캐시 시뮬레이터 보고서

류지원 202101265
한국외국어대학교 글로벌캠퍼스 컴퓨터공학부
jiwon0913@hufs.ac.kr

요 약

캐시 시뮬레이터를 통해 다양한 조합의 캐시 크기, 블록 크기, 연관성, 교체 정책에 대한 cache miss 를 추적하고 분석하였다. 연구 결과를 바탕으로 “캐시 사이즈와 “연관성”에 따른 “Miss rate”을 블록 크기에 대해 그래프로 나타내어 가장 효과적인 캐시 구조에 대한 결론을 도출하였다. 사전에 설정한 가설인 "fully associative mapping 방식이 대체로 더 효율적인 결과를 도출한다"에 대해서 실험 결과를 통해 옳지 않음을 검증하였다. 또한, 블록 크기가 적절한 크기로 설정되어야 데이터의 일관성을 유지하면서도 cache miss 를 최소화할 수 있다는 것을 확인하였다. 이를 통해 최적의 캐시 구조를 찾기 위해서는 단순히 연관도를 높인다 거나 블록 사이즈를 키우는 것만으로는 충분하지 않으며, 캐시 사이즈, 연관도, 블록 사이즈의 적절한 조합을 고려해야 한다는 결론을 도출하였다.

캐시 시뮬레이터 구현코드 설명

1. CacheBlock

구현하기 위해 생성한 자료구조에 대해 설명하도록 하겠다.

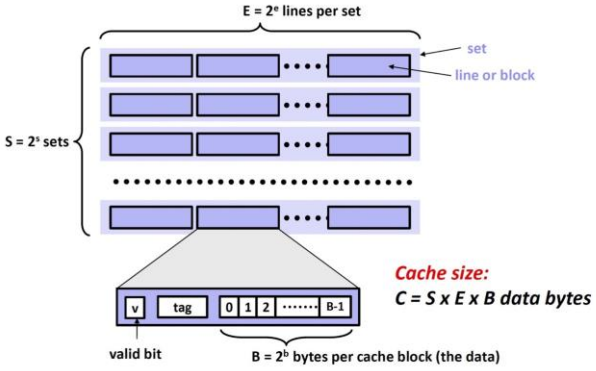
```
// LRU 알고리즘을 위한 자료구조
typedef struct {
    int valid;      // 유효 비트 (1: 유효, 0: 무효)
    int tag;        // 태그 비트
    int timestamp;  // 타임스탬프
    int dirty;      // (1: 일치하지 않음, 0: 일치함)
} CacheBlock;
```

LRU 알고리즘을 위해 생성한 자료구조로, I-cache 와 D-cache 를 따로 생성하지 않고 하나로 선언한 뒤, 각각 구조체 변수명을 달리하여 캐시 구조체를 생성했다.

```
CacheBlock** instruction_cache;  // i-cache
CacheBlock** data_cache;         // d-cache
```

캐시 구조를 최대한 비슷하게 구현하기 위해서 이중 배열로 캐시를 구현했다. (row: entry, column: set.)

배열을 동적으로 할당하기 위해 사용한 방식은 calloc 으로, 생성 시 메모리를 할당하고 해당 메모리를 0 으로 초기화하는 특징이 있어 일일히 0 으로 초기화하는 malloc 동적할당보다 효율적이라고 판단했다.




```
// Create cache arrays for instruction cache and
data cache dynamically
    instruction_cache = calloc(total_set,
sizeof(CacheBlock*));;
    data_cache = calloc(total_set,
sizeof(CacheBlock*));;

    for (int i = 0; i < total_set; i++) {
        instruction_cache[i] = calloc(assoc,
sizeof(CacheBlock));
        data_cache[i] = calloc(assoc,
sizeof(CacheBlock));
    }
```

2. tag

태그는 주소별로 다른 값이 설정되어야 hit 판별이 가능해지기 때문에 태그 값은 다음과 같이 설정했다.

```
int tag = (addr / block_size) * total_set;
```

