JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 25 No. 3, pp. 11-17, March 2020
https://doi.org/10.9708/jksci.2020.25.03.011

Cache Memory and Replacement Algorithm Implementation and Performance Comparison

Na Eun Park*, Jongwan Kim**, Tae Seog Jeong***

*Student, Dept. of Convergence Security Engineering, Sungshin Women's University, Seoul, Korea.
**Professor, Smith College of Liberal Arts, Sahmyook University, Seoul, Korea.
***Professor, Dept. of Business Administration, Sahmyook University, Seoul, Korea.

[Abstract]

In this paper, we propose practical results for cache replacement policy by measuring cache hit and search time for each replacement algorithm through cache simulation. Thus, the structure of each cache memory and the four types of alternative policies of FIFO, LFU, LRU and Random were implemented in software to analyze the characteristics of each technique. The paper experiment showed that the LRU algorithm showed hit rate and search time of 36.044% and 577.936ns in uniform distribution, 45.636% and 504.692ns in deflection distribution, while the FIFO algorithm showed similar performance to the LRU algorithm at 36.078% and 554.772ns in even distribution and 45.662% and 489.574ns in bias distribution. Then LFU followed, Random algorithm was measured at 30.042% and 622.866ns at even distribution, 36.36% at deflection distribution and 553.878ns at lowest performance. The LRU replacement method commonly used in cache memory has the complexity of implementation, but it is the most efficient alternative to conventional alternative algorithms, indicating that it is a reasonable alternative method considering the reference information of data.

▸ **Key words**: Cache Replacement, LRU, LFU, FIFO, Random Replacement

[요 약]

본 논문은 캐시 시뮬레이션을 통해 각 교체 알고리즘의 캐시 히트(Cache Hit) 및 검색시간을 측정함으로써 캐시 교체 정책에 대한 실용적인 결과를 제시한다. 프로세서의 성능이 향상되면서 캐시 메모리 또한 성능을 향상하기 위한 많은 연구가 활발히 진행되고 있다. 캐시 메모리는 일반적으로 LRU(Least Recently Used) 교체방식을 사용하고 있으며 LRU 방식 이외에도 대표적으로 FIFO(First-In First-Out), LFU(Least Frequently Used) 및 Random 교체방식이 있다. 논문에서는 캐시 메모리 구조 및 교체 알고리즘을 소프트웨어로 구현하여 각 기법의 특징을 분석한다. 논문의 실험결과 LRU 알고리즘이 균등 분포에서 36.044%, 577.936ns, 편향 분포에서 45.636%, 504.692ns의 히트율(Hit ratio)과 검색시간을 보였으며, FIFO 알고리즘은 균등 분포에서 36.078%, 554.772ns, 편향 분포에서 45.662%, 489.574ns로 LRU와 유사한 성능을 보였다. Random 교체방식은 균등 분포에서 30.042%, 622.866ns, 편향 분포에서 36.36%, 553.878%로 가장 낮은 성능을 보였다. 이는 캐시 메모리에서 일반적으로 사용되는 LRU 교체방식이 타 교체 알고리즘보다 최선의 성능을 보이면서도 데이터의 참조 정보를 고려하는 합리적인 알고리즘임을 나타내는 것이다.

▸ **주제어**: 캐시 교체, 최근최소사용, 최소빈도사용, 선입선출, 무작위 교체기법

• First Author: Na Eun Park, Corresponding Author: Tae Seog Jeong
*Na Eun Park (pne0521@gmail.com), Dept. of Convergence Security Engineering, Sungshin Women's University
**Jongwan Kim (kimj@syu.ac.kr), Smith College of Liberal Arts, Sahmyook University
***Tae Seog Jeong (bigstone@syu.ac.kr), Dept. of Business Administration, Sahmyook University
• Received: 2020. 03. 05, Revised: 2020. 03. 25, Accepted: 2020. 03. 26.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

캐시 메모리(Cache Memory)는 프로세서-메모리 간 속도 차이를 줄이기 위해 완충 장치로 사용되고 있는 고속 메모리이다. 캐시 메모리가 없는 시스템은 중앙처리장치(CPU, Central Processing Unit)가 명령을 실행할 때마다 매번 메모리에 접근하여 필요한 명령 데이터를 찾게 되므로 많은 시간적 비용이 소모된다. 그러나 캐시 메모리를 사용하면 메모리에 접근하기 이전에 캐시 메모리를 통해 데이터를 가져올 수 있어서 적은 시간 비용 대비 성능향상을 기대할 수 있다. 이러한 이유로 프로세서(Processor, CPU)의 향상과 더불어 캐시 메모리의 성능향상을 위한 연구도 활발하게 진행되는 추세다.

