분산 이질형 객체 환경에서 캐싱 알고리즘의 설계 및 성능 분석

(Design and Performance Analysis of Caching Algorithms for Distributed Non-uniform Objects)

반 효 경 † 노 삼 혁 †† 민 상 렬 ††† 고 건 ††††

(Hyokyung Bahn) (Sam H. Noh) (Sang Lyul Min) (Kern Koh)

요 약 캐싱 기법은 저장 장치 계층 간의 속도차를 완충시키기 위해 캐쉬 메모리, 페이징 기법, 버퍼링 기법 등으로 널리 연구되어 왔다. 하지만, 최근 웹을 비롯한 다양한 광역 분산 환경의 보편화에 따라 단일 시스템 내의 저장 장치 간에 이루어지는 캐싱 기법 뿐 아니라 타 노드의 객체를 캐싱하는 기법의 중요성이 커지고 있다. 광역 분산 환경에서의 캐싱 기법은 객체의 캐싱에 드는 비용과 캐싱으로 인한 이득이 객체의 근원지 노드의 위치에 따라 이질적이기 때문에 비용 차이를 고려한 캐쉬 교체 알고리즘이 필요하다. 한편, 캐쉬 교체 알고리즘은 온라인 알고리즘으로서 매 시점 교체 대상이 되는 객체를 즉시 선택해야 하기 때문에 알고리즘의 시간 복잡도가 지나치게 높지 않아야 한다. 그러나, 광역 분산 환경에서의 교체 알고리즘에 대한 지금까지의 연구는 객체들의 이질성을 고려하는 문제와 캐쉬 운영의 시간 복잡도 측면 모두에서 만족스러운 결과를 보이지는 못하고 있다. 본 논문은 이러한 점을 극복하여 우수한 성능을 나타내면서 효율적인 구현이 가능한 새로운 교체 알고리즘을 설계하고, 그 우수성을 트레이스 기반 모의실험을 통해 보여 준다.

Abstract Caching mechanisms have been studied extensively to buffer the speed gap of hierarchical storages in the context of cache memory, paging system, and buffer management system. As the wide-area distributed environments such as the WWW extend broadly, caching of remote objects becomes more and more important. In the wide-area distributed environments, the cost and the benefit of caching an object is not uniform due to the location of the object; which should be considered in the cache replacement algorithms. For online operation, the time complexity of the replacement algorithm should not be excessive. To date, most replacement algorithms for the wide-area distributed environments do not meet both the non-uniformity of objects and the time complexity constraint. This paper proposes a replacement algorithm which considers the non-uniformity of objects properly; it also allows for an efficient implementation. Trace-driven simulations show that proposed algorithm outperforms existing replacement algorithms.

1. 서 론

캐싱 기법은 저장 장치 계층 간의 속도 차이를 완충시켜주기 위해 컴퓨터 구조, 운영 체제, 데이타베이스 등의 분야에서 각각 캐쉬 메모리, 페이징 기법, 버퍼링 기법 등으로 널리 연구되어 왔다. 하지만, 최근 웹(WWW)을 비롯한 다양한 광역 분산 환경의 보편화에 따라 단일 시스템 내에서 속도차가 있는 저장 장치 간에 이루어지는 캐싱 기법 뿐 아니라 타 노드의 객체를 캐싱하는 기법의 중요성이 커지고 있다. 특히, 웹의 인기가 높아감에 따라 네트웍의 병목 현상과 그에 의한 사용자 지연 시간 증가 등이 점점 더 심각해져가는 추세이며, 이를 완화시키기 위해 최근 다양한 각도에서의 웹 캐싱 기

· 이 논문은 2000년도 두뇌한국21사업에 의하여 지원되었음
† 학생회원 : 서울대학교 전산과학과
hyokyung@summer.snu.ac.kr
†† 종신회원 : 홍익대학교 정보컴퓨터공학부 교수
noh@cs.hongik.ac.kr
††† 종신회원 : 서울대학교 컴퓨터공학부 교수
symin@dandelion.snu.ac.kr
†††† 종신회원 : 서울대학교 전산과학과 교수
kernkoh@june.snu.ac.kr
논문접수 : 1999년 5월 10일
심사완료 : 2000년 3월 13일

법에 관한 연구가 활발히 이루어지고 있다 [1, 2, 4, 6, 9, 13].

캐싱 기법의 성능은 캐쉬 교체 알고리즘에 의해 크게 좌우된다. 캐쉬 교체 알고리즘은 미래의 참조를 미리 알지 못하는 상태에서 한정된 캐쉬 공간에 보관하고 있을 객체와 삭제할 객체를 동적으로 결정하는 온라인 알고리즘이다. 이러한 알고리즘은 가상 메모리의 페이지 교체 알고리즘과 블럭 캐싱 기법 등에서 광범위하게 연구되어 왔다 [3, 5, 11, 12]. 하지만, 광역 분산 환경에서의 캐싱은 근원지 노드의 위치에 따라 객체의 인출 비용이 다르기 때문에 전통적인 캐쉬 교체 알고리즘으로는 우수한 성능을 기대하기 어렵다. 즉, 전통적인 캐싱 기법에서는 일반적으로 단위 객체를 느린 저장 장치에서 빠른 저장 장치(캐쉬)로 읽어오는 비용이 동일하다. CPU 캐쉬의 경우 캐쉬 부재(cache miss)시 객체를 주 메모리(main memory)에서 캐쉬 메모리로 읽어오게 되고, 블럭 캐싱이나 페이징의 경우에는 디스크에서 주 메모리로 읽어오기 때문에 그 비용이 균일하다. 반면, 광역 분산 환경에서는 객체를 가지고 있는 근원지 노드의 위치 및 특성에 따라 객체를 읽어오는 비용이 다르기 때문에 비용 차이를 고려한 캐쉬 교체 알고리즘이 필요하다. 또한, 전통적인 캐싱 기법에서는 캐싱 단위가 블럭, 페이지 등 동일한 사이즈였으나 웹 캐싱에서는 웹 문서를 이루는 파일 단위로 캐싱이 되기 때문에 캐싱 단위의 사이즈가 다르다. 전통적인 캐싱 기법과 이러한 차이점이 있는 환경을 본 논문에서는 이질형 캐싱 환경(non-uniform caching environment)이라 부르겠다. 그림 1은 전통적인 캐싱 기법이 다루는 동질형 캐싱 환경(*uniform caching environment*)과 웹 캐싱과 같은 이질형 캐싱 환경의 차이점을 보여주고 있다.