3. write back

캐시 구현과정에서 가장 많이 고민했던 부분은 캐시가 disk 에 접근하여 write back 을 실행하는 시점이었다. Write back 의 경우 cache hit 일 때 dirty bit 를 기록해두고 disk 에 바로 접근하지 않는다. 캐시가 다 찬 시점에 dirty block 이 캐시에서 빠져나가는 경우 해당 dirty block 을 disk 에 기록한다. 위 로직을 구현하기 위하여 블록에 접근하는 시간을 기록하는 void update_timestamp( ) 함수를 구현했다.

```
void update_timestamp(CacheBlock* cache, int assoc,
int used_block) {
    int i;
    for (i = 0; i < assoc; i++) {
        if (i != used_block) {
            cache[i].timestamp++;
```

```
        }
    }
    cache[used_block].timestamp = 0;
}
```

블록에 접근한 경우 해당 블록의 timestamp 는 0 으로 초기화되고, 같은 set 의 다른 블록들은 timestamp 가 1 씩 증가한다. 한 명령어가 종료되는 시점마다 위 함수로 timestamp 를 업데이트하여 세트 내의 LRU 순서를 기록했다. 캐시를 이중 배열로 생성했기에, 같은 set 의 블록 간 timestamp 를 처리하기 편리했다.

set 가 가득 찬 경우, 함수 int find_lru_block( )을 통해 timestamp 가 가장 큰, 즉 접근한 시간이 가장 오래된 캐시를 리턴했다.

```
int find_lru_block(CacheBlock* cache, int assoc) {
    int lru_block = 0;
    int max_timestamp = cache[0].timestamp;
    int i;
    for (i = 1; i < assoc; i++) {
        if (cache[i].timestamp > max_timestamp) {
            lru_block = i;
            max_timestamp = cache[i].timestamp;
        }
    }
    return lru_block;
}
```

4-1. read operation

d-cache 에서 읽고 쓰는 void read_op( )는 동일 set 내 entry 를 탐색하며 valid bit 가 1 인지 판별하고 tag bit 를 비교하여 cache hit 을 판별했다.

```
    int i;
    for (i = 0; i < assoc; i++) {
        if (data_cache[index][i].valid &&
data_cache[index][i].tag == tag) {
            // Cache hit
            update_timestamp(data_cache[index], assoc,
i);
            return;
        }
    }
```

Cache miss 의 경우에는 d_miss 값을 1 증가시키고 캐시를 적을 LRU block 을 찾는다. update_timestamp( ) 로직에 따라 캐시가 가득 차기 전까지는 값이 들어있는 블록을 갱신하지 않는다. LRU block 이 dirty bit 인 경우에는 해당 dirty block 을 disk 에 적는다. 이때 d_write 값이 1 증가한다. 새로 업데이트된 블록은 유효하고 read 한 값이므로 disk 와 동기화된 값이다. 모든 동작이 끝난 후 timestamp 를 업데이트한다.

```
    // Cache miss
    d_miss++;

    int lru_block = find_lru_block(data_cache[index],
assoc);
    if (data_cache[index][lru_block].dirty) //이걸 왜
확인해? write back 의 경우... dirty bit 확인해
        d_write++;  // Write back to memory if the
block is dirty

    // Update cache with new block
    data_cache[index][lru_block].valid = 1;
    data_cache[index][lru_block].tag = tag;
    data_cache[index][lru_block].dirty = 0;
    update_timestamp(data_cache[index], assoc,
lru_block);
```

void fetch_inst( ) 로직은 read_op( )과 동일하다.

4-2. write operation

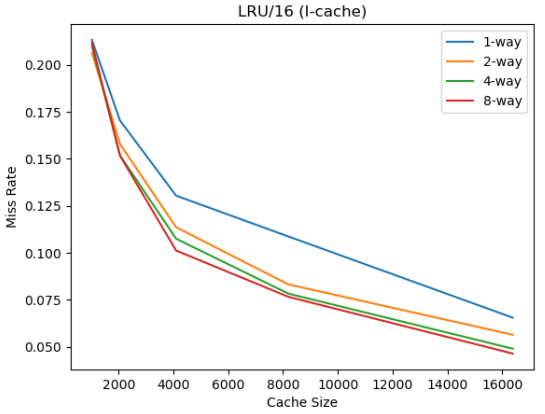
Write back 을 구현하기 위해 write_op( )에서 cache hit 의 경우에는 새로 적은 블록의 dirty bit 을 1 로 설정하여 disk 와 동기화되지 않았다는 것을 기록했다.

