 특성과 인덱스 갱신으로 인해 발생하는 교체비용이 포함되도록 일반화시킨다. 비용기반 교체 알고리즘은 단위시간 및 단위공간당 최소의 비용을 갖는 객체를 교체한다.

비용 모델을 이용한 버퍼 관리 방법은 객체지향 데이타베이스 이외의 분야에서도 유사한 시기에 연구된 바가 있다[10,11,12]. 그러나, 이러한 연구들은 객체지향 데이타베이스에서 고려되어야 할 교체 비용은 비용 모델에 포함시키지 않고 있다. Scheuermann, Shim, 그리고 Vingralek[11]은 데이타 웨어하우스(Data Warehouse) 시스템에서 질의 결과들을 캐쉬하기 위한 비용기반 모델을 제시하고 있다. 여기서 제안한 모델은 질의 결과의 크기와 질의의 수행 빈도, 그리고 질의의 수행시간을 고려하지만 질의 결과가 교체될 때에는 비용이 발생하지 않으므로 교체 비용은 고려하지 않는다. Sinnwell과 Weikum[12]은 분산 웍스테이션(Networks of workstations) 환경에서 분산 캐쉬(distributed caching)을 위한 비용기반 알고리즘을 제안하고 있으며, 이 알고리즘에서는 웍스테이션들에 분산되어 저장된 객체들을 인출하는데 드는 비용과 객체의 크기를 고려하지만 마찬가지로 교체 비용은 고려하지 않는다. Aggarwal, Wolf, 그리고 Yu[10]는 객체의 크기를 고려하도록 LRU를 확장한 비용기반 알고리즘을 제안하고 있지만 교체 비용은 다루지 않는다. 아울러, 본 논문에서 제시한 아이디어는 참고 문헌 [10], [11], [12]의 아이디어와는 독립적으로 개발되었음을 밝히는 바이다[13,14].

본 논문의 구성은 다음과 같다. 2 절에서는 교체 알

고리즘들의 기본 개념을 소개하고, 3 절에서는 비용기반 교체 알고리즘을 제안한다. 4 절에서는 성능평가를 수행한 결과를 제시한다. 마지막으로, 5 절에서는 결론을 맺는다.

## 2. 기존의 교체 알고리즘들

본 절에서는 교체 알고리즘을 설명하는데 필요한 용어를 정의하고 기존의 페이지 교체 알고리즘들의 기반이 되는 개념을 소개한다. 데이타베이스에 저장된 페이지들의 집합을 $N = 1, ..., n$이라 할 때, DBMS의 상위 구성요소로부터 발생하는 페이지 참조들의 순열(sequence)을 *참조 스트링(reference string)* $w$라 정의하고, 수식 (1)과 같이 표현한다[3,5].

$$w = r_1 r_2 ... r_t ..., r_t \in N \quad (1)$$

여기서, $r_t$는 $t$번째 요청을 나타내며 $r_t = p$로서 페이지 $p$를 참조함을 표현한다. $t$는 순열상의 위치를 나타내지만 이산(discrete) 시간 축상에서의 하나의 시점으로 해석할 수도 있다. 즉, 시점 $t$에서 페이지 $p$가 참조되었다고 볼 수 있다.

참조할 페이지가 버퍼에 상주하지 않을 경우 *버퍼 폴트(buffer fault)*가 일어난다고 한다[4]. 하나의 버퍼 폴트를 처리하는데 드는 비용은 폴트를 일으킨 페이지를 버퍼로 인출하는 *인출 비용(fetch cost)*과 갱신된 희생자를 디스크에 다시 기록하는 *교체 비용(replacement cost)*으로 이루어지는데, 페이지 버퍼에서는 교체 비용을 무시하며 인출 비용이 일정하다고 간주하므로 어떤 참조 스트링에 대해서 발생하는 총 비용은 버퍼 폴트의 횟수에 비례하게 된다[4].

페이지 $p$가 시점 $t$에서부터 앞으로 참조될 시점까지의 거리를 *전방거리(forward distance)*라 하고 $d_t(p)$로 표시한다[9]. 만약 시점 $t$로부터 $i(i \geq 0)$ 번째 시점에서 페이지 $p$가 처음 참조된다면(즉, $r_{t+i} = p$이면), $d_t(p) = i$ 이고, $p$가 한번도 참조가 안되면 $d_t(p) = \infty$이라고 정의한다. 예를 들면, 참조 스트링이 1 2 1 4 3 2 라고 할 때, $r_6 = 2$이므로 $d_3(2) = 6-3 = 3$이 된다.

일반적으로, 페이지 교체 알고리즘에서 버퍼 폴트의 횟수를 최소화하기 위한 최적의 전략은 참조가 예상되는 시점이 가장 오래된 페이지를 교체하는 것이다. 이러한 기준을 사용하는 최적의 알고리즘에는 페이지들에 대한 전방거리를 미리 알 수 있는지 여부에 따라서 Belady의 알고리즘[15]과 Denning, Chen, Shedler의 알고리즘(이후로 *DCS*라고 칭함)[16]으로 두 가지가 있다[3,9]. Belady의 알고리즘은 참조 스트링이 미리 알려져 있다고 가정하고 현재 버퍼에 있는 페이지 중에서 전방거리가 가장 큰 것을 교체한다. 이 알고리즘은 최적의 알고리즘으로서 어떤 참조 스트링을 입력으로 받아도 항상 최소의 비용을 발생시킨다[3]. 그러나, 일반적으로 참조 스트링은 미리 알 수 없으므로 실제적인 교체 알고리즘으로는 사용이 불가능하고 최소의 버퍼 폴트 횟수를 계산하는 기준으로만 활용되고 있다. DCS는 간단한 확률적 모델을 기반으로 하는 알고리즘으로서, 앞으로 발생할 참조 스트링을 모르는 상태에서 전방거리의 기대값(expected value)이 가장 큰 페이지를 교체한다. 즉, 이 알고리즘은 어떤 시점 $t$에서 $E[d_t(p)]$가 가장 큰 페이지 $p$를 교체한다. 참조 스트링의 형태가 독립적 참조 모델(independent reference model) [3]이란 확률적 모델을 따른다는 가정하에서 이 알고리즘에 의해 발생하는 비용의 기대값은 최소이다[3].

*독립적 참조 모델*이란 참조 스트링 $r_1 r_2 ... r_t ...$ 가 독립적인 확률변수(random variable)들의 순열이고, 이들은 모두 동일한 안정적(stationary) 이산 확률 분포(discrete probability distribution) $\{p_1, ..., p_n\}$을 갖는다는 가정이다. 다시 말해서, 각각의 참조들은 서로 독립적이어서 어떤 페이지 $p$가 참조될 확률은 시점과 무관하게 일정하다는 것이다. 즉, $Pr[r_t = p] = Pr[r_{t+1} = p] = ... = p_p$이다. 그러면, $d_t(p)$는 시점 $t$에서 페이지 $p$에 대한 전방거리를 나타내는 확률변수로서 다음과 같은 안정 상태의 기하분포(geometric distribution)를 갖는다[9].

$$Pr[d_t(p) = k] = p_p(1 - p_p)^{k-1}, \ k = 1, 2, ...$$

따라서, 시점 $t$에 상관없이 페이지 $p$에 대한 전방거리의 기대값 $E[d_t(p)]$은 항상 $1/p_p$이 된다.