이와 같은 이질형 캐싱 환경에서는 캐쉬 교체 알고리즘의 우수성 역시 동질형 캐싱 환경과는 다른 의미를 가진다. 동질형 캐싱 환경에서는 가까운 미래에 참조될 가능성이 가장 적은 객체를 캐쉬에서 삭제하여 캐쉬 부재의 회수를 줄이는 것이 캐쉬 교체 알고리즘의 목표인 반면, 이질형 캐싱 환경에서는 객체마다 캐쉬 부재 시의 비용이 다르기 때문에 캐쉬 부재로 인해 발생하는 비용의 합을 줄이는 것이 더 중요한 의미를 지닌다.

한편, 효율적인 캐쉬 교체 알고리즘이 되기 위해서는 시간과 공간 복잡도 측면에서 현실성이 있어야 한다. 캐쉬 내에 있는 객체의 수가 n개라고 할 때 시간 복잡도 측면에서는 캐쉬 운영에 드는 시간이 $O(\log_2 n)$을 넘지 않는 것이 바람직하며, 공간 복잡도 측면에서는 캐쉬 내의 객체 하나당 부가적으로 저장되는 정보가 상수 - 즉, $O(1)$ - 를 넘지 않는 것이 바람직한 것으로 알려져 있다 [2, 12]. 동질형 캐싱 환경에서의 교체 알고리즘들은 이러한 조건을 쉽게 만족하지만, 이를 이질형 캐싱 환경에 맞게 변형할 경우 복잡도가 높아져 캐쉬 운영을 위한 온라인 알고리즘으로 사용하기 어려운 경우가 많다 [7, 9, 13]. 대부분의 기존 연구들은 이질형 캐싱 환경의 특성을 잘 반영하는 문제와 캐쉬 운영의 복잡도 측면 모두를 만족시키지는 못하고 있다. 본 논문은 먼저 이질형 캐싱 환경에서 지금까지 연구되어 온 방법들을 분석한다. 그리고, 기존 방법들의 한계를 극복하여 이질형 환경의 특성을 잘 반영하면서 효율적인 캐쉬 운영의 조건을 충족시키는 캐쉬 교체 알고리즘을 제안하고 그 우수성을 트레이스 기반 모의 실험을 통해 보여 준다.

본 논문의 2장에서는 이질형 캐싱 환경에 대한 정형적인 정의와 기존 연구에 대해 살펴 본다. 3장에서는 본 논문이 제안하는 캐쉬 교체 알고리즘을 설계하고, 4장에서는 설계된 캐쉬 교체 알고리즘의 효율적인 구현 방법을 제시한다. 5장에서는 기존의 연구들과 트레이스 기반 모의 실험을 통해 성능을 비교한다. 끝으로 6장에서는 본 논문의 결론을 제시한다.

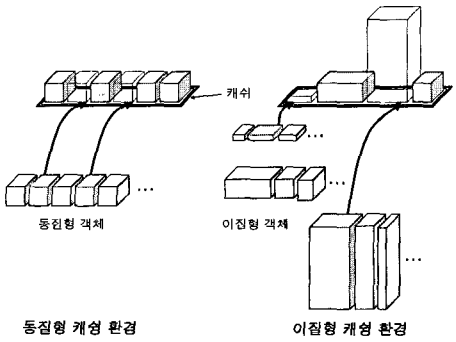

그림 1 동질형 캐싱 환경과 이질형 캐싱 환경

2. 관련 연구

2.1 이질형 캐싱 모델의 정의

본 절에서는 먼저 이질형 캐싱 환경에 대한 논리적인 모델을 정의하고, 정의된 모델에서 캐쉬 교체 알고리즘 및 알고리즘의 우수성을 나타내는 성능 척도를 정의한다. 하나의 노드에 크기가 S인 캐쉬가 있고, 노드가 필요로 하는 객체의 참조 순서(reference sequence)는 미리 알려져 있지 않으며 현재 요청되는 객체만이 알려진다고 가정한다. 요청된 객체는 캐쉬에 보관되며, 아직

요청되지 않은 객체를 미리 캐쉬에 보관하는 것(prefetching)은 허용하지 않는다. 객체 i의 사이즈가 s_i, 객체 i를 다른 노드에서 캐쉬로 인출해 오는 비용을 c_i라 하고, 객체 i의 총 요청 회수를 r_i, 이 중 캐쉬에 이미 존재하여 적중된 회수를 h_i라 하자. 그러면 캐쉬 교체 알고리즘은 매번 캐쉬 내의 객체들의 사이즈의 합이 캐쉬의 크기 S를 초과하지 않으면서 현재 요청된 객체를 서비스하는 문제가 된다. 이 때, 이 알고리즘의 성능 척도는 참조 요청을 모두 처리한 시점에서 요청된 객체들에 대한 인출 비용의 합이 최소가 되도록 하는 것이다. 이 문제는 Knapsack[1] 문제 [8]와 유사하지만, 일반적인 Knapsack 문제가 정적인 상태에서 한정된 공간에 사이즈와 가치가 다른 객체들을 담는 문제인 반면 이질형 캐싱 환경에서의 교체 알고리즘은 이러한 일을 동적으로 수행하는 온라인 알고리즘이라는 차이점이 있다. 성능 척도를 동질형 캐싱 환경에서의 성능 척도인 캐쉬 적중률(hit rate)과 유사한 방법으로 정의하면 식 (1)과 같다. 만약, 객체들의 사이즈 s_i와 인출 비용 c_i가 모두 균일한 경우라면, 본 절에서 정의한 내용은 동질형 캐싱 환경이 되고, 식 (1)은 캐쉬 적중률을 의미하게 된다.

$$\text{비용절감률} = \sum(c_i \cdot h_i) \;/\; \sum(c_i \cdot r_i) \qquad (1)$$

한편, 동질형 캐싱 환경에서는 객체의 미래 참조 순서를 미리 다 아는 경우 가장 먼 미래에 참조될 객체를 캐쉬에서 삭제하는 단순한 방법이 오프라인 최적 알고리즘임이 알려져 있다 [3]. 그러나, 이질형 캐싱 환경에서는 객체의 미래 참조 순서를 미리 다 안다 해도 최적의 알고리즘을 찾는 방법이 NP-hard 문제로 알려져 있다 [2].