캐시 메모리에 접근했을 때, 찾고자 하는 데이터가 캐시 메모리에 존재하면 적중 혹은 캐시 히트(Cache hit)이며 존재하지 않는 경우는 실패 혹은 캐시 미스(Cache Miss)라 한다. 일반적으로 캐시 메모리의 성능을 나타내기 위해서 히트율을 이용한다. 캐시 메모리는 전체 메모리 데이터의 일부만 포함하므로, 캐시 메모리를 효율적으로 이용하기 위해서는 히트율을 높일 수 있는 캐시 교체 정책 사용이 필요하다.

이론상으로 알려진 가장 이상적인 캐시 교체 정책은 Belady의 MIN 알고리즘인 OPT(Optimal) 알고리즘이다. MIN 알고리즘은 가장 오랫동안 참조되지 않을 데이터를 교체하는 방식이다. 그러나 미래의 참조 여부를 예측할 수 없으므로 구현할 수 없으므로 타 알고리즘의 비교 지표 역할로만 사용하고 일반적으로는 LRU(Least Recently Used) 방식을 사용한다. LRU 방식 이외에도 FIFO, LFU 및 Random 교체방식이 존재한다 [1].

본 논문에서는 언급된 LRU, FIFO, LFU, Random 총 4가지 교체 정책을 포함하여 캐시 메모리 구조를 소프트웨어로 구현한다. 구현된 알고리즘으로 시뮬레이션 실험을 통해 주요 평가 지표인 히트율 및 검색시간을 측정한다.

본 논문의 구성은 다음과 같다. II 절에서 관련 연구에 관해 소개하고, III 절에서는 시뮬레이션 구현과정 및 환경을 소개한다. IV 절에서는 시뮬레이션 실험을 통한 정책별 히트율 및 시간을 비교한다. 마지막으로 V 절에서 향후 연구 방향과 함께 결론을 맺는다.

II. Preliminaries

1 FIFO Replacement

FIFO 교체방식은 각 데이터가 메모리에 적재될 때마다 시간 정보를 저장 및 활용하는 방식이다. 데이터 부재가 일어났을 때 가장 먼저 적재된 데이터를 우선으로 교체한다.

Fig. 1은 FIFO 교체방식에 따라 데이터를 참조할 때 캐시 메모리의 변화를 나타낸 것이다. 0-4-1의 순서로 참조된 후 4가 참조될 경우, 캐시에 이미 적재된 데이터이므로 변화 없이 참조를 진행한다. 그러나 2가 참조되어 캐시 메모리 부재가 발생하면 FIFO 교체방식에 따라 가장 먼저 참조된 0을 교체하고 해당 위치에 2를 적재한다.

FIFO 교체방식은 이해하기 쉽고 간단하게 구현할 수 있으나 데이터의 중요도나 참조 빈도와 관계없이 교체 대상이 된다는 한계점이 존재한다 [2].

Reference Number	0	4	1	4	2	4	3	4	2	0
Cache Memory	0	0	0	-	2	-	2	2	-	0
	-	4	4	-	4	-	3	3	-	3
	-	-	1	-	1	-	1	4	-	4

Fig. 1. Example of FIFO Replacement

2 LRU Replacement

LRU 교체방식은 최근에 가장 오랫동안 사용되지 않은 데이터를 교체하는 알고리즘이다. 가장 오랫동안 접근되지 않은 데이터는 앞으로도 접근되지 않을 것이라는 시간 지역성(Temporal Locality)을 고려하여 설계되었으며 캐시 메모리에서 일반적으로 사용되고 있는 교체방식이다 [3, 4].

LRU 교체방식에 따른 데이터 변화는 Fig. 2와 같다.

데이터 3을 참조하여 캐시 미스가 발생한 경우, 메모리 내에서 가장 오랫동안 참조되지 않은 1이 교체되고 3을 적재한다.