LRU[4]나 LRU-K[5]는 모두 DCS의 모델을 기반으로 하며, 페이지 $p$에 대한 전방거리의 기대값을 예측하기 위해서 시점 $t$ 이전에 $p$가 참조된 시점까지의 거리를 활용한다. 페이지 $p$가 시점 $t$에서부터 그 이전에 마지막 $k$번째 참조된 시점까지의 거리를 *후방 k-거리(backward k-distance)*라 하고 $b_t(p,k)$로 표시한다[5]. 만약 시점 $t$로부터 뒤로 $j(j \geq 1)$ 번째 시점에서 $p$가 마지막 $k$ 번째로 참조되었다면(즉, $r_{t-j} = p$이고 그 이후로 $t$시점 이전까지 $k$-1번 참조되었으면), $b_t(p,k) = j$ 이고, 시점 $t$ 이전에 $p$가 $k$-1번 이내로 참조되었으면 $b_t(p,k) = \infty$이라고 정의한다. 예를 들면, 참조 스트링 1 2 1 4 3 2에서, 시점 $t$=6에서 페이지 $p$=1에 대한 마지막 두번째 참조는 $r_1$이므로 $b_6(1,2) = 6-1 = 5$가 된다. LRU-K는 LRU를 일반화한 알고리즘으로서 페이지 $p$

에 대한 후방 $k$-거리를 이용해서 전방거리의 기대치를 예측한다. 두 개의 페이지 $p_1,p_2$가 있을때 $b_t(p_1, k) > b_t(p_2, k)$이면, 이 알고리즘은 후방 $k$-거리가 큰 페이지 $p_1$을 교체한다. LRU는 $b_t(p,1)$이 가장 큰 페이지 $p$를 교체하므로 LRU-k의 특수한 경우인 LRU-1이라고 볼 수 있다.

## 3. 비용기반 객체 버퍼 교체 알고리즘

본 절에서는 비용 모델을 도입한 새로운 객체버퍼 교체 알고리즘을 제안한다. 3.1 절에서는 비용기반 교체 알고리즘의 개념을 설명하고, 3.2 절에서는 제안된 알고리즘의 최적성을 보인다. 3.3 절에서는 비용기반 알고리즘의 유효성을 나타내는 척도인 유효비율을 정의하고, 3.4 절에서는 전방거리의 기대값을 예측하기 위한 방법을 설명한다. 마지막으로, 3.5 절에서는 알고리즘을 기술한다.

### 3.1 기본 개념

본 절에서는 교체 비용과 객체의 크기를 고려하도록 DCS를 확장한 객체 버퍼 교체 알고리즘을 제안한다. 객체 $x$의 인출비용과 교체비용을 각각 비용 함수 $Cfetch(x)$와 $Crep(x)$라고 표기하고, 객체 $x$의 크기를 $size(x)$라고 표현한다. 이러한 비용함수들은 객체에 가해지는 연산의 종류, 인덱스 존재 여부, 그리고 OODBMS의 하부 저장 모델에 따라서 결정된다. 본 논문의 실험에 사용된 OODBMS를 위한 비용함수들이 부록에 정의되어 있다. 교체 비용의 경우 객체에 대한 연산이 갱신 연산이면 갱신되는 속성에 인덱스가 존재하는지에 따라서 비용이 크게 달라진다.

하나의 객체를 교체할 때 발생하는 비용을 살펴보면, 임의의 시점 $t$에서 어떤 객체 x를 교체한다면 $d_t(x)$ 시간동안 $size(x)$만큼의 공간을 확보하기 위해 $Cost(x) = Crep(x) + Cfetch(x)$만큼의 비용을 지불한다고 볼 수 있다. 객체 $x$가 교체된 시점 $t$에서 $Crep(x)$의 비용이 발생하고, $d_t(x)$ 시간 이후에 객체가 참조되어 $Cfetch(x)$의 비용이 발생하는데, $d_t(x)$ 시간동안 $size(x)$만큼의 공간이 확보되기 때문이다. DCS의 모델에서와 같이 $d_t(x)$는 미리 알 수 없으므로 예상되는 전방거리인 $\mathrm{E}[d_t(x)]$를 사용하여 객체 $x$를 교체하는데 드는 단위시간 및 단위공간당 비용을 수식 (2)과 같이 정의할 수 있다.

$$Cunit(x) = (Cost(x)/size(x)) \ / \ \mathrm{E}[d_t(x)] \qquad (2)$$

본 논문에서는 $Cunit(x)$를 객체 $x$의 *단위 비용(unit cost)*이라 정의한다.

본 논문에서는 교체 전략으로서 단위 비용이 최소인 객체를 교체하는 알고리즘을 제안한다. 이 알고리즘은 기존의 알고리즘들과는 달리 객체들의 비용을 고려하므로 *비용기반 객체버퍼 교체 알고리즘(Cost-based Object Buffer Replacement Algorithm — Cobra)*이라고 부른다. Cobra는 DCS를 일반화한 것으로 볼 수 있다. DCS는 모든 객체들에 대해서 $Crep(x) = c_1$, $Cfetch(x) = c_2$, $size(x) = c_3$이고 $c_1$, $c_2$, $c_3$가 모두 상수인 Cobra의 특수한 경우이다. 그러면, 상수 $c_4 = (c_1+c_2)/c_3$라고 할 때 Cobra는 $c_4/\mathrm{E}[d_t(x)]$가 최소가 되는 객체를 교체하므로 $\mathrm{E}[d_t(x)]$가 가장 큰 것을 교체하는 DCS와 동일하게 된다. 이에 반하여, Cobra는 $Crep(x)$, $Cfetch(x)$, $size(x)$가 모두 변수인 경우에도 비용을 최소화할 수 있다.

### 3.2 Cobra의 최적성

본 절에서는 독립적 참조 모델을 가정할 때 Cobra는 최적의 알고리즘임을 보인다. 일반적으로 하나의 버퍼 폴트가 일어날 때 교체되는 객체의 크기와 인출되는 객체의 크기는 다를 수 있다. 따라서, 크기가 큰 객체가 인출되면 하나 이상의 객체들이 교체될 수 있고, 마찬가지로, 크기가 큰 객체가 교체되면 하나 이상의 객체들이 교체가 없이 연속적으로 인출될 수 있다. 이러한 문제로 인하여 알고리즘의 비용 분석이 어려워지므로 문제를 단순화하기 위하여 본 논문에서는 각 버퍼 폴트당 하나만의 객체가 교체된다고 가정한다. 또한, 객체들의 가변길이 특성을 감안하여 최소화의 기준을 객체들을 교체하고 인출하는데 드는 단위 공간당의 기대 비용으로 정한다.

객체 버퍼의 크기를 $m$이라고 할 때 데이타베이스에 존재하는 객체들의 집합 $N = \{1, \ldots, n\}$ 중에서 $k$개의 객체가 버퍼에 상주할 수 있다고 하면, 시점 $t$에서 버퍼의 상태 $S_t(m)$을 수식 (3)과 같이 표현할 수 있다.

$$S_t(m) = \{x_i \mid x_i \in N, \ 1 \le i \le k\} \qquad (3)$$

편의상 $x_k$는 다음 번에 교체되는 객체라고 가정한다.