2.2 객체들의 가치 평가 방법

캐쉬 교체 알고리즘은 미래의 참조를 알지 못 하는 상태에서 한정된 캐쉬 공간에 보관할 객체를 동적으로 결정하는 온라인 알고리즘으로서 그 효율성은 궁극적으로 미래의 참조 성향을 얼마나 잘 예측하는가에 좌우된다. 또한, 이질형 캐싱 환경에서는 미래 참조 성향에 대한 예측 외에도 객체들의 사이즈와 인출 비용을 함께 고려해서 캐쉬 내의 객체들에 대해 합리적인 평가를 할 수 있어야 한다. 본 절에서는 이질형 캐싱 환경에서 우수한 교체 알고리즘을 설계하기 위해 객체들의 가치 평가시 고려해야 할 요소들을 설명하겠다.

2.2.1 과거 참조 기록

캐쉬 내의 객체를 평가하는 가장 일반적인 방법이 과거 참조 기록에 의한 방법이다. 이는 전통적인 캐싱 기법에서 널리 연구되어 온 LRU, LFU 등과 같이 캐싱된 객체의 최근 참조 성향과 참조 빈도 등에 근거하여 미래의 참조 성향을 예측하는 방법을 말한다. 이는 최근에 참조된 객체가 다시 참조될 가능성이 높다는 점과 참조 회수가 많은 객체일수록 또 다시 참조될 가능성이 높다는 두 가지 사실에 입각한 것이다. 전자를 시간 지역성(temporal locality)이라 하고, 후자를 객체의 인기도(popularity)라 하여 이 두 가지 개념은 컴퓨터 프로그램의 참조 반응(program reference behavior)을 모델링하는데 널리 사용하는 방법이다 [10]. 하지만, 이와 같은 두 가지 요소는 작업 부하 및 캐싱 환경에 따라 특정한 성향이 더 강하게 나타내기 때문에 캐쉬 교체 알고리즘도 이러한 사실을 반영할 수 있어야 한다.

2.2.2 이질성에 대한 고려

동질형 캐싱 환경에서는 사용자가 요청한 객체가 캐쉬에 존재하는 경우 그 효용이 모두 동일하므로 캐쉬 교체 알고리즘은 참조 가능성이 높은 객체를 캐쉬에 보관하는 것으로 충분하다. 하지만, 이질형 캐싱 환경에서는 참조 가능성 이외에 객체의 사이즈와 인출 비용을 고려한 합리적인 가치 평가를 해야 한다. 즉, 객체를 캐쉬에 보관하고 있을지를 결정하는 가치 평가 기준은 객체의 참조 가능성에 의한 가치[2]와 캐쉬에서 적중될 경우 실제로 절약할 수 있는 인출 비용을 동시에 고려해야 한다는 점이다. 한편 캐쉬 용량은 정해져 있으므로 가능한 캐쉬 공간을 적게 차지하면서 절약하는 비용이 큰 객체일수록 가치 평가를 높게 하는 것이 바람직하다.

2.2.3 통계적 정보

객체의 미래 참조 성향을 예측하는 가장 일반적인 방법이 2.2.1절에서 설명한 과거 참조 기록에 의한 방법이지만, 캐쉬가 사용되는 환경 및 작업 부하의 특성을 분석하여 이를 캐쉬 교체 알고리즘에 활용하면 좀 더 정확한 미래 참조 예측이 가능하다. 즉, 이는 해당 캐싱 환경의 참조 패턴에 대한 특성을 통계적으로 분석하여 그 결과를 바탕으로 미래 참조를 예측하는 방법을 말한다. 블럭 캐싱 기법의 경우 순환 참조를 고려하는 것이 그 대표적인 사례이다. 웹 캐쉬의 경우 웹 문서의 사이즈, 유형, 근원지 서버 등 사용할 수 있는 메타 정보가 많기 때문에 이러한 통계적 정보에 의한 참조 성향 예측이 대단히 유용할 수 있다.

1) Knapsack 문제는 한정된 공간에 가치와 크기가 다른 객체들을 담아 그 가치의 합을 가장 크게 하는 문제로서 NP-hard 문제로 알려져 있다.

2) 참조 가능성에 대한 가치는 일반적으로 2.2.1에서 소개한 과거 참조 기록을 바탕으로 평가하게 된다.

2.3 기존의 교체 알고리즘

본 절에서는 지금까지 연구된 이질형 캐쉥 환경에서의 대표적인 캐쉬 교체 알고리즘들을 설명하도록 하겠다. 각 방법들에서 캐쉬 공간이 필요한 경우 어떤 객체를 삭제하는가를 설명하였으며, 수식으로 주어진 경우에는 *Value*의 계산값이 가장 작은 객체를 삭제하게 된다. 각 알고리즘에서 s_i와 c_i는 각각 객체 i의 사이즈와 인출 비용을 나타낸다.

- **LRU** (Least Recently Used): 최근 참조된 시각이 가장 오래된 객체를 삭제한다.
- **LFU** (Least Frequently Used): 참조 회수가 가장 적은 객체를 삭제한다.
- **SIZE** [4]: 사이즈가 가장 큰 객체를 삭제한다.
- **LLF** (Lowest-Latency-First) [6] : 인출 비용(인출 지연 시간)이 가장 작은 객체를 삭제한다.
- **LRU-MIN** [1]: 필요한 캐쉬 공간을 s라 할 때, 사이즈가 s 이상인 객체 중에서 최근 참조된 시간이 가장 오래된 객체를 삭제한다. 만약 그런 객체가 없으면 $s \leftarrow s/2$로 하여 필요한 공간이 마련될 때까지 이 부분을 반복 시행한다.
- **LRV** (Lowest Relative Value) [13]:

$$Value(i) = \begin{cases} P_1(s_i) \cdot (1-D(t)) \cdot c_i/s_i & \text{if } n=1 \\ P_n \cdot (1-D(t)) \cdot c_i/s_i & \text{otherwise} \end{cases}$$

n은 객체 i의 참조 회수, t는 직전 참조된 후부터의 시간을 나타낸다. P_n은 참조 빈도에 근거해서 미래 참조를 예측하는 부분으로서 n번 참조되었다는 전제 하에 $n+1$번 참조될 확률을 나타낸다. $P_n(s_i)$는 여기에 객체의 사이즈를 같이 고려해서 구한 확률을 나타낸다. $D(t)$는 참조 시간 간격(interaccess time)의 분포 함수로서, $1-D(t)$는 가장 최근에 참조된 후부터의 시간이 t일 때 재참조될 확률을 나타낸다. LRV는 웹 프락시 캐쉬의 교체 알고리즘으로서 P_n과 $D(t)$를 충분히 큰 프락시 캐쉬 로그를 분석하고 그에 근거해서 구하고 있다.