LRU 알고리즘 구현을 위해서는 데이터별 참조된 시간 정보가 기록·저장되어 있어야 한다. 따라서 LRU 교체방식 사용 시 각 데이터에 대한 참조 시간 정보를 포함하여야 하므로 오버헤드가 발생하고 구현하기 복잡하다는 한계가 존재한다 [2].

Reference Number	0	4	1	4	2	4	3	4	2	0
Cache Memory	0	0	0	-	2	-	2	-	-	2
	-	4	4	-	4	-	4	-	-	4
	-	-	1	-	1	-	3	-	-	0

Fig. 2. Example of LRU Replacement

3 LFU Replacement

LFU 교체방식은 각 데이터의 참조횟수를 기록하여 이용한 교체방식이다. Fig. 3과 같이 2가 참조되어 데이터 부재가

발생했을 때 참조횟수가 가장 적은 데이터 0을 교체한다.

LFU 교체방식은 데이터의 참조 정보만 기록하기 때문에 참조의 최근성을 고려하지 않는다. 이러한 특성으로 인해 과거에 빈번하게 참조된 데이터가 캐시 메모리를 차지하는 문제가 발생한다. 즉, 최근에 참조된 데이터가 적재된 후 새로운 데이터를 참조하여 캐시 미스가 발생했을 경우 참조횟수가 낮다는 이유로 교체 대상이 되어 충분한 시간 동안 머물지 못할 수 있다. 따라서, 과거에 빈번히 사용된 데이터가 참조횟수가 많아 캐시를 차지하는 캐시 오염(Cache Pollution)이 발생한다 [5].

Reference Number	0	4	1	4	2	4	3	4	2	0
Cache Memory	0	0	0	-	2	-	3	-	2	0
	-	4	4	-	4	-	4	-	4	4
	-	-	1	-	1	-	1	-	1	1

Fig. 3. Example of LFU Replacement

4 Conventional Studies

프로세서와 메모리의 성능을 향상하기 위한 연구는 현재도 활발하게 진행 중이며 그중에서도 교체 정책에 관한 연구도 지속적으로 이루어지고 있다. 메모리 교체 정책에 관한 이전의 연구에서는 기존에 사용되고 있는 교체 정책들의 문제점과 성능적 한계점을 분석하고 이를 보완하거나 새로운 알고리즘을 제안하여 기존 교체 정책과의 성능을 비교하는 연구가 수행되었다 [4, 5, 6, 7, 8, 9, 13].

각 교체 정책의 성능을 비교하는 연구에서는 특정 환경을 가정하고 해당 환경 기반에서의 알고리즘 분석이나 특정 환경에서 적용할 수 있는 알고리즘을 제안하는 연구를 찾아볼 수 있다 [10, 11, 12]. 그러나 기존 연구들은 환경에 영향을 받지 않고 알고리즘 자체에 초점을 두어 성능을 분석하지 않았다.

지역성에 따른 교체 정책의 성능에 관한 또 다른 연구 [3]에서는 참조 지역성의 수준에 따라 교체 알고리즘 성능이 어떻게 변화하는지 연구하였다. 해당 연구는 참조 열의 다양성을 통해 다양한 페이지가 존재할 때 페이지 교체 알고리즘은 어떻게 동작하는가에 초점을 두었다. 또한, 참조할 페이지를 대신하여 0 ~ 9까지의 정수를 나타내도록 하고 프레임의 크기 및 참조 열의 크기를 변화시켜 다양한 상황에서 비교할 수 있도록 하였다. 실험에서는 FIFO, LRU, OPT 알고리즘을 프로그램으로 구현하고 시뮬레이션을 통해 결과를 도출하였다 [3].

본 논문에서는 FIFO, LRU, LFU 및 Random 알고리즘을 프로그램으로 구현하고, 캐시 메모리의 크기는 실제 프로세서에서 사용하는 Level 1 ~ Level 3의 메모리 크기와 비례하게 메모리 크기를 설정한다. 참조 데이터는 0 ~ 50000까지의 데이터를 설정하여 실제 프로세서 작업 환경과 유사하게 구현하여 시뮬레이터를 구현하여 각 교체 정책의 성능을 실험한다.

III. Implementation

본 절에서는 논문의 캐시 시뮬레이션을 위한 알고리즘 및 구현과정과 실행 환경에 관해 설명한다.

