Project 4: Multi-level Cache Model and Performance Analysis

202011047 김승태

1. 컴파일/실행 방법, 환경

1-1. 환경 구성

환경은 Oracle VM VirtualBox에서 실행하였으며, Ubuntu를 이용하였다. 컴파일러는 gcc를 이용하였으며, 각각의 version은 다음과 같다. Ubuntu 20.04.6 LTS, gcc 11.4.0

```
seungtae@seungtae-VirtualBox:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 20.04.6 LTS
Release:        20.04
Codename:       focal
seungtae@seungtae-VirtualBox:~$ gcc --version
gcc (Ubuntu 11.4.0-2ubuntu1~20.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1-2. 컴파일과 실행 방법

실행 방법은 g++ 컴파일러를 이용하였으므로, 다음과 같다. .cpp 파일을 기본으로 열 경우에는, 컴파일을 실행시켜서 실행 프로그램을 제작 후 실행시키면 되므로, g++ -o runfile runfile.cpp 를 사용하여서 실행 파일을 만든다. 사진에 자세히 나타내었으니, 확인하면 좋을 것이다.

1. Ubuntu 20.04 환경에서 g++ 11.4.0 버전을 이용하여 다음 명령어로 컴파일 하였다. g++ -o runfile runfile.cpp

```
seungtae@seungtae-VirtualBox:~/CAassi2$ gedit runfile.cpp
seungtae@seungtae