- **HYBRID** [6]:

$$Value(i) = (t_s + p/b_s)(n_i)^q/s_i$$

t_s는 근원지 서버 s에 연결(connect)하는데 걸리는 지연 시간을 나타내며, b_s는 근원지 서버 s에 대한 대역폭(bandwidth)을 나타낸다. n_i는 객체 i가 캐쉬에 들어온 후의 총 참조 회수를 나타내고, p와 q는 상수이다.

- **GD-SIZE** (Greedy-Dual Size) [2]: 처음 캐쉬에 들어온 객체의 *Value*는 $Value(i) = c_i/s_i$이다. 캐쉬 공간이 필요한 경우 *Value*가 가장 작은 객체를 삭제하고, 캐쉬 내의 모든 객체의 *Value*를 방금 전에 삭제한 객체의 *Value*만큼 낮춰 준다. 캐쉬 내에서 재참조된 객체의 *Value*는 $Value(i) = c_i/s_i$로 복구시킨다.
- **LNC** (Least Normalized Cost-Replacement) [7, 9]:

$$Value(i) = (k/(t_c - t_k)) \cdot c_i/s_i$$

t_c는 현재 시각, t_k는 과거 k번 이전에 참조되었던 때의 시각을 뜻한다.

LRU, LFU, SIZE, LLF 등은 이질형 캐쉥 환경의 특성을 반영하지 않거나, 한 가지 특성만을 편중되게 반영한 알고리즘들로서 대부분의 연구에서 기본적인 비교 대상으로 사용하는 알고리즘들이다 [1, 2, 6, 7, 13]. LRU-MIN은 사이즈가 작은 객체에 좀 더 높은 가치를 부여하여 삭제되는 객체의 수를 줄이기 위해 LRU를 변형한 형태이지만, 객체의 사이즈 및 인출 비용에 따른 정규화된 가치 평가를 하지는 못한다는 단점이 있다. HYBRID는 사용자 지연 시간(latency)을 줄이기 위한 웹 프락시 캐쉬의 교체 알고리즘으로서 캐쉬 운영을 $O(\log_2 n)$의 시간 복잡도에 할 수 있다는 장점이 있지만, 부가적으로 저장하고 있어야 할 정보의 양이 많고, 최근 참조 성향을 고려하지 않는다는 단점이 있다. LRV는 웹 프락시 캐쉬의 교체 알고리즘으로서 광범위한 프락시 트레이스의 분석을 통해 교체 알고리즘을 설계하기 때문에 작업 부하의 특성이 바뀔 경우 추가적인 트레이스 분석 과정과 교체 알고리즘의 설계가 필요하다는 단점이 있다. 또한, 이 방법은 객체의 인출 비용이 사이즈에 비례하지 않을 경우 캐쉬 운영의 시간 복잡도가 $O(n)$에 이르러 캐쉬 운영을 위한 온라인 알고리즘으로 사용하기에는 현실적으로 어렵다는 문제점이 있다. GD-SIZE는 LRU를 이질형 캐쉥 환경에 맞게 변형한 알고리즘으로서 캐쉬 운영을 $O(\log_2 n)$에 할 수 있다는 장점이 있지만, 과거 참조 기록을 이용하는데 있어 참조 빈도를 고려하지 않는다는 단점이 있다. LNC는 LRU-k [5]를 이질형 캐쉥 환경에 맞게 변형한 알고리즘으로서 과거 참조 기록의 활용에 최근 k번 참조되는 동안의 시간을 이용하기 때문에 참조 빈도를 고려할 수 있다는 장점이 있지만 캐쉬 운영의 시간 복잡도가 $O(n)$에 이르러 효율적인 구현이 어렵다는 단점이 있다. [9]에서는 이러한 문제점을 해결하기 위해 주기적으로 캐쉥된 객체들의 *Value*를 재계산하는 근사 방법을 쓰고 있지만, 이러한 방법의 시간 복잡도 역시 $O(\log_2 n)$보다 훨씬 높기 때문에 효율적인 구현이 어렵다. 또한, 이 방법은 과거 k번째 참조되었던 때의 시각을 이용하기 때문에

이보다 최근에 참조된 k-1번의 시점을 반영하지 못한다는 단점이 있다.

한편, 이러한 알고리즘들에 있어 캐쉬 교체가 필요할 때마다 모든 객체들의 *Value*를 다 조사하여 그 중 *Value*가 가장 작은 객체를 찾는 방법은 오버헤드가 너무 크기 때문에 일반적으로 *Value*의 크기에 따라 리스트나 힙 구조를 유지하여 캐쉬 운영의 시간 복잡도를 줄이게 된다. LRU 방법의 경우에는 참조 시간 리스트를 구성하여 새롭게 참조된 객체를 리스트의 가장 뒷부분에 연결하는 방법을 사용하면 캐쉬 운영을 $O(1)$에 할 수 있다. LRU를 제외한 대부분의 알고리즘들은 새롭게 참조된 객체라 하더라도 캐쉬 내에서 가장 큰 *Value*를 갖지는 않기 때문에 *Value*의 크기에 따른 힙을 구성하여 삽입, 삭제 및 재참조 연산을 $O(\log_2 n)$에 수행할 수 있게 한다. 한편, 힙에 기반한 캐쉬 운영을 하기 위해서는 캐쉬에 새로 들어오거나 재참조되는 객체를 제외한 모든 캐쉬 내의 객체들 간에는 *Value*의 대소 관계가 변하지 않아야 한다. 동질형 캐슁 환경의 교체 알고리즘에서는 이러한 대소 관계가 대부분의 경우 유지되지만, 이를 이질형 캐슁 환경에 맞게 수정한 알고리즘에서는 대소 관계가 유지되지 않아 힙에 기반한 캐쉬 운영이 어려운 경우가 많다. LRV와 LNC가 그 대표적인 경우이며, LRU-MIN도 시간 복잡도가 $O(n)$에 이르러 효율적인 구현이 어렵다. 표 1은 본 절에서 설명한 알고리즘들을 여러 가지 측면에서 비교한 것이다. 표에서 보는 것과 같이 기존의 알고리즘들은 나름대로 장점과 단점을 다 가지고 있다는 것을 알 수 있다.