코드 구현은 Windows 10, Visual Studio 2019 버전에서 C를 이용하여 시뮬레이션 코드를 작성하였다.

캐시 메모리의 계층적 구조 및 접근 순서는 Fig. 4와 같고 다음과 같이 설계한다. 검색 데이터의 범위는 1 ~ 50000까지의 정수형 데이터로 가정하고 맨 하위 계층인 HDD 계층에 모두 저장되어 있도록 한다. 접근 순서는 하나의 임의 데이터가 필요할 때 Level 1 Cache를 우선으로 접근하여 데이터의 존재 여부를 확인한다. 데이터가 Level 1 Cache에 존재하지 않는 경우, 다음 레벨 계층에 차례로 접근한다.

Level 1 ~ Level 3의 Cache 계층 내에 필요 데이터가 있는 경우 캐시 히트, Level 3 Cache까지 접근하였으나 찾는 데이터가 없는 경우 캐시 미스로 간주한다. 캐시 미스인 경우 RAM, HDD 계층 순서로 접근하여 필요 데이터를 찾는다. 각 계층에서 찾은 데이터는 발견 데이터를 기점으로 Level 1 Cache까지 각 계층에 역방향으로 접근하며 저장한다. 접근한 계층의 데이터가 꽉 찬 경우에는 구현한 캐시 교체 정책을 통해 데이터를 교체한다.

각 계층에 접근 시 계층별로 다른 접근시간이 발생하며 계층별 접근시간은 Level 1 Cache의 경우 5ns, Level 2 Cache는 10ns, Level 3 Cache는 15ns, RAM과 HDD 계층은 각각 50ns, 1000ns로 가정한다. 접근시간은 CPU에 의한 실행시간의 영향을 받지 않도록 각 저장 장치에 접근할 때마다 정수형 변수에 가정한 값을 더하는 방식으로 구현한다. 본 논문에서 구현한 시뮬레이션의 경우 검색 과정에서의 순방향 접근 시에만 접근시간을 발생시키고 데이터 검색 후 저장 과정에서 발생하는 역방향 접근에는 접근시간을 계산하지 않는다.

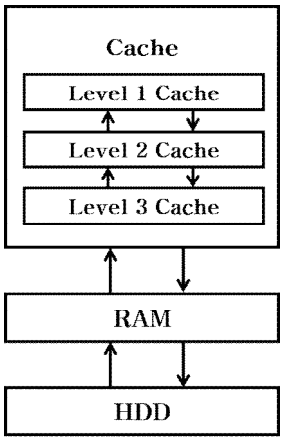

Fig. 4. Memory Hierarchy

본 논문에서의 캐시 히트율 및 검색시간 측정은 하나의 데이터 세트를 이용한 검색 과정 동안의 총 캐시 히트 횟수 및 접근시간을 각각의 변수에 누적하여 저장한다. 그 결괏값을 데이터 세트에서의 데이터 참조횟수로 나누어 평균값을 반환한다. 계산된 평균값은 일정한 구간마다 파일에 출력하여 각 데이터 세트별 결과를 저장한다.

계층별 캐시 메모리의 크기는 Fig. 5와 같다. 실제 캐시 메모리의 메모리 비율과 유사하게 구현하기 위하여 Level 별 메모리 크기를 512, 2048, 20480으로 설정하였으며, RAM과 HDD 계층은 30000과 50000으로 설정하였다.

```
1   #define L1_SIZE 512
2   #define L2_SIZE 2048
3   #define L3_SIZE 20480
4   #define RAM_SIZE 30000
5   #define HDD_SIZE 50000
6
7   typedef struct {
8      int front;
9      int rear;
10     int* data;
11  } QueueType;
```

Fig. 5. Memory Size and Queue

LRU 교체방식을 구현할 경우, 모든 데이터에 대한 참조 정보를 저장하여야 하므로 많은 구현비용이 들고 구현이 복잡하다는 단점이 있다. 본 논문에서는 이러한 단점을 해결하고자 Queue 형식의 자료구조를 사용하여 캐시 메모리를 구현한다.