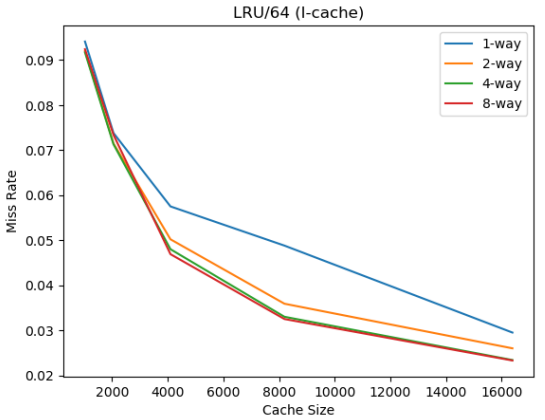
```
    for (i = 0; i < assoc; i++) {
        if (data_cache[index][i].valid &&
data_cache[index][i].tag == tag) {
            // Cache hit
            update_timestamp(data_cache[index], assoc,
i);
            data_cache[index][i].dirty  = 1;
            return;
        }
    }
```

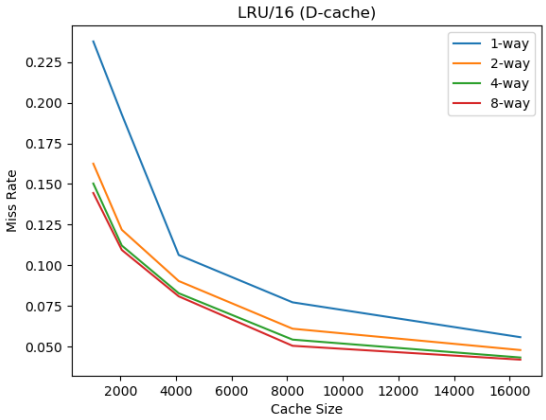
Cache miss 의 경우에는 read_op( )와 로직이 대부분 비슷하나, write 명령어의 주소는 새로 적는 값이므로 disk 와 동기화되지 않은 값이기에 dirty bit 를 1 로 설정한다.

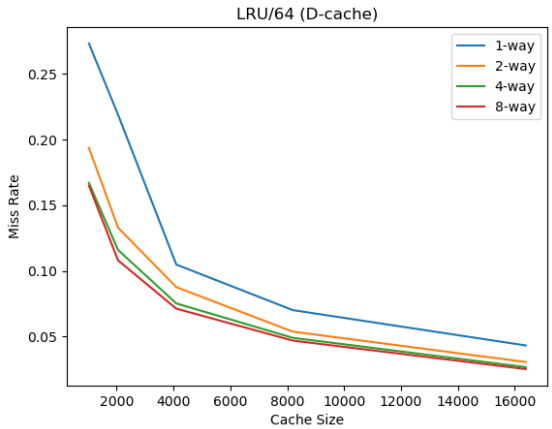
```
    // Cache miss
    d_miss++;
    int lru_block = find_lru_block(data_cache[index],
assoc);
    if (data_cache[index][lru_block].dirty)
        d_write++;  // Write back to memory if the
block is dirty
    // Update cache with new block
    data_cache[index][lru_block].valid = 1;
    data_cache[index][lru_block].tag = tag;
    data_cache[index][lru_block].dirty = 1;
    update_timestamp(data_cache[index], assoc,
lru_block);
```