표 1 각 알고리즘의 비교

알고리즘	이용하는 과거 참조 정보		객체의 이질성에 대한 고려	시간 복잡도
	최근 참조 성향	참조 빈도		
LRU	직전 참조 시각	고려 안함	고려 안함	$O(1)$
LFU	고려 안함	참조 회수	고려 안함	$O(\log_2 n)$
SIZE	고려 안함	고려 안함	사이즈에 편중된 고려	$O(\log_2 n)$
LLF	고려 안함	고려 안함	인출 비용에 편중된 고려	$O(\log_2 n)$
LRU-MIN	직전 참조 시각	고려 안함	사이즈에 편중된 고려	$O(n)$
LRV	직전 참조 시각	참조 회수	사이즈 및 인출 비용 고려	$O(n)$
HYBRID	고려 안함	참조 회수	사이즈 및 인출 지연 시간 고려	$O(\log_2 n)$
GD-SIZE	직전 참조 시각	고려 안함	사이즈 및 인출 비용 고려	$O(\log_2 n)$
LNC	과거 k번째 참조 시각	k번째 참조 시각에 기초	사이즈 및 인출 비용 고려	$O(n)$

3. 설 계

본 논문이 제안하는 캐쉬 교체 알고리즘은 캐슁으로 인한 비용 절감 효과를 최대화시키는 것을 목표로 하고 있다. 이를 위해서는 현 시점에서 참조 가능성이 높고 비용 절감 효과도 큰 객체를 캐쉬에 보관하도록 하는 합리적인 객체 평가 방법이 있어야 한다. 본 논문에서는 참조 가능성에 대한 가치를 과거 참조 기록으로 평가하며, 여기에 객체의 비용 차이에 의한 가중치를 곱하여 객체의 정규화된 가치 평가를 하게 된다. 본 논문이 제안하는 캐쉬 교체 알고리즘을 LUV(Least Unified Value) 알고리즘이라고 부르겠다. LUV는 캐쉬 내의 객체들의 가치를 식 (2)와 같은 방법으로 평가한다.

$$Value(i) = H(i) \cdot Weight(i) \quad (2)$$

$H(i)$는 과거 참조 기록에 의한 가치로서 이는 객체들이 참조 가능성을 평가하는 부분이다. $Weight(i)$는 객체들의 단위 크기당 인출 비용을 나타내는 가중치로서 그 정의는 식 (3)과 같다.

$$Weight(i) = c_i/s_i \quad (3)$$

c_i는 객체 i의 인출 비용을 나타내며 s_i는 객체 i의 사이즈를 나타낸다. 이와 같은 방법은 캐쉬 내의 객체들에 대해 정규화(normalization)된 가치를 부여하므로 이질형 객체에 대한 합리적인 평가가 가능하다.

과거 참조 기록으로 미래의 참조를 예측하는 방법은 지금까지 많은 연구가 있어 왔다. 블럭 캐슁 기법에서는 캐쉬 용량이 매우 작은 경우 최근 참조 성향만을 고려하는 것이 효율적이고, 캐쉬 용량이 클 경우에는 참조 빈도를 함께 고려하는 것이 더 효율적인 것으로 알려져 있다 [12]. 웹 캐슁 등 이질적인 특성의 작업 부하및 돌발적인(bursty) 작업 부하가 존재하는 분산 환경에서는 최근 참조 성향보다 참조 빈도를 고려하는 방법이 더 우수한 성능을 나타내는 것으로 알려져 있다 [5, 9]. 하지만, [2]에서는 최근 참조 성향이 웹 객체의 재참조 성향에 주도적인 영향을 미치는 것으로 분석하고 있다. 이는 각각의 연구에서 사용한 트레이스의 특성이 다르기 때문인 것으로 볼 수 있다. 본 논문에서 참조 가능성을 예측하는 함수인 $H(i)$는 이러한 캐슁 환경 및 작업 부하의 특성에 맞게 참조 빈도와 최근 참조 성향을 적절히 조절할 수 있다는 장점을 가진다. 이는 동질형 캐슁 환경의 알고리즘인 LRFU [12] 방법에 입각한 것으로서, 과거 시점에서의 각각의 참조가 객체의 가치를 높이

는 데에 기여하는 바를 각 시점의 최근성에 근거하여 계산하고 이를 더하는 방법이다. 예를 들어 객체 i가 지금까지 세 번 참조되었고 각 참조 시점부터 현재까지 흐른 시간이 각각 δ_1, δ_2, δ_3라 하자. 그러면, 현재 시점에서의 객체 i의 H 값은 세 번의 참조에 의한 기여도를 각각 더해서 식 (4)와 같이 계산하게 된다.

$$H(i) = F(\delta_1) + F(\delta_2) + F(\delta_3) \quad (4)$$

식 (4)에서 함수 $F(x)$는 x 시간 이전의 참조에 의한 가치 기여도를 나타내는 함수로서 이 함수에 의해 참조 빈도와 최근 참조 성향의 영향력을 조절하게 된다. 한편, $F(x)$는 최근의 참조가 객체의 가치를 높이는데 더 큰 역할을 할 수 있도록 하기 위해 일반적으로 감소 함수로 정의하게 된다. 현재 시각이 t_c이고, 객체 i가 캐쉬에 들어온 후 n번 참조되었고, 각각의 참조 시각이 t_1, t_2, …, t_n이라고 하면 $H(i)$의 정의는 식 (5)와 같다.

$$H(i) = \sum_{k=1}^{n} F(t_c - t_k) \quad (5)$$

각각의 참조에 대한 가치를 평가하는 함수인 $F(x)$는 [12]에서 사용한 함수인 $F(x)=(1/2)^{\lambda x}$ $(0 \le \lambda \le 1)$를 그대로 사용하였다.

4. 구 현

본 장에서는 LUV 교체 알고리즘에 대한 효율적인 구현 방법을 설명한다. 3장에서 설계한 교체 알고리즘에 의하면 각 객체들에 대한 *Value*의 계산 과정에 과거 참조되었던 모든 시각이 사용된다. 하지만, 모든 과거 참조 시각을 다 유지할 경우 하나의 객체당 저장해야 하는 메타 정보가 지나치게 많아 공간 복잡도 측면에서 LUV 정책은 현실성이 없게 된다. 또한, 객체의 *Value*는 시간이 지남에 따라 값이 변하게 되므로 매번 캐쉬 교체가 필요할 때마다 모든 객체에 대한 *Value*를 재계산해야 하는 과정이 필요할 수 있다. 이러한 방법은 알고리즘의 시간 복잡도를 $O(n)$에 이르게 하여 온라인으로 캐쉬 운영을 하는 것을 현실적으로 어렵게 한다. 하지만, 다음 두 가지 성질은 이러한 공간 복잡도와 시간 복잡도를 해결하여 효율적인 캐쉬 운영을 가능하게 한다.