Fig. 6은 LRU, FIFO, LFU, Random 교체 정책을 C언어로 구현한 함수 알고리즘이다.

```
1   void LRU(QueueType* q, int size, int num) {
2      int index = check(q, size, num);
3      if (index != -1) {
4         drop(q, size, index);
5         q->data[++q->rear] = num;
6      }
7      else {
8         if (is_full(q, size)) {
9            drop(q, size, 0);
10        }
11        q->data[++(q->rear)] = num;
12     }
13  }
14
15  void FIFO(QueueType* q, int size, int num)
    {
16     int index = check(q, size, num);
17     if (index == -1) {
18        if (is_full(q, size)) {
19           drop(q, size, 0);
20        }
21     q->data[++(q->rear)] = num;
22     }
23  }
24
25  void LFU(QueueType* q, int freq[], int size,
    int num) {
26     int index = check(q, size, num);
27     if (index != -1) {
28        freq[index] ++;
29        }
30     else {
31        if (is_full(q, size)) {
32           int min = freq[0], min_index = 0;
33           for (int i = 1; i < size; i++) {
34              if (min > freq[i]) {
35                 min_index = i;
36                 break;
37              }
38           }
39           q->data[min_index] = num;
40           freq[min_index] = 0;
41        }
42        q->data[++q->rear] = num;
43     }
44  }
45
46  void RANDOM(QueueType* q, int size, int
    num) {
47     int index = check(q, size, num);
48     if (index == -1) {
49        int rIndex = rand() % size;
50        q->data[rIndex] = num;
51     }
52  }
```

Fig. 6. Implementation Algorithm

Fig. 7은 Queue 형식으로 캐시 메모리를 구현하였을 때 LRU 교체 알고리즘의 동작 방식이다. 데이터를 저장할 때마다 차례로 데이터를 저장하고, 맨 마지막 노드인 rear를 증가시킨다. 교체 시 맨 좌측(front) 위치의 데이터를 교체하고 맨 우측(rear)에 새 데이터를 저장한다. 이미 저장된 데이터를 참조하는 경우, Queue 데이터를 갱신하여 참조 정보 갱신과 같은 효과가 있도록 구현한다.

Time	Cache Memory size = 5				
1	3	-	-	-	-
2	3	2	-	-	-
3	3	2	5	-	-
4	3	2	5	1	-
5	3	2	5	1	7
6	2	5	1	7	9
7	5	1	7	9	6
8	5	1	9	6	7
9	5	1	9	6	7
10	5	1	9	7	6

Replacement Update

Fig. 7. Example of LRU Algorithm

IV. Experiment

실험에서는 Windows 10 내에서 Microsoft Visual Studio 2019 버전에서 C언어로 작성한 코드를 이용하였다. 시뮬레이션을 진행한 PC의 환경은 Intel Core i7-1065G7, 1.3GHz, 8GB RAM이다.

시뮬레이션에 사용한 데이터는 설정한 데이터 범위(1 ~ 50000) 내에서 rand() 함수를 이용, 데이터 10만 개를 무작위로 추출하여 사용하였다. Fig. 8과 같이 균등한 분포를 가진 데이터 세트, Fig. 9와 같은 30000 이내 범위로 편향된 편향 데이터 세트를 각각 5세트씩 추출하여 총 10세트의 데이터 세트를 이용하여 실험하였다. 실험은 각 데이터 세트마다 10만 개 데이터를 전체를 하나씩 차례로 접근하며 검색하는 시뮬레이션을 수행하도록 진행하였으며 각 교체 알고리즘별 히트율 및 검색시간을 비교하였다.