trace1-out graph









trace1-out.txt

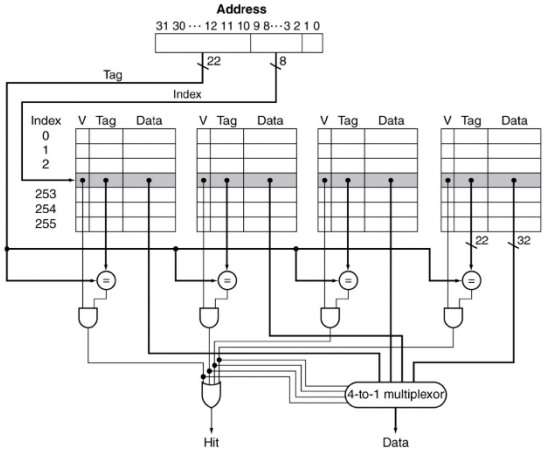
cache size	block size	associative	d-miss rate	i-miss rate	mem write
1024	16	1	0.2376	0.2133	2636
1024	16	2	0.1625	0.2064	1623
1024	16	4	0.1503	0.2101	1461
1024	16	8	0.1445	0.2118	1409
1024	64	1	0.2731	0.0941	2856
1024	64	2	0.1936	0.092	1916
1024	64	4	0.167	0.0917	1467
1024	64	8	0.1649	0.0924	1436
2048	16	1	0.1928	0.1704	2080
2048	16	2	0.1219	0.1579	1126
2048	16	4	0.1122	0.1518	1008
2048	16	8	0.1095	0.1517	998
2048	64	1	0.2189	0.0738	2368
2048	64	2	0.1329	0.0715	1140
2048	64	4	0.1159	0.0713	847
2048	64	8	0.108	0.0734	778
4096	16	1	0.1064	0.1304	902
4096	16	2	0.0904	0.1136	728
4096	16	4	0.0829	0.1075	660
4096	16	8	0.081	0.1012	651
4096	64	1	0.1048	0.0575	831
4096	64	2	0.0876	0.0502	578
4096	64	4	0.0752	0.048	461
4096	64	8	0.0712	0.0469	425
8192	16	1	0.0773	0.1087	536
8192	16	2	0.0611	0.0832	288
8192	16	4	0.0544	0.0783	244
8192	16	8	0.0506	0.0766	217
8192	64	1	0.0701	0.0488	507
8192	64	2	0.0537	0.0359	281
8192	64	4	0.049	0.033	239
8192	64	8	0.0469	0.0325	230
16384	16	1	0.0559	0.0655	191
16384	16	2	0.048	0.0564	107
16384	16	4	0.0434	0.049	31
16384	16	8	0.0421	0.0464	21
16384	64	1	0.0432	0.0295	213
16384	64	2	0.0306	0.026	115
16384	64	4	0.0267	0.0234	85
16384	64	8	0.0252	0.0233	73

ANALYSIS

fully associative mapped 캐시는 hit time 은 많이 소요되나 cache 에 conflict miss 없이 적재가 가능하기에 데이터가 교체될 가능성이 적어 memory 에 접근하는 횟수가 타 연관 방식 대비 적다. 그렇다면 fully associative mapping 방식을 사용하는 것이 대체로 더 효율적인 결과를 도출하지 않을까 하는 의문이 들었었는데, 해당 과제를 통해 해답을 찾을 수 있었다.

과제 수행 후 도표와 그래프를 그려본 결과, 실제로 연관성이 높아질수록 d_write 횟수가 적어지는 것을 확인할 수 있었다. 이는 연관성이 높아질수록 disk 에 접근하는 횟수가 줄어든다는 것을 의미한다. 그러나 연관성에 따른 miss rate 를 그래프로 그려본 결과 2-way 이후에는 연관성에 따른 miss rate 이 유의미하게 줄어들지 않는 것을 확인할 수 있었다. 이 때, 조금이라도 miss rate 이 적은 연관도를 선택하는 것이 옳은 선택일까?

‘가장 효율적인 캐시 구조’를 찾기 위해 고려해야 할 사항은 여러가지가 있다. 우선 direct mapped 캐시에서는 entry 가 들어갈 수 있는 블록이 하나밖에 없기 때문에 인덱스만 사용하여 캐시에 접근하므로 비교기가 하나만 있으면 된다. 4-way 집합 연관 캐시에서는 선정된 집합의 4 개의 원소 중 하나를 선택하는 데 4:1 멀티플렉서와 4 개의 비교기가 필요하다. 집합 연관 캐시를 사용하려면 추가로 필요한 비교기의 비용과 집합의 원소들을 비교하고 선택하는 데 걸리는 시간 지연을 감수해야 한다. [1]



메모리 계층 구조에서 연관 정도를 선택할 때는 miss penalty 대비 연관 구현 비용을 시간과 추가 하드웨어 측면에서 다방면으로 고려해야 한다. 따라서 “연관성이 높을수록 성능이 좋은 캐시이다.”라는 기존에 갖고 있었던 생각은 옳지 않다는 결론에 도달했다.