성질 1. $t'>t$이고 $t'-t=\delta$인 두 시각 t와 t'에서의 객체 i의 *Value*가 각각 $Value_t(i)$, $Value_{t'}(i)$이고, t와 t' 사이에 객체 i가 참조된 적이 없으면 다음 식을 만족한다.

$$Value_{t'}(i) = Value_t(i) \cdot F(\delta)$$

(증명) 객체 i가 시각 t 이전에 n번 참조되었다고 하자. 이 때, 각각의 참조 시점과 시각 t 사이의 시간 간격을 δ_1, δ_2,…, δ_n이라 하면, 다음과 같은 방법에 의해 위의 식이 성립한다.

$$\begin{aligned}
&Value_{t'}(i)\\
&= Weight_{t'}(i)\ H_{t'}(i)\\
&= Weight_{t'}(i)\{F(\delta_1+\delta) + F(\delta_2+\delta) + \cdots + F(\delta_n+\delta)\}\\
&= Weight_{t'}(i)\{(1/2)^{\lambda(\delta 1+\delta)} + (1/2)^{\lambda(\delta 2+\delta)} + \cdots + (1/2)^{\lambda(\delta n+\delta)}\}\\
&= Weight_{t'}(i)\{(1/2)^{\lambda\delta 1} + (1/2)^{\lambda\delta 2} + \cdots + (1/2)^{\lambda\delta n}\}(1/2)^{\lambda\delta}\\
&= Weight_{t'}(i)\{F(\delta_1)+F(\delta_2)+\cdots+F(\delta_n)\}F(\delta)\\
&= Weight_t(i)\ H_t(i)\ F(\delta) = Value_t(i)\ F(\delta)
\end{aligned}$$ □

성질 1은 각각의 객체에 대한 과거 참조 시각을 모두 유지하지 않고 가장 최근 계산된 *Value*와 그 때의 시각 t만 저장하고 있으면 객체의 특정 시각에서의 *Value*를 계산할 수 있다는 점을 암시한다. 이는 앞에 언급했던 공간 복잡도의 문제를 각 객체당 $O(1)$의 공간으로 해결할 수 있음을 뜻한다. 성질 1에서는 객체가 다시 참조되지 않은 경우에 대한 *Value*의 계산 방법만을 다루고 있지만, 객체가 시각 t'에 다시 참조된 경우에는 $H_{t'}(i)$의 계산에 $F(0)$을 더해주는 방법으로 식 (6)과 같이 계산할 수 있다. 이 때, 객체의 *Value*를 계산하기 위해서는 *Weight*를 더 저장해주면 된다.

$$\begin{aligned}
&Value_{t'}(i)\\
&= Weight_{t'}(i)\ H_{t'}(i)\\
&= Weight_{t'}(i)\{F(\delta_1+\delta) + F(\delta_2+\delta) + \cdots + F(\delta_n+\delta)+F(0)\}\\
&= Weight_t(i)\{H_t(i)\ F(\delta) + 1\}\\
&= Value_t(i)\ F(\delta) + Weight_t(i) \quad (6)
\end{aligned}$$

성질 2. $Value_t(a) > Value_t(b)$이고 시각 t 이후 객체 a, b 모두 참조되지 않았으면, $t'>t$인 임의의 시각 t'에 대해 $Value_{t'}(a) > Value_{t'}(b)$를 만족한다.

(증명) $t'-t=\delta$라 하면,

$$\begin{aligned}
Value_{t'}(a) &= Value_t(a)\ F(\delta)\\
&> Value_t(b)\ F(\delta) = Value_{t'}(b)\\
&(\because F(\delta) > 0)
\end{aligned}$$ □

성질 2는 캐쉬 내에 있는 객체 중에서 참조되지 않은 객체들 간에는 *Value*의 대소 관계가 그대로 유지된다는 의미로서, 힙(heap) 구조를 이용하여 시간 복잡도 $O(\log_2 n)$에 캐쉬 운영을 할 수 있는 바탕이 된다. LRFU에서도 이와 유사한 성질이 성립하지만, 그 경우 과거 참조 기록만으로 객체의 가치를 평가하는 경우이고, 본 논문에서는 각각의 객체에 임의의 *Weight*를 주었을 경우에도 이러한 성질이 성립한다는 것을 증명한 것이다. 그림 2는 이러한 두 가지 성질을 바탕으로 각

객체들을 *Value*의 순서에 따라 최소 힙(min. heap) 구조로 유지하는 모양을 나타낸 것이다. 성질 2에 의해 재참조되는 경우를 제외하고는 *Value*의 대소 관계가 변하지 않으므로 힙 구조는 그대로 유지된다. 캐쉬 내의 객체가 재참조되거나 삽입, 삭제 연산 등이 이루어지는 경우 하나의 노드만이 힙 구조의 대소 관계에 어긋난 값을 가지게 되므로 그 노드의 위치만 찾아주면 된다. 이는 성질 1을 이용하여 위치를 찾는 경로 상에 있는 노드들만 현재 시점에서의 *Value*를 재계산하면 되므로 이러한 연산은 모두 $O(\log_2 n)$에 수행할 수 있다.

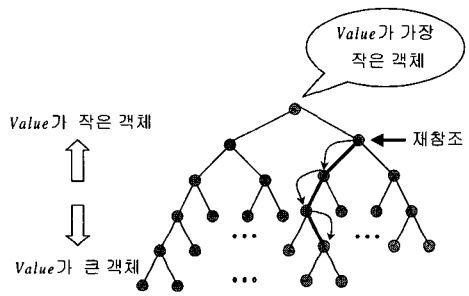

그림 2 힙 구조에 기반한 캐쉬 운영

5. 성능 분석

본 장에서는 트레이스 기반 모의 실험을 통해 LUV 방법과 기존의 캐쉬 교체 알고리즘인 LRU, SIZE, HYBRID, LRV, GD-SIZE, LNC 등의 성능을 비교하였다. LRV의 경우에는 본 실험에서 사용된 트레이스를 미리 분석하여 P_n과 $D(t)$ 등의 식이 트레이스의 특성을 정확히 반영할 수 있도록 최적화시켜 주었으며, LUV 정책에서도 우수한 성능을 나타내는 λ값으로 조절해서 실험을 하였다.