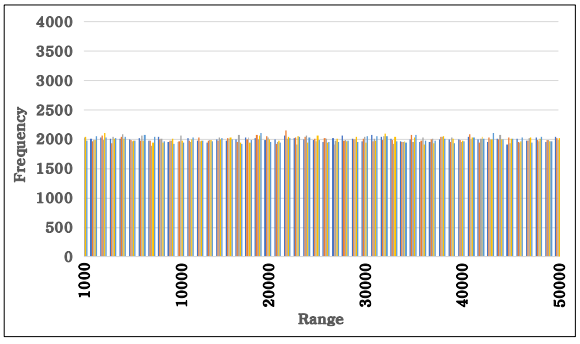

Fig. 8. Uniform Data Distribution

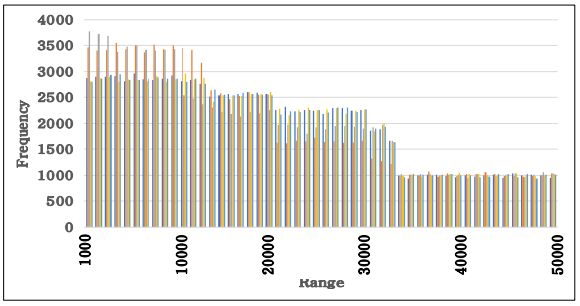

Fig. 9. Skewed Data Distribution

실험결과에 대한 그래프는 Fig. 10, Fig. 11과 같다. 균등 분포 데이터 세트를 이용한 경우 약 10000회의 데이터를 검색했을 때는 각 알고리즘 간 캐시 히트의 차이가 크게 드러나지 않았다. 그러나 30000회를 기점으로 Random 알고리즘과 나머지 세 알고리즘의 차이를 확인할 수 있었고 전체적으로 LRU, LFU, FIFO 알고리즘 간의 차이는 크게 나타나지 않았다.

편향 분포 데이터 세트를 사용할 경우 각 알고리즘의 캐시 히트율이 균등 분포 데이터 세트에 비해 높게 측정된 것을 확인할 수 있었다.

데이터 세트별 히트율에서는 균등 분포 데이터 세트와 같이 Random 알고리즘이 가장 낮은 성능을 보였고 LRU, FIFO 알고리즘이 유사하였으며, LFU 알고리즘이 LRU, FIFO 알고리즘보다 근소하게 낮았다.

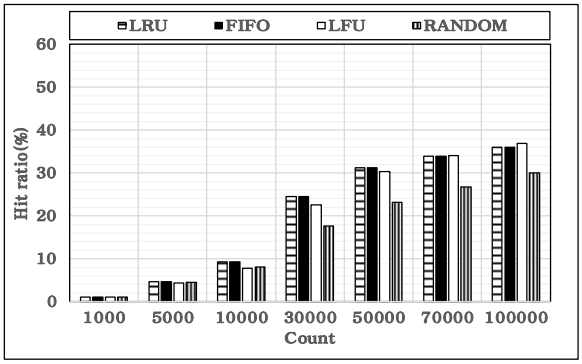

Fig. 10. Hit Ratio of Equality Data Distribution

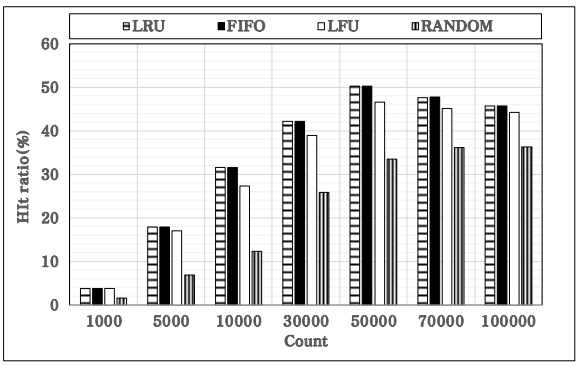

Fig. 11. Hit Ratio of Biased Data Distribution

데이터 간 검색시간 측정 결과 그래프는 Fig. 12, Fig. 13과 같이 측정되었다. 균등 분포 데이터 세트에서는 10000회까지 알고리즘 간 성능의 차이가 없다가 30000회를 기점으로 점차 차이가 크게 나타나는 것을 확인할 수 있었다.

편향 분포 데이터 세트의 경우에는 LRU, LFU, FIFO 세 알고리즘의 탐색시간은 유사하게 변화하였다. 그러나 Random 알고리즘의 경우 10000회까지 낮은 폭으로 검색시간이 감소하였으나 30000회를 기점으로 큰 폭으로 감소하는 것을 확인할 수 있었다.

두 데이터 세트 모두 30000회를 기점으로 알고리즘 간 차이가 벌어지기 시작했으며, Random 알고리즘이 나머지 세 교체 알고리즘에 비해 낮은 히트율과 높은 검색시간이 측정되었다.