버퍼 상태 $S_t(m)$에서 하나의 참조가 일어날때 발생하는 단위공간당 기대비용을 $C(S_t(m))$이라 하면 이것은 수식 (4)와 같이 표현할 수 있다. 여기서 $D$는 $x_k$를 포함하여 버퍼에 상주하지 않은 객체들의 집합으로서 $D = \{x_k, \ldots, x_n\}$이고 $B = \sum_{j=k}^{n} p_{x_j}$이다.

$$C(S_t(m)) =$$
$$\sum_{i=k}^{n} \frac{Cfetch(x_i)}{size(x_i)} \Pr[r_t = x_i] + \sum_{j=k}^{n} \frac{Cfetch(x_j)}{size(x_j)} \Pr[x_j \in S_t(m) | x_j \in D] \Pr[r_t \in D]$$
$$- \sum_{l=k}^{n} \frac{Cfetch(x_l) + Crep(x_l)}{size(x_l)} \Pr[x_l \in S_t(m) | x_l \in D] \Pr[r_t = x_l] \qquad (4)$$

$$= \sum_{i=k}^{n} \frac{Cfetch(x_i)}{size(x_i)} p_x + \sum_{j=k}^{n} \frac{Cfetch(x_j)}{size(x_j)} \frac{p_{x_j}}{B} B - \sum_{l=k}^{n} \frac{Cfetch(x_l) + Crep(x_l)}{size(x_l)} \frac{p_{x_l}}{B} p_{x_l}$$

$$= \sum_{i=k}^{n} \frac{Cfetch(x_i) + Crep(x_i)}{size(x_i)} p_x - \sum_{l=k}^{n} \frac{Cfetch(x_l) + Crep(x_l)}{size(x_l)} \frac{p_{x_l}}{B} p_{x_l} \qquad (5)$$

$$= \sum_{i=k}^{n} Cunit(x_i) - \sum_{l=k}^{n} Cunit(x_l) \frac{p_{x_l}}{B} \qquad (6)$$

$$\approx \sum_{i=k}^{n} Cunit(x_i) \qquad (7)$$

수식 (4)는 $D$에 속한 객체의 참조가 일어날 경우 하나의 객체를 인출하는 단위공간당 기대 비용 (첫째 항)과 $D$에 속한 한 객체가 버퍼에 상주한 상태에서 교체될 경우 발생하는 단위공간당 기대 비용(둘째 항)을 더하고 여기에서 $D$에 속한 객체가 버퍼에 상주하고 있는 상태에서 다시 참조되는 경우를 고려하여 그 객체가 인출되고 교체되는 단위공간당 기대 비용(세째 항)을 제외한다. 첫째 항에서는 $D$에 속한 각 객체의 인출 비용에 참조 확률을 곱한 것을 객체 크기로 나누어 모두 합하고, 둘째 항에서는 각 객체의 교체 비용에 그 객체가 교체될 확률을 곱한 것을 객체 크기로 나누어 모두 합한다. $D$에 속한 어떤 객체가 교체될 확률은 그 객체가 이미 버퍼에 올라와 있을 확률 $\Pr[x_j \in S_t(m) | x_j \in D]$에 $D$에 속한 객체가 참조될($D$에 속하지 않은 객체이면 교체가 일어나지 않기 때문임) 확률 $\Pr[r_t \in D]$ = $B$의 곱으로 표현된다. $\Pr[x_j \in S_t(m) | x_j \in D]$는 $D$에 속한 객체 $x_j$가 시점 $t$ 이전에 참조되었을 조건부 확률이므로 $p_{x_j}/B$이다[1]. 세째 항은 한 객체가 교체되고 인출되는 단위공간당 비용에 D에 속한 객체가 버퍼에 상주할 확률 $\Pr[x_l \in S_t(m) | x_l \in D]$을 곱하고 그 객체가 시점 $t$에서 참조될 확률 $\Pr[r_t = x_l] = p_{x_l}$을 곱한 값들의 합이다.

수식 (4)의 첫째와 둘째 항은 하나의 항으로 묶여서 수식 (5)와 같이 전개된다. 독립적 참조 모델을 가정하면 $p_{x_i}$ = 1/E$[d_t(x_i)]$이므로, 이식을 단위비용을 이용하여 다시 수식 (6)과 같이 전개 할 수 있다. 여기서 두번째 항은 데이타베이스가 크면 무시해도 좋으므로 첫번째 항만을 살펴보기로 한다.

Cobra에서는 시간이 지남에 따라서 버퍼의 상태 $S_t(m)$은 단위 비용이 가장 높은 $k$-1개의 객체들을 포함하게 된다. 즉, $S_t(m)$에 속한 객체들은 $Cunit(x_i) \le Cunit(x_j)$, $1 \le i < k$, $k \le j \le n$의 관계를 갖게 된다. 그 이유는 Cobra는 단위 비용이 가장 낮은 객체를 교체하기 때문에 단위 비용이 가장 높은 $k$-1개의 객체들은 일단 버퍼에 상주하게 되면 교체되지 않기 때문이다. DCS에서와 유사하게, 본 논문에서는 이러한 상태를 *안정화 상태(steady state)*라고 정의한다[9]. Cobra의 안정화 상태에서는 $D$에 속한 객체들의 단위 비용이 모두 $S_t(m)$에 속한 $k$-1개의 객체들보다 작으므로 $\sum_{i=k}^{n} Cunit(x_i)$은 임의의 버퍼 상태에 대하여 최소가 된다. 따라서, Cobra의 교체 전략, 즉 단위 비용이 최소인 객체를 교체하는 전략은 안정화 상태에서 하나의 참조가 일어날때 단위공간당 기대비용인 $C(S_t(m))$을 최소화하는 전략이다.

### 3.3 유효 비율

임의의 교체 알고리즘 $A$에 대하여, 크기가 $m$인 버퍼에서 참조 스트링 $w = r_1 \ldots r_T$에 대하여 발생하는 총 비용 $C(A,m,w)$를 수식 (8)과 같이 정의한다.

$$C(A, m, w) = \sum_{t=1}^{T} \left( \sum_{x \in X_t} Cfetch(x) + \sum_{y \in Y_t} Crep(y) \right) \qquad (8)$$

수식 (8)에서 $X_t$는 시점 $t$에서 인출되는 객체들의 집합, $Y_t$는 시점 $t$에서 교체되는 객체들의 집합이다. 이 수식은 DCS의 모델[3]을 교체 비용이 포함되도록 확장하고, 교체 비용과 인출 비용을 객체에 대한 함수로 표현하며, 버퍼 크기 $m$을 바이트(byte) 단위의 값으로 정의한 것이다.

다음은 교체 알고리즘들의 유효성을 나타내기 위한 척도를 정의한다.

**정의 1. 유효비율(Effectiveness Ratio)**: 크기가 $m$인 버퍼와 참조 스트링 $w$에 대해서, 알고리즘 $A$의 알고리즘 $B$에 대한 *유효비율*을 $ER(A,B,m,w)$ = $C(B,m,w)$ / $C(A,m,w)$와 같이 정의한다.