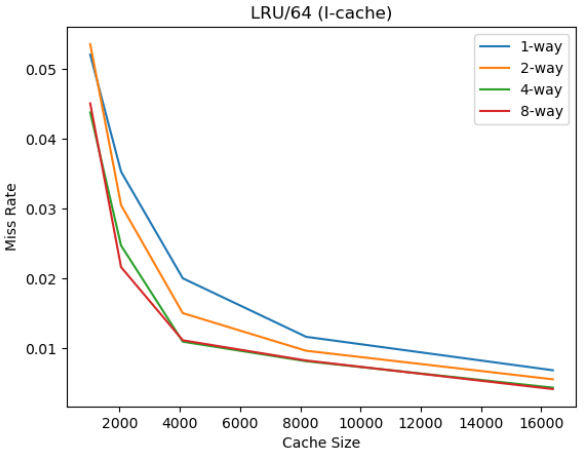
다른 요인들도 살펴보자. 전반적으로 캐시 사이즈가 커지고 연관도가 높아질수록 miss rate 가 줄어드는 것을 확인할 수 있다. 다만 I-cache 와 D-cache 가 다른 양상을 띄는 요인이 있는데, 바로 블록 사이즈이다.

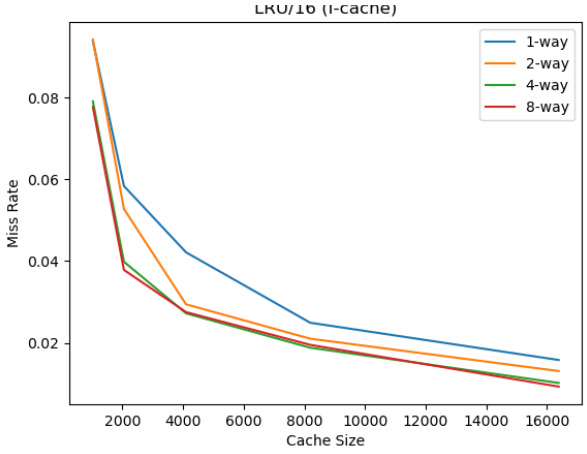
I-cache 의 경우 블록 사이즈가 큰 캐시의 miss rate 가 대체적으로 낮게 측정되었지만, D-cache 의 경우는 캐시 사이즈가 8K 가 되기까지 블록 사이즈가 큰 캐시의 miss rate 와 d_write 횟수가 더 높게 나오는 것을 확인할 수 있었다. Trace2-out.txt 파일로 도출한 그래프에서도 이와 같은 양상이 확인된다. 이와 같은 결과가 도출되는 이유가 무엇일까?

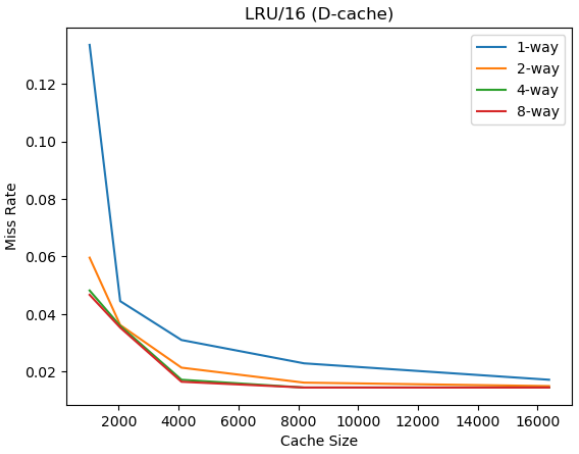
블록 사이즈가 커질수록 한 번에 가져올 수 있는 데이터 양이 증가하므로, 인접한 메모리 주소의 데이터를 더 많이 가져올 수 있게 된다. 이는 공간 지역성을 더 잘 활용할 수 있게 되어 cache miss 의 발생을 줄일 수 있다. 그러나 데이터 캐시 블록의 크기가 지나치게 커지면, 캐시에 저장된 데이터의 유효성을 유지하기 위해 데이터의 갱신이 필요한 경우가 늘어나게 된다. 이로 인해 쓰기 연산이 증가하고, 데이터 갱신을 위한 disk 접근이 필요 해져서 cache miss 가 발생할 수 있다.