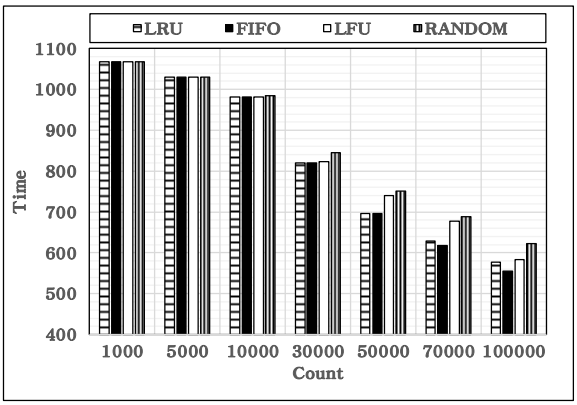

Fig. 12. Search Time of Equality Data Distribution

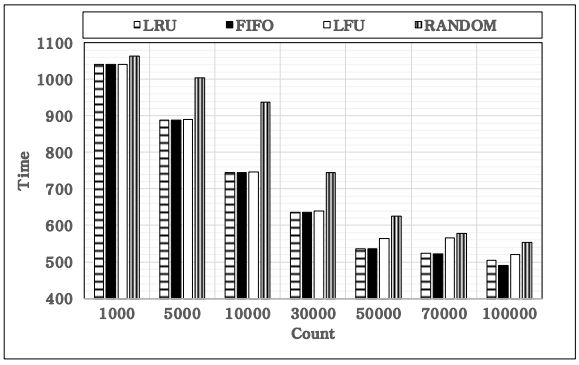

Fig. 13. searching Time of Biased Data Distribution

캐시 히트 측면에서 Random 알고리즘을 제외한 세 교체 알고리즘이 유사한 성능을 보였다. 그러나 반복 횟수가 늘어날수록 FIFO, LRU, LFU 순서로 검색시간이 느렸다. 따라서 FIFO 알고리즘보다 LRU, LFU 교체 알고리즘 순서로 HDD 계층까지 접근한 횟수가 더 많음을 알 수 있다.

V. Conclusions

본 논문에서는 Cache, RAM, HDD를 포함한 저장 메모리 계층 및 캐시 메모리에서의 데이터 교체 방법인 LRU, LFU, FIFO 및 Random 교체 알고리즘을 소프트웨어로 구현함으로써 알고리즘별 히트율 및 검색시간을 비교하였다.

균등 분포 데이터 세트보다 편향 분포 데이터 세트가 히트율, 검색시간 면에서 좋은 성능을 보였으며, LRU 교체 정책과 FIFO 교체 정책이 가장 뛰어난 성능을 보였다. 따라서 두 알고리즘 중에서도 데이터의 참조 정보를 고려하여 데이터를 교체하는 LRU 교체방식이 기존 교체방식들보다 효율적이고 합리적인 알고리즘임을 알 수 있다.

본 논문에서의 실험결과 LRU 알고리즘이 데이터들의 참조 정보를 고려하면서도 가장 높은 효율을 보이는 알고리즘임을 알 수 있었다. 그러나 LRU 알고리즘은 단순히 참조 시간만으로 교체 대상을 선택하기 때문에 데이터의 참조 빈도를 고려하지 않는다. 그러므로 데이터들이 차례로 일정하게 참조될 경우 높은 성능을 보이지 못하는 문제가 있다. 향후 LRU 알고리즘의 문제점을 분석하여 이를 보완하는 새로운 알고리즘을 연구하고자 한다.

ACKNOWLEDGEMENT

This paper was supported by the Sahmyook University Research Fund in 2019.

REFERENCES

[1] Jongjung Woo, “Computer Architecture,” Hanbit Academy, 2014.

[2] Jeong-Gook Koh, “Design and Implementation or a Web-based Simulator for Educating Page Replacement Algorithms,” Journal of Korea Multimedia Society, Vol. 15, No. 4, Korea Multimedia Society, Korea, pp. 552-559, April 2012.

[3] Chang-Gweon Son, Seung-Gi Lee, Seung-Gu Lee, Seung-HeeKim, “An Exploratory Study on the Effects of Reference Locality on Page Replacement Performance,” The Journal of Korean Institute of Information Technology, Vol. 15, No. 9, pp. 37-47, Korean Institute of Information Technology, Korea, September 2017, DOI : 10.14801/jkiit.2017.15.9.37

[4] Mikyung Lee, Duki Lee, Mincheol Shin, Sanghyun Park, “SWSC(Sequential Write Spatial Clock) Buffer Replacement Algorithm For Mobile Flash Storage,” Proceedings of the Korea Information Processing Society Conference, Vol. 21, No. 2, pp. 771-774, Korea

Information Processing Society, Korea, November 2014.