유효비율은 두 개의 알고리즘의 비용을 역의 비율로 표현함으로써 하나의 알고리즘이 다른 알고리즘에 비해 어느 정도 우수한 지를 나타낸다. 4 절에서는 실험을 통하여 Cobra의 DCS에 대한 유효비율을 보인다.

### 3.4 전방거리의 기대값 추정 방법

전방거리의 기대값 E$[d_t(x)]$는 확률 분포가 미리 주어지는 경우 $1/p_x$가 된다. 그러나, 실제로는 확률 분포가 미리 주어지지 않으므로, 이론적인 값인 E$[d_t(x)]$를 추정하기 위한 실용적인 방법이 필요하다.

본 논문에서는 전방거리의 기대값을 추정하기 위하여 과거의 $k$번째까지의 참조 시점을 사용하는 방법을 제안한다. 이 방법은 참조시점들간의 간격들에 대한 평균으로써 기대값을 유추하는 통계적 방법에 근거한 휴리스

---

1) 이 계산은 한번에 교체되는 객체는 한개이라는 가정하에 성립된다. 즉, D에 속한 객체가 버퍼에 있을 경우 자기 자신이 참조되는 경우를 제외하고는(이 경우는 세째 항에서 고려됨) 반드시 교체되어야 하고 버퍼에 남아있을 수가 없다고 가정한다.

틱이다. 즉, 시점 $t$에서 객체 $x$에 대해 과거의 $k$번째 참조시점까지 마지막으로 발생된 $k$-1($k$ > 1)개의 참조시점들간의 간격들을 $i_1 \ldots i_{k-1}$라고 하면, 이들의 평균 $I$는

$$I = \sum_{j=1}^{k-1} i_j/(k-1) = (b_t(x,k) - b_t(x,1))/(k-1)$$

이다. 단, $k$ = 1인 경우에는 객체가 한번만 참조되어 참조시점들간의 간격이 존재하지 않으므로 $I$ = ∞라고 가정한다. 그림 1은 객체 $x$가 참조되는 시점들을 시간축상에서 보여주고 있다.

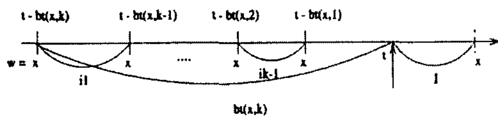


그림 1 전방거리의 기대값을 예측하기 위한 Cobra-k의 방법

여기서, 평균 $I$는 현재 시점 $t$에서 다음 참조시점까지의 예상거리로 활용된다. 이와 같이 마지막 $k$ 번째 참조 시점을 활용하여 전방거리의 기대값을 예측하는 알고리즘을 본 논문에서는 *Cobra-k*라고 정의한다. LRU-k는 Cobra-k와 마찬가지로 마지막 $k$번째까지의 참조 시점을 활용하지만 현재 시점으로부터 마지막 참조 시점까지의 거리인 $b_t(x,k)$를 사용하여 예상되는 거리 $I$를 예측한다. 그러나, 이 방법은 시점 $t$가 변함에 따라서 $I$값이 달라지기 때문에 비용기반 교체 알고리즘에 적용할 경우 매 시점마다 모든 객체들의 비용을 다시 계산해야 한다는 문제가 생긴다. 참고 문헌 [12]에서는 LRU-k의 이러한 문제를 부분적으로 해결하기 위해 일정 주기마다 객체들의 비용을 다시 계산하는 방법을 제안하고 있다. 그러나, 이러한 주기를 결정하기가 어려울 뿐아니라 비용을 다시 계산하는데 드는 오버헤드도 크므로 본 논문에서는 Cobra-k의 방법을 실용적인 해결책으로 제안한다. 또한, 실제로 실험을 한 결과 Cobra-2와 LRU-2의 방법의 결과들은 1% 이내의 차이를 보인다.

Cobra-k나 LRU-k 모두 $k$값을 결정하는 문제가 있다. 직관적으로는 $k$ = ∞인 경우가 최적일 것으로 예상되지만, 실제로는 객체에 대한 참조 간격들에 많은 차이가 있기 때문에 그렇지 않게 나타난다. LRU-k에서는 $k$ > 2인 경우보다 $k$ = 2인 경우가 일반적으로 좋은 결과를 보이므로, 참고 문헌 [5]에서는 LRU-2를 일반적인 알고리즘으로 제안하고 있다. 본 논문에서도 마찬가지로 $k$ = 2로 하는 Cobra-2를 일반적인 알고리즘으로 제안한다. Cobra-k의 경우에도, 실제 실험에 의하면 $k$ > 2인 경우보다 $k$ = 2인 경우가 더 좋은 결과를 보인다.

### 3.5 알고리즘

Cobra-2는 기본적으로 LRU-2 알고리즘과 유사하다. LRU-2 알고리즘과의 가장 큰 차이점은 객체를 교체하는 기준으로서 단위 비용을 사용한다는 점과 마지막 두 참조 시점들간의 간격을 이용하여 객체의 전방거리를 예측한다는 것이다.

Cobra-2에서 사용하는 데이타 구조는 LRU-2를 단순화한 2Q 알고리즘[17]에서 제시한 구조들을 기반으로 한다. *Cobra-2* 알고리즘에서는 다음과 같은 정보를 메모리에 유지한다:

- HIST($x$,1)과 HIST($x$,2) : 객체 $x$가 마지막으로 참조된 두 개의 시점들을 저장한다. 즉, HIST($x$,1)에는 객체 $x$가 마지막으로 참조된 시점을 저장하며, HIST($x$,2)에는 마지막 직전에 참조된 시점을 저장한다. 객체 $x$가 한번만 참조된 경우에는 HIST($x$,2)의 값은 -∞를 나타낸다.
- COST($x$) : 객체 $x$에 의해 발생하는 비용을 저장한다. 이 비용은 수식 (2)에서 설명된 바와 같이 $Crep(x)$ + $Cfetch(x)$ 로 계산된다.

이러한 정보들은 객체와 별도로 유지하며, 객체가 교체되어도 일정 시간동안 메모리에 남아 있어 객체가 다시 참조될 때 과거의 참조 기록으로 활용할 수 있도록 한다. 이러한 정보를 *보유정보(Retained Information)* [5]라고 하는데, 본 논문에서는 객체 $x$에 대한 보유정보를 RI($x$)라고 표기한다.