이외의 정책들에서도 그들이 필요로 하는 파라미터들과 비용 정의 방법을 각 정책의 특성에 맞게 조절하였다. 한편, LNC에서는 캐쉬에서 삭제되는 객체들에 대해서도 과거 k번 참조되는 시간을 모두 다 유지하도록 되어 있다. 그러나, 이러한 방법은 알고리즘의 공간 복잡도를 지나치게 높게 할 수 있기 때문에 본 논문의 실험에서는 캐쉬에 있는 객체들에 대해서만 과거 참조 시간을 유지하는 방법을 사용했다. 실험에 이용한 트레이스는 NLANR에서 학술용으로 제공하는 웹 프락시 캐쉬의 공개 트레이스[3]로서, 그 기본적인 특징은 표 2와 같다. 이는 미국 캘리포니아 지방에 설치된 웹 프락시 서버의 로그에서 얻은 트레이스로서 웹 캐싱 알고리즘의 성능 측정에 사용하는 대표적인 트레이스이다.

표 2 실험에 사용한 트레이스의 특성

Trace	Total Requests	Total Unique Requests	Total MBytes	Total Unique MBytes
sj_sanitized-access_990207	104473	80829	1174.71	868.328
sj_sanitized-access_990211	21888	14941	287.794	120.151

그림 3과 그림 4는 기존의 캐쉬 교체 알고리즘들과 LUV 방법을 표 2에서 제시한 트레이스를 이용하여 모의 실험을 한 결과 비용절감률을 보여주고 있다. 이 때, 인출 비용(c_i)에 대한 정의는 HYBRID, LNC 등 대부분의 기존 방법에서 사용한 것과 마찬 가지로 인출 지연 시간으로 정의하였다. 그림에서 Infinite는 캐쉬 용량을 무한히 크게 했을 경우의 비용절감률을 나타내며, 가로축에 표시한 캐쉬의 크기는 무한 캐쉬 용량에 대한 상대적 크기로 표시한 것이다. 무한 캐쉬 용량은 각 트레이스에 있어 표 2에서 표시한 'Total Unique Mbytes'의 크기를 나타낸다. 비교 대상이 되는 알고리즘의 개수가 많아 가시성을 높이기 위해 각각의 트레이스에 대한 실험 결과를 두 개의 그래프로 나누어 표시하였다. 이질형 캐싱 환경에서의 비용 측면을 잘 반영시킨 알고리즘들은 대체로 그래프의 상단에 두고, 그렇지 않은 알고리즘들은 하단에 두어 알고리즘들 간의 상대적인 비교를 용이하게 하였다. 그림에서 보는 바와 같이 대부분의 경우 비용 측면을 정규화시키는 방법으로 고려한 상단의 알고리즘들이 그렇지 않은 하단의 알고리즘들보다 더 우수한 성능을 나타내었다는 것을 알 수 있다. 또한, 캐쉬 용량이 작은 경우에는 알고리즘들 간의 성능 차이가 뚜렷하나 캐쉬 용량이 큰 경우에는 어떤 알고리즘을 사용하든 간에 성능 차이가 그다지 크지 않다는 점을 알 수 있다. 그림에서 보는 바와 같이 LUV는 대부분의 경우 가장 우수한 성능을 나타내었다. 그 다음으로 GD-SIZE가 우수한 성능을 나타내었으며 LRV, HYBRID, LNC 등이 트레이스 종류 및 캐쉬 크기에 따라 조금씩 우열을 보이며 그 뒤를 따랐다. 인출 비용의 차이를 제대로 고려하지 않은 LRU, SIZE 등은 캐쉬 용량이 클 경우에는 다른 알고리즘들과 비슷한 성능을 나타낸 반면, 캐쉬 용량이 작은 경우에는 좋지 않은 성능을 나타내었다. 캐쉬 용량이 작은 경우 LUV가 GD-SIZE보다 우수한 성능을 나타낸 것은 과거 참조

3) ftp://ircache.nlanr.net/Traces/에서 얻을 수 있으며 자세한 특성은 http://www.ircache.net/Cache에서 찾을 수 있다.

기록에 있어 GD-SIZE가 참조 빈도(frequency)를 전혀 고려하지 않고 가장 최근 참조 시점만을 고려하기 때문이다. 즉, 이 두 알고리즘의 성능 차이는 바로 과거의 참조 빈도가 미래 참조 경향을 설명해 주는 부분이라 할 수 있다. 반면, 최근 참조 시점을 전혀 고려하지 않고 과거 참조 회수만을 고려한 HYBRID 기법이나 과거 k번째 참조된 시각만을 고려하는 LNC 방법은 캐쉬 용량이 작을 경우 LUV 및 GD-SIZE보다 좋지 않은 성능을 나타내었는데, 이는 과거 참조 기록 중 참조 회수(frequency)보다는 최근 참조 성향(recency)이 미래 참조의 예측에 좀 더 좋은 힌트를 준다는 것을 암시한다. 하지만, 이 둘을 동시에 고려한 LUV는 전체 결과에서 LUV 다음으로 좋은 결과를 나타낸 GD-SIZE보다 소용량의 캐쉬에서는 6.6%에서 52%까지 더 우수한 성능을 나타내었다.

6. 결 론

본 논문에서는 광역 분산 환경에서의 이질형 객체에

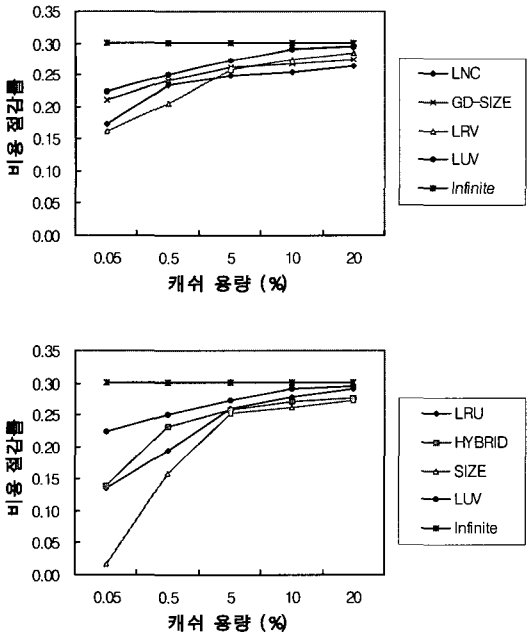

그림 3 sj_sanitized-access_990207 트레이스를 이용한 각 알고리즘의 성능 비교

대한 캐쉬 기법이 고려해야 할 사항들을 여러 가지 측면에서 분석하고, 기존의 연구가 지니는 한계를 극복하는 새로운 캐쉬 교체 알고리즘을 설계하였으며, 그 효과를 트레이스 기반 모의 실험을 통해 검증하였다.