[5] Donghee Lee, Sam H. Noh, Sang Lyul Mim, Yookun Cho, “LRFU: A Block Replacement Policy which exploits Infinite History of References,” Journal of KISS(A): Computer Systems and Theory Vol. 24, No. 7, pp. 632-641, The Korean Institute of Information Scientists and Engineers, Korea, July 1997.

[6] Seunghoon Lee, Jongmoo Choi, Seongje Cho, Yookun Cho, “Mem ory Reference Patterns and Page Replacement Policies,” Journal of The Korean Institute of Information Scientists and Engineers Vol. 27, No. 2, pp. 50-52, The Korean Institute of Information Scientists and Engineers, Korea, October 2000.

[7] Jae-Seung Ko, Joon-Won Lee, Heon-Young Yeom, “An Alternativ e Scheme to LRU for Efficient Page Replacement,” Journal of KISS(A): Computer Systems and Theory Vol. 23, No. 5, pp. 478-4 86, The Korean Institute of Information Scientists and Engineers, Korea, May 1996.

[8] JongIck Lee, SeungIl Sonh, MoonKey Lee. “The Performance Com parison of the True-LRU and the Pseudo-LRU Algorithm in Cache Memory,” Journal of KISS(A): Computer Systems and Theory Vol. 23, No. 11, pp. 1148-1160, The Korean Institute of Information Scientists and Engineers, Korea, November 1996.

[9] Seon Kim, Sungjae Lee, Inhwan Lee, “Buffer Cache Block Replac ement Algorithm Based on LRU and Prefetching,” Journal of The Korean Institute of Information Scientists and Engineers Vol. 37, No. 2B, pp. 298-303, The Korean Institute of Informativon Scienti sts and Engineers, Korea, November 2010.

[10] Hyoil Lee, Sooyoung Kim, Jonghyun Kim, “Performance Analysi s of Replacement Policies for Internet Proxy Cache,” Journal of The 1999 Fall Conference of The Korea Society For Simulation, pp. 138-143, The Korea Society For Simulation, Korea, October 1999.

[11] Jungmin Yoo, Hoongyu Choi, Ted "Taekyong" Kwon, Yanghee Choi, “Performance comparison of Cache replacement policy in Named Data Network,” Proceedings of Symposium of the Korean Institute of communications and Information Sciences (Fall), pp. 7-8, Korea Institute Of Communication Sciences, Korea, Novemb er 2013.

[12] Do Young Jung, Yong Surk Lee, “Cache Replacement Policy Based on Dynamic Counter for High Performance Processor,” Journal of the Institute of Electronics Engineers of Korea Vol. 50, No. 4, pp. 52-58, The Institute of Electronics and Information Engineers, Korea, Apirl 2013.

[13] Duk Jun Bang, “An Adaptive Cache Replacement Policy to Evict Dirty Block for Last-Level-Cache,” The Graduate School Sejong University, February 2015.

[14] Inkook Chun, Yonghae Kong, Sangho Ha, “Easily written data structure in C language,” Saengneung Publisher, 2019.

[15] Microsoft docs, C/C++ Compiler and build tools errors and warnin gs, *https://docs.microsoft.com.*

Authors

Na Eun Park is currently a Student of Convergence Security Engineering at Sungshin Women's University. Na Eun Park is currently a Student of Convergence Security Engineering at Sungshin Women's University. She is interested in Computer System, Computer Security and Machine Learning.

Jongwan Kim received the Ph.D. degrees in Computer Science and Engineering from Korea University. Dr. Kim joined the faculty of the Smith College of Liberal Arts at Sahmyook University in 2016. He is currently a Professor and he is interested in Big Data, Machine Learning and Distributed Computing.

Tae Seog Jeong received the Ph.D. degrees in MIS from Sogang University, Korea, in 2011. Dr. Jeong joined the faculty of the Department of Business Administration at Sahmyook University, Seoul, Korea, in 2001. He is currently a Professor and he is interested in IT Management, Service Science.