보유정보들은 A1, Am, Aout의 세 가지의 데이타 구조에서 관리된다. A1은 마지막 하나의 참조 시점에 대한 정보만 있는 객체들의 보유정보들을 유지하는 First-In-First-Out(FIFO) 큐(queue)이다. Am은 두 개의 참조 시점에 대한 정보가 있는 객체들의 보유정보를 유지하며, 객체의 단위 비용인 (COST($x$)/$size(x)$)/E[$d_t(x)$])값을 키로 하는 최소히프(min-heap)로서 단위 비용이 최소인 객체를 삭제할 수 있도록 한다. 여기서 E[$d_t(x)$] 값은 마지막 두 개의 참조 시점들간의 간격인 HIST($x$,1) - HIST($x$,2)로써 추정한다. Aout은 교체된 객체들에 대한 보유정보를 관리하는 FIFO 큐이다. 이러한 데이타 구조들은 2Q 알고리즘에서 사용하는 A1in, Am, 그리고 A1out 큐들과 유사하다. 2Q와 다른 점은 Cobra-2의 Am이 단위 비용을 키로 하는 최소히프라는 점과 2Q의 A1out이 A1in에서 교체된 객체들의 보유정보만을 관리하는 것과는 달리 Cobra-2에서는 Aout이

A1과 Am 모두에서 교체된 객체들의 보유정보를 관리한다는 점이다.

Cobra-2알고리즘이 동작하는 과정이 각각 알고리즘 1과 2에 기술되어 있다. 알고리즘 1에서는 시점 $t$에서 객체 $x$가 참조될 때 수행되는 과정을 보인다. 알고리즘 2는 객체 $x$를 수용하기 위한 버퍼공간을 확보하고 객체를 인출하는 과정을 나타낸다.

알고리즘 1에서, 2 행부터 4 행까지는 객체에 관한 보유정보를 계산하는 과정이다. 먼저, 객체의 보유정보를 최근 시점에 맞추어서 수정한다(2 행과 3 행). 다음, 객체의 비용을 계산한다(4 행).

6 행부터 16 행까지는 객체의 보유정보가 어떤 데이타 구조에서 관리되는지 검사하는 과정을 나타낸다. RI($x$)가 A1이나 Am에 있으면 객체 $x$가 버퍼에 상주하면서 2 회 이상 참조되는 경우이므로 RI($x$)를 A1이나 Am에서 제거하고 Am에 다시 삽입한다(7 행과 8 행).

RI($x$)가 A1과 Am에 없으면 객체가 버퍼에 상주하지 않은 경우이다. RI($x$)가 Aout에 있으면 객체가 과거에 참조된 적이 있으므로 RI($x$)를 Aout에서 제거하고 Am에 삽입한 후 객체 $x$를 인출한다(10 행부터 12 행). 반면, RI($x$)가 어디에서도 유지되지 않고 있으면, 객체가 처음 참조된 것으로 간주해서 RI($x$)를 A1 큐에 삽입하고 객체 $x$를 인출한다(14 행과 15 행).

다음으로 알고리즘 2에서, 객체를 위한 여분의 연속적인 공간이 존재하는지 확인하고(2 행), 연속적인 공간이 있으면 객체를 인출한다(3 행). 공간이 부족하고 A1이 비어있지 않으면, A1에 있는 객체들을 FIFO 순서로 교체한다(5 행부터 10 행까지). 여기서 교체된 객체의 RI($x$)는 Aout에 삽입한다.

여전히 공간이 부족할 경우에는 Am에 있는 객체들을 하나씩 차례로 연속적인 공간이 확보될 때까지 교체한다(12 행부터 17 행까지). 이때 단위 비용이 최소인 객체가 삭제 된다(14 행). 마찬가지로 교체된 객체의 RI($x$)는 Aout에 삽입한다(16 행).

Aout의 크기가 최대 크기인 $Kout$을 넘어서면 끝에서부터 차례로 보유정보를 제거해준다(19 행부터 22 행까지). 본 알고리즘에서는 Aout 큐에서 유지되는 동안 참조가 일어나지 않은 객체는 오랫동안 참조가 안된 것으로 간주하여 보유정보를 제거함으로써 추후에 참조가 일어나도 처음 참조되는 객체들과 동일하게 취급한다. 마지막으로, 객체를 확보된 공간으로 인출한다 (24 행).

**Algorithm 1: reference(x,t)**

```
1)  begin
2)      HIST(x,2) := HIST(x,1)
3)      HIST(x,1) := t
4)      COST(x) := Cfetch(x) + Crep(x)
5)
6)      if (RI(x) is in A1 or Am) then
7)          remove RI(x) from A1 or Am
8)          insert RI(x) to Am
9)      else if (RI(x) is in Aout) then
10)         remove RI(x) from Aout
11)         insert RI(x) to Am
12)         reclaim_for(x)
13)     else
14)         add RI(x) into the head of A1
15)         reclaim_for(x)
16)     endif
17) end
```

**Algorithm 2: reclaim_for(x)**

```
1)  begin
2)      if (there is contiguous free space for x) then
3)          fetch x into the free space
4)      else
5)          while (there is no free space for x and A1 is
            not empty) do
6)          begin
7)              remove the tail RI(y) of A1
8)              replace y
9)              add RI(y) into the head of Aout
10)         end
11)
12)         while (there is no free space for x and Am is
            not empty) do
13)         begin
14)             remove RI(y) from Am where
                  (COST(y)/size(y))/(HIST(y,1)-HIST(y,2))
                  is minimum
15)             replace y
16)             add RI(y) into the head of Aout
17)         end
18)
19)         while (|Aout| > Kout)
20)         begin
21)             remove the tail RI(z) of Aout
22)         end
23)
24)         fetch x into the reclaimed space
25)     end if
26) end
```

본 알고리즘의 CPU 오버헤드는 하나의 참조 스트링

을 처리하기 위한 전체 시간에 비하면 비교적 작다고 볼 수 있다. 실제 구현에서는 객체 버퍼 관리자는 객체를 위한 두 가지 연산(''Fix''와 ''Unfix'')들을 지원할 수 있다. Fix 연산은 객체를 버퍼 메모리 내에 고정시켜서 객체를 버퍼상에서 직접 참조할 수 있도록 하며, Unfix 연산은 객체가 자유롭게 이동하거나 교체될 수 있도록 한다. 본 논문에서는 독립적 참조 모델을 가정하므로 하나의 객체의 Fix/Unfix 기간 내에서 그 객체에 대해서 일어나는 일련의 연관된 참조들을 하나의 독립적인 참조로 간주할 수 있다. 그러면 하나의 객체에 대한 한번의 참조는 최대 두번의 min-heap 연산(한번의 삭제와 한번의 재삽입)을 수반하며, 각각의 연산은 버퍼에 상주하는 객체의 수를 $n$이라고 하면 $O(\log n)$ 만큼 걸린다. 또한, 전방거리의 기대값을 계산하기 위해 한번의 감산과 객체의 단위공간당의 비용을 계산하기 위해 한번의 제산(division)이 일어난다[2]. 따라서, 본 알고리즘의 CPU 오버헤드는 하나의 참조가 일어나는 전체 시간 (Fix/Unfix 기간)에 비하면 무시할 수 있다.