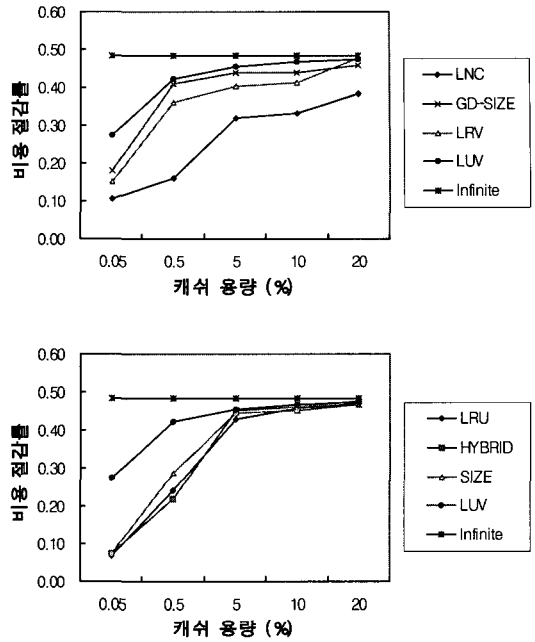

그림 4 sj_sanitized-access_990211 트레이스를 이용한 각 알고리즘의 성능 비교

이질형 객체에 대한 캐쉬 교체 알고리즘은 캐쉬 내의 객체에 대한 평가시 참조 빈도와 최근 참조 시각 등 과거 참조 기록에 의한 평가 외에 객체들의 인출 비용 차이를 함께 고려한 합리적인 평가가 가능해야 하며, 캐쉬 운영의 시간 복잡도 및 공간 복잡도 측면에서 현실성이 있어야 한다. 본 논문이 제안한 방법은 과거 참조 기록을 사용하는데 있어 기존 방법들처럼 제한된 과거 기록만을 이용하는 것이 아니라 모든 과거 참조 기록을 참조 시점의 최근성에 근거하여 반영하며, 여기에 객체의 이질성에 의한 가치를 함께 고려하여 객체들에 대한 정규화된 가치 평가를 가능하게 한다. 또한, 본 논문의 방법은 캐쉬 운영의 시간 복잡도가 $O(\log_2 n)$으로 매우 효율적이며, 객체당 부가적으로 저장되는 정보 역시 $O(1)$로서 매우 효율적이다. NLANR의 공개 트레이스를 이용한 모의 실험을 통해 본 논문의 알고리즘은 기존의 교체 알고리즘인 LRU, SIZE, HYBRID, LRV, GD-SIZE, LNC보다 우수한 성능을 나타낸다는 것을 검증할 수 있었다.

한편, 본 논문에서는 캐쉬 교체 알고리즘의 설계시 2.2.3절에서 설명한 통계적 정보는 활용하지 않았다. 하지만, 이러한 통계적 정보를 활용하여 *Value*를 정의하더라도 캐쉬 내에 서 참조되지 않은 객체들의 *Value*에 대해서는 대소 관계가 그대로 유지되기 때문에 힙에 기반한 $O(\log_2 n)$의 효율적인 구현이 여전히 가능하다. 따라서, 본 논문의 향후 연구 과제로 데이타베이스나 웹 캐싱 등 특정 환경 및 작업 부하에서 의미 있는 통계적 정보를 분석하여 이를 객체에 대한 가치 평가에 활용할 생각이다.

참 고 문 헌

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching proxies: Limitations and potentials," *Proc. 4th Int'l WWW Conf.*, pp.119-133, 1995.

[2] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms," *Proc. USENIX Symp. on Internet Technology and Systems*, pp. 193-206, 1997.

[3] L. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM System Journal*, vol.5, pp.78-101, 1966.

[4] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox, "Removal policies in network caches for world-wide web documents," *Proc. ACM SIGCOMM*, pp.293-305, 1996.

[5] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-k Page Replacement Algorithm for Database Disk Buffering," *Proc. ACM SIGMOD*, pp. 297-306, 1993.

[6] R. Wooster and M. Abrams, "Proxy caching that estimates page load delays," *Proc. Sixth Int'l WWW Conf.*, 1997.

[7] P. Scheuermann, J. Shim, and R. Vingralek, "WATCHMAN: A Data Warehouse Intelligent Cache Manager," *Proc. 22nd VLDB Conf.*, 1996.

[8] M. Garey and D. Johnson, *Computers and Intractability*, W. H. Freeman and Company, San Francisco, 1979.

[9] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Design: Algorithms, Implementation and Performance," *IEEE Trans. on knowledge and data engineering*, vol.11, no.4, 1999.

[10] J. Spirn, *Program Behavior: Models and Measurements*, Elsevier North-Holland, 1977.

[11] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," *Proc. 20th VLDB Conf.*, pp.439-450, 1994.

[12] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the LRU and LFU Policies," *Proc. ACM SIGMETRICS*, pp.134-143, 1999.

[13] P. Lorenzetti, L. Rizzo, and L. Vicisano, "Replacement Policies for a Proxy Cache," Technical Report, http://www.iet.unipi.it/~luigi/caching.ps.gzeunis, 1996.

반 효 경

1997년 서울대 전산과학과 학사. 1999년 전산과학과 석사. 1999년 ~ 현재 서울대 전산과학과 박사 과정. 관심분야는 운영체제, 알고리즘, 웹 캐싱, 유전 알고리즘 등

노 삼 혁

정보과학회논문지 : 시스템 및 이론 제 27 권 제 3 호 참조

민 상 렬

정보과학회논문지 : 시스템 및 이론 제 27 권 제 3 호 참조

고 건

1974년 서울대 응용물리학과 학사. 1979년 미국 버지니아대학교 전산학 석사. 1981년 미국 버지니아 대학교 전산학 박사. 1974년 ~ 1976년 KIST 연구원. 1981년 ~ 1983년 Bell Lab연구원. 1988년 IBM 왓슨 연구소에서 분산 운영체제를 1년간 연구함. 1991년 ~ 1993년 서울대 중앙교육연구전산원 원장. 1983년 ~ 현재 서울대 전산과학과 교수. 관심분야는 운영체제, 컴퓨터 구조, 컴퓨터 시스템 성능 분석, 분산처리 시스템 등