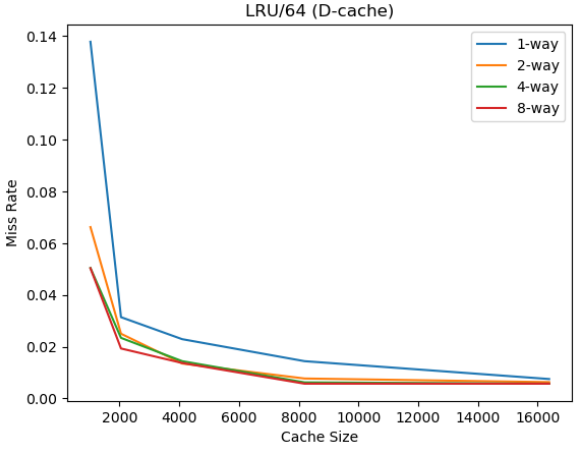
따라서, 캐시 사이즈에 따른 캐시 블록의 크기를 적절하게 설정하는 것이 중요하다는 결론에 도달했다.

trace2-out graph









trace1-out.txt

cache size	block size	associative	d-miss rate	i-miss rate	mem write
1024	16	1	0.1335	0.0937	1177
1024	16	2	0.0596	0.0941	531
1024	16	4	0.0482	0.079	398
1024	16	8	0.0467	0.0776	379
1024	64	1	0.1378	0.0521	1305
1024	64	2	0.0662	0.0536	674
1024	64	4	0.0504	0.0438	489
1024	64	8	0.0503	0.0451	430
2048	16	1	0.0445	0.0583	315
2048	16	2	0.0362	0.0527	253
2048	16	4	0.0359	0.0398	253
2048	16	8	0.0353	0.0378	241
2048	64	1	0.0314	0.0353	220
2048	64	2	0.025	0.0305	183
2048	64	4	0.0234	0.0247	170
2048	64	8	0.0193	0.0216	126
4096	16	1	0.031	0.0421	152
4096	16	2	0.0214	0.0294	87
4096	16	4	0.0172	0.0272	56
4096	16	8	0.0165	0.0275	49
4096	64	1	0.0229	0.02	151
4096	64	2	0.0135	0.015	69
4096	64	4	0.0144	0.0109	78
4096	64	8	0.0137	0.0111	81
8192	16	1	0.0229	0.0249	77
8192	16	2	0.0162	0.021	19
8192	16	4	0.0145	0.0188	4
8192	16	8	0.0145	0.0195	3
8192	64	1	0.0144	0.0116	79
8192	64	2	0.0077	0.0096	19
8192	64	4	0.0062	0.0081	11
8192	64	8	0.0057	0.0082	3
16384	16	1	0.0172	0.0158	36
16384	16	2	0.015	0.0131	9
16384	16	4	0.0145	0.0102	0
16384	16	8	0.0145	0.0093	0
16384	64	1	0.0075	0.0068	24
16384	64	2	0.0063	0.0055	9
16384	64	4	0.0057	0.0043	0
16384	64	8	0.0057	0.0041	0

CONCLUSION

캐시 시뮬레이터 과제를 통해 얻은 결론은 다음과 같다.

캐시 사이즈와 연관도는 miss rate 를 감소시키는 요소로 작용한다. 큰 캐시 사이즈와 높은 연관도는 miss rate 를 줄이는데 효과적이다.

연관성이 높아질수록 disk 에 접근하는 횟수가 줄어든다. 이는 연관성이 높을수록 데이터의 일관성을 유지할 수 있기 때문이다. 하지만 연관성에 따른 miss rate 를 그래프로 분석한 결과, 2-way 연관 캐시 이후에는 연관성에 따른 miss rate 의 감소가 크지 않음을 확인했다. 따라서 연관도 선택 시 가장 낮은 miss rate 를 가진 연관도를 선택하는 것이 항상 최적의 선택이라고 볼 수는 없다.

블록 사이즈는 I-cache 와 D-cache 에서 다른 영향을 미치는 것을 확인할 수 있다. I-cache 의 경우, 큰 블록 사이즈가 캐시 miss rate 를 낮추는데 도움이 되지만, D-cache 의 경우에는 캐시 사이즈에 따른 블록 사이즈가 지나치게 커지면 쓰기 연산이 증가하고 disk 접근이 필요한 cache miss 가 발생할 수 있기 때문에 적절한 블록 사이즈 설정이 필요하다.

따라서, 최적의 캐시 구조를 찾기 위해서는 캐시 사이즈, 연관도, 그리고 블록 사이즈를 적절하게 설정하는 것이 중요하며, 공간 지역성과 데이터 갱신의 특성을 종합적으로 고려해야 한다.

참고문헌

[1] Computer Organization and Design _ The Hardware/Software Interface (2013).