## 4. 성능평가

본 절에서는 Cobra, DCS, Cobra-2, 그리고 LRU-2의 성능을 비교한다. Cobra와 DCS는 객체들의 정적인 참조 확률을 이용하는 반면, Cobra-2와 LRU-2는 동적으로 변하는 객체들에 대한 마지막 두 개의 참조 시점들을 이용하여 객체들의 전방거리의 기대값을 추정한다. Cobra와 Cobra-2는 모두 부록에 정의되어 있는 비용함수들을 이용하여 객체들의 단위 비용을 계산한다. 객체들의 참조 확률은 참조 스트링에서 각 객체가 참조된 횟수를 참조 스트링의 전체 길이로 나눈 값으로 얻는다. 모든 알고리즘들은 한국과학기술원에서 개발한 오디세우스(ODYSSEUS)[18]에서 구현하고, LRU-2는 Cobra-2의 특수한 경우로서 인출 비용($c_1$), 교체 비용($c_2$), 그리고 객체의 크기($c_3$)가 모두 상수로 설정된 경우로 취급한다. 본 절에서는 또한, 실험 결과를 바탕으로 Cobra-2와 Cobra의 LRU-2와 DCS에 대한 유효비율을 살펴본다.

### 4.1 성능평가 환경과 변수

성능 평가에 사용된 데이타베이스는 현재 OODBMS의 성능 평가 모델로 널리 활용되고 있는 OO7 벤치마크[19]의 데이타 모델이다. OO7 모델은 계층구조로 이루어진 어셈블리(Assembly)들과 여기에 연결된 복합부품(Composite Part)들을 다루는 CAD/CAM이나 CASE 등의 응용들을 대표하는 모델이다.

성능 평가는 크게 읽기 전용 운행(read-only traversal)(T1), 단순 갱신 운행(simple-update traversal)(T2), 그리고 인덱스 갱신 운행(index-update traversal)(T3)의 세가지 운행 연산에 대해서 수행한다. T1은 갱신이 수반되지 않아서 객체들의 교체 비용이 0인 상황, T2는 인덱스가 구축되지 않은 애트리뷰트의 단순한 갱신으로 인하여 객체들간의 교체 비용이 0이 아닌 상황, 그리고 T3는 인덱스가 구축된 애트리뷰트의 갱신으로 인하여 교체 비용의 차이가 객체마다 크게 다르게 나타나는 상황을 모델링한다.

버퍼 교체 알고리즘의 성능은 데이타베이스 크기에 대한 버퍼 크기의 비율에 의해 좌우되므로, 모든 실험은 객체 버퍼의 크기를 최소 2000KB부터 데이타베이스의 크기까지 증가시키면서 수행하고 여기서 발생한 페이지 엑세스 횟수를 측정한다. 본 논문에서는 페이지 버퍼가 단순히 디스크 엑세스를 위해서만 존재한다고 가정하므로, 페이지 버퍼는 최소한의 크기인 1개의 페이지로 설정한다. 따라서, 결과로 측정된 페이지 엑세스 횟수는 디스크 엑세스 횟수와 동일하다. 데이타베이스 크기는 OO7 벤치마크의 ''small3'' 데이타베이스[19]로 선정한다. 보유정보가 부족한데서 오는 영향을 줄이기 위하여 객체들의 보유정보의 크기(알고리즘 2에서 *Kout* 변수)는 무한대로 설정한다.

### 4.2 성능 평가 결과

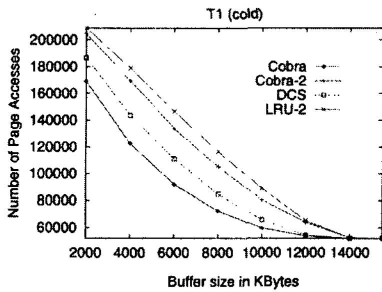
각각의 운행들은 ''cold''와 ''warm''으로 구분해서 수행한다. cold 운행은 버퍼가 빈 상태에서 객체들의 집합을 운행하는 것으로서 초기에 객체들을 버퍼에 적재하는 비용을 포함한다. warm 운행은 cold 운행을 마친 시점에서 같은 집합의 객체들을 다시 운행하는 것으로서 안정화 상태에서 발생하는 비용을 보이기 위한 것이다. 이 실험에서는 cold 운행을 마친 상태에서 warm 운행을 두번 수행한 후 두번째 warm 운행의 결과를 보고한다. 이렇게 함으로써 두번째 warm 운행에서는 모든 객체들에 대해서 최소한 두번의 참조 시점에 대한 기록을 확보할 수 있도록 하여 버퍼의 상태를 안정화 상태에 가깝게 만들 수 있다.

먼저 T1 운행을 살펴본다. 그림 2는 T1의 cold와 warm 운행 결과들을 보인다.

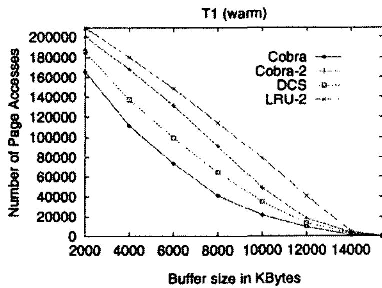
T1에서는 객체들의 교체 비용이 0이므로 Cobra와 DCS의 차이는 객체들의 크기가 다름으로 인해서 생긴

---

2) 객체의 단위공간당의 비용은 객체가 디스크로부터 버퍼로 처음 인출되는 시점에서 미리 계산할 수 있으며, 각 애트리뷰트가 처음으로 갱신되는 시점마다 다시 계산할 수 있다. 이러한 계산 비용은 디스크에서 객체를 인출하거나 교체하는 비용에 비하면 무시될 수 있다.




(a) cold




(b) warm

그림 2 T1의 cold와 warm 운행

다. Cobra는 단위 비용에서 객체의 크기를 함께 고려하는 반면, DCS는 참조 확률만을 고려하므로 Cobra가 더 우수한 결과를 보인다. 또한, Cobra-2와 LRU-2는 각각 대응되는 알고리즘들인 Cobra와 DCS에 유사한 경향으로 근접함을 알 수 있다.[3)]

cold 운행(그림 2(a))과 warm 운행(그림 2(b))을 비교하면, cold 운행보다 warm 운행의 결과들이 더 낮은 비용을 보인다. 이것은 cold운행에서는 객체들의 초기 인출 비용들이 포함되기 때문이다. 버퍼 크기가 데이타베이스 크기에 근접하면서 cold 운행에서는 모든 알고리즘들의 비용이 일정한 상수값(52000 근처)에 수렴하는 반면, warm 운행에서는 0에 수렴한다. cold 운행에서 수렴하는 상수값은 모든 객체들을 버퍼에 한번 인출하는 비용과 동일하다.

다음은 T2 운행의 결과를 살펴본다. 그림 3은 T2의 warm 운행 결과를 보인다.

T2의 경우 교체비용 $Crep(x)$가 0이 아니므로 모든 알고리즘들의 결과가 T1 연산에서보다 더 큰 비용을 보인다. 또한, 여기서는 Cobra와 DCS, 그리고 Cobra-2와 LRU-2의 차이가 T1에서보다 더 크게 나타난다. 그 이유는 Cobra와 Cobra-2에서는 객체들의 교체비용을 고려하는 반면 DCS나 LRU-2는 이를 무시하기 때문이다.

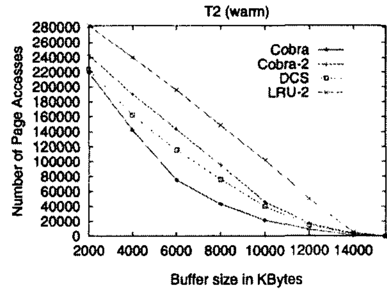
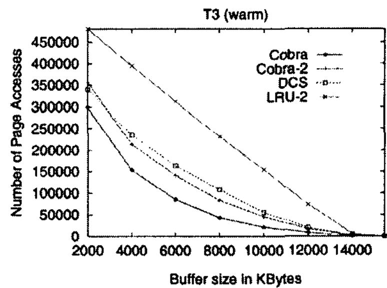
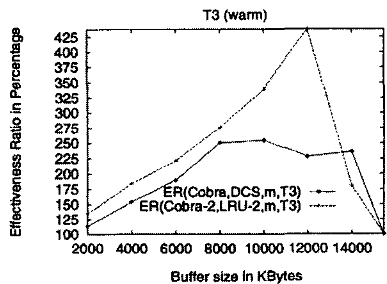



그림 3 T2의 warm 운행

마지막으로, 그림 4(a)는 T3의 운행 결과를 보인다. T3에서는 교체비용에 높은 인덱스 갱신 비용이 포함되기 때문에 Cobra와 DCS, 그리고 Cobra-2와 LRU-2의 차이가 T2에서보다 현저하게 나타난다. 그림 4(b)는 이들 알고리즘간의 차이를 유효비율로써 표시하고 있다. 버퍼 크기 4000KB에서 8000KB[3)]까지 Cobra의 DCS에 대한 유효비율이 약 150%에서 250%까지, Cobra-2의




(a) 페이지 엑세스 횟수




(b) Cobra와 Cobra-2의 유효비율

그림 4 T3의 warm 운행

3) 800KB 이후로는 유효비율이 매우 높게 나타나더라도 알고리즘들의 절대적인 비용이 작아져서 버퍼 크기가 데이타베이스 크기에 근접함에 따라 유효비율의 의미도 작아진다.

LRU-2에 대한 유효비율은 약 175%에서 275%까지로 나타난다. 이것은 인덱스 갱신이 수반되는 경우 Cobra-2가 LRU-2에 비하여 2배 이상의 좋은 성능을 보일 수 있음을 입증한다.

## 5. 결론 및 향후 연구

본 논문에서는 비용 모델을 기반으로 하는 객체버퍼를 위한 교체 알고리즘인 Cobra를 제안하였다. 비용기반 모델은 기존의 LRU와 LRU-k등의 기반이 되는 Denning, Chen, Shedler의 모델[16]을 확장하여 객체의 인출 비용과 교체 비용을 포함시킨다. 이러한 모델을 바탕으로 Cobra는 버퍼에 상주한 객체들 중에서 단위 비용이 최소인 객체를 교체한다.

성능 평가에서 Cobra-2는 LRU-2 보다 항상 우수한 결과를 보였다. 읽기 전용 연산에서는 Cobra-2가 객체의 크기를 고려함으로써 LRU-2에 비해 우수하며, 갱신 연산에서는 객체의 교체 비용을 고려하기 때문에 보다 우수한 결과를 보였다. 특히, 인덱스 갱신 연산에서는 LRU-2에 비해 2 배이상의 성능을 보였다.

Cobra는 많은 분야에서 적용될 수 있는 포괄적인 알고리즘이다. 한가지 예로서 다양한 인출 및 기록 시간을 갖는 이질적인 저장 매체들을 사용하는 대용량 저장 시스템을 위한 버퍼 관리가 있을 수 있다. 이러한 시스템에서는 자기 디스크, CD-ROM, 광자기 디스크 등 엑세스 시간이 매체에 따라서 차이가 나므로 효율적인 버퍼 관리가 필요하다. Cobra는 앞으로 새롭게 등장하는 다양한 응용에 널리 활용될 것으로 기대된다.

## 참 고 문 헌


[1] Kemper, A. and Kossmann, D., "*Dual-Buffering Strategies in Object Bases*," In *Proc. Int'l Conf. on Very Large Data Bases*, Santiago, Chile, pp. 427-438, Sept. 1994.

[2] Kim, W., *Introduction to Object-Oriented Databases*, MIT Press, 1990.

[3] Coffman, E. G. Jr. and Denning, P. J., *Operating Systems Theory*, Prentice-Hall, 1973.

[4] Effelsberg, W. and Haerder, T., "Principles of Database Buffer Management," *ACM Trans. on Database Systems*, Vol. 9, No. 4, pp.560-595, Dec. 1984.

[5] O'Neil, E. J., O'Neil, P. E., and Weikum, G., "The LRU-K Page Replacement Algorithm For Database Disk Buffering," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., May 1993.

[6] UniSQL, Inc., UniSQL/X Application Program Interface Reference Guide, UniSQL, 1995.

[7] Versant Object Technology Corp., VERSANT C++ Reference Manual, 1993.

[8] Butterworth, P., Otis, A., and Stein, J., "The Gemstone Object Database Management System," *Comm. of the ACM*, Vol. 34, No. 10, pp. 64-77, Oct. 1991.

[9] Aho, A. V., Denning, P. J., and Ullman, J. D., "Principles of Optimal Page Replacement," *Journal of the ACM*, Vol. 18, No. 1, pp. 80-93, Jan. 1971.

[10] Aggarwal, C., Wolf, J. L., and Yu, P. S., "Caching on the World Wide Web," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 11, No. 1, Jan./Feb. 1999.

[11] Scheuermann, P., Shim, J., and Vingralek, R., "WATCHMAN: A Data Warehouse Intelligent Cache Manager," In *Proc. Int'l Conf. on Very Large Data Bases*, Mumbai, India, 1996.

[12] Sinnwell, M. and Weikum, G., "A Cost-Model-Based Online Method for Distributed Caching," In *Proc. Int'l Conf. on Data Engineering*, IEEE, Birmingham, U.K., Apr. 1997.

[13] Park, C.-M., Persistent Object and Object Buffer Management in Object-Oriented Database Systems, Ph. D. Thesis, Department of Computer Science, Korea Advanced Institute of Science and Technology, 1997.

[14] Whang, K.-Y. and Park, C.-M., "A Cost-based Object Buffer Replacement Algorithm for Object-Oriented Database Systems," Korean Patent, File No. 96-66427, 1996.

[15] Belady, L. A., "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal*, Vol. 5, No. 2, pp. 78-101, 1966.

[16] Denning, P. J., Chen, Y. C., and Shedler, G. S., A Model for Program Behavior under Demand Paging, IBM Research Report RC-2301, IBM T. J. Watson Research Center, Yorktown Heights, N. Y., Sept. 1968.

[17] Johnson, T. and Shasha, D., "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," In *Proc. Int'l Conf. on Very Large Data Bases*, pp. 439-450, Santiago, Chile, Sept. 1994.

[18] Park, C.-M., Shim, J.-G., Lee, J.-H., Woo, J.-H., Cho, W.-S., and Whang, K.-Y., "ODYSSEUS: A Multi-User Object-Oriented Database System for UNIX," In *Proc. Spring Biannual Conf. of Korea Information Science Society*, Vol. 21, No. 2, pp. 31-34, Apr. 1994.

[19] Carey, M. J., DeWitt, D. J., and Naughton, J. F., "The OO7 Benchmark," In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington,

D.C., May 1993.

[20] Whang, K.-Y., "Constructing Cost Formulas for Relational Database Query Optimizers: A Tutorial," In *Proc. IEEE TENCON '87*, IEEE, Seoul, Korea, Aug. 1987.

## 부록: 비용 함수들

이 부록에서는 실험에 사용된 오디세우스(ODYSSEUS) OODBMS[18]의 비용 모델을 기술한다. 이 비용 모델은 참고문헌[20]에 기술된 저장 모델을 기초로 하고 있으며, 다른 객체지향 데이타베이스의 저장구조에 맞추어 확장 또는 변경될 수 있다. 사용된 표기법은 아래와 같다.

$C$ : 클래스
$A$ : 애트리뷰트
$n_C$ : 클래스 C에 속한 객체의 수
$p$ : 페이지 크기
$L_A$ : 애트리뷰트 A를 위한 인덱스 페이지의 블록킹 인수
$F_A$ : 애트리뷰트 A의 선택률

기초적인 비용 함수들은 하나의 객체를 읽거나 쓰는 비용, 그리고 하나의 애트리뷰트에 대한 인덱스를 수정하는 비용 함수들로 구성된다. 모든 비용 함수들의 단위는 디스크 페이지 엑세스 횟수로 정의한다.

**함수** $OR(x)$: 객체 $x$를 디스크에서 객체 버퍼로 읽어오는 비용

$$OR(x) = \lceil size(x)/p \rceil$$

객체를 읽는 비용은 그 객체가 차지하는 디스크 페이지 수와 같다. 따라서, $p$가 페이지 크기라고 하면 객체를 읽는 비용은 $\lceil size(x)/p \rceil$이다.

**함수** $OW(x)$: 객체 버퍼에 있는 객체 $x$를 디스크에 쓰는 비용

$$OW(x) = 2 \cdot (\lceil size(x)/p \rceil)$$

객체를 쓰는 비용도 객체를 읽는 비용과 마찬가지로 계산할 수 있다. 단, 객체가 저장된 각각의 페이지를 읽고 갱신한 다음 다시 디스크에 써야 하므로 두배만큼이 든다.

**함수** $IU(A,C)$: 클래스 C의 애트리뷰트 A에 구축된 인덱스에 하나의 엔트리를
삽입하거나 삭제하는 비용

$$IU(A, C) = \lceil \log_{L_A} n_C \rceil + (\lceil 0.5 \cdot F_A \cdot n_C / L_A \rceil - 1) + 1$$

인덱스를 갱신할 경우, 삽입할 엔트리나 삭제할 엔트리가 속한 리프 페이지를 루트로부터 찾은 다음, 같은 키 값을 갖는 엔트리들 중에서 삽입되거나 삭제될 해당 엔트리를 찾는다[20]. 첫째 항목은 루트로부터 리프까지 인덱스를 탐색하는 비용이다. 둘째 항목은 리프 페이지에서 해당 엔트리를 찾는 비용이며 평균적으로 같은 키 값을 갖는 엔트리 수의 반만큼 찾는다. 여기서 첫번째 리프 페이지를 엑세스하는 비용은 첫째 항목에 포함되어 있으므로 하나의 페이지 엑세스를 감한다. 마지막 항목은 갱신된 페이지를 디스크에 쓰는 비용이다.

이와 같은 기초적인 비용 함수들을 사용하여 객체의 인출 비용과 교체 비용은 각각 다음과 같이 유도된다:

**함수** $Cfetch(x)$: 객체 $x$를 인출하는 비용

$$Cfetch(x) = OR(x)$$

**함수** $Crep(x)$: 객체 $x$를 교체하는 비용

A. 객체의 애트리뷰트가 변경되지 않은 경우

$$Crep(x) = 0$$

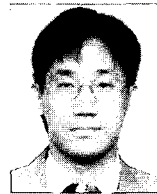
B. 인덱스가 구축되지 않은 애트리뷰트만 변경된 경우

$$Crep(x) = OW(x)$$

C. 인덱스가 구축된 애트리뷰트 $A_1,...,A_k$가 변경된 경우

$$Crep(x) = OW(x) + \sum_{i=1}^{k} IU(A_i, C) \cdot 2$$

객체를 교체하는데 드는 비용은 객체의 상태에 따라서 달라진다. 객체의 어떤 애트리뷰트도 변경되지 않았으면 교체 비용은 발생하지 않는다. 객체의 애트리뷰트 중에서 인덱스가 구축된 것이 변경되지 않았으면 객체 자신만 갱신되고 인덱스 갱신은 일어나지 않는다. 반면, 인덱스가 구축된 애트리뷰트가 변경되었으면 객체 자신도 갱신될 뿐아니라 갱신된 애트리뷰트에 해당하는 인덱스마다 이전 키값의 삭제와 새로운 키값의 삽입이 일어난다. 따라서, $IU(A_i, C) \bullet 2$의 비용이 든다.



박 종 목

1990년 2월 연세대학교 전산과학과 학사. 1992년 2월 한국과학기술원 전산학과 석사. 1997년 2월 한국과학기술원 전산학과 박사. 1997년 3월 ~ 1997년 7월 한국과학기술원 인공지능 연구센타 박사후 연구원. 1997년 8월 ~ 1998년 8월 IBM Almaden Research Center (Post-doc researcher). 1998년 9월 ~ 1999년 3월 한국과학기술원 인공지능 연구센타 박사후 연구원. 1999년 4월 ~ 현재 삼성전자 중앙연구소 소프트웨어 센타 선임 연구원. 관심분야는 객체지향 데이타베이스, 객체 관계형 데이타베이스, 멀티미디어

한 욱 신

1994년 2월 경북대학교 컴퓨터공학과 학사. 1996년 2월 한국과학기술원 전산학과 석사. 1996년 3월 ~ 현재 한국과학기술원 전산학과 박사과정. 관심분야는 객체지향 DBMS, 객체 관계형 DBMS, 객체 캐쉬 관리

**황 규 영**

1973년 서울대학교 전자공학과 졸업(B.S.). 1975년 한국과학기술원 전기 및 전자학과 졸업(M.S.). 1982년 Stanford University (M.S.). 1983년 Stanford University (Ph.D.). 1975년 ~ 1978년 국방과학연구소(ADD), 선임연구원. 1983년 ~ 1990년 IBM T.J. Watson Research Center, Research Staff Member. 1992년 ~ 1994년 한국정보과학회 데이타베이스 연구회(SIGDB) 운영위원장. 1995년 한국정보과학회 이사 겸 논문지 편집위원장. 1999년 ~ 현재 한국정보과학회 부회장. Editior: The VLDB Journal, 1990 ~ 현재. Editor: Distributed and Parallel Databases: An International Journal, 1991 ~ 1995. Editor: International Journal of Geographical Information Systems, 1994 ~ 현재. Associate Editor: IEEE Data Engineering Bulletin, 1990 ~ 1993. 1998년 ~ 2004년: Trustee, The VLDB Endowment. 1990년 ~ 현재 한국과학기술원 전산학과 교수. 1999년 ~ 현재 첨단정보기술연구센터(과학재단 우수연구센터) 소장. 관심분야는 데이타베이스 시스템, 멀티미디어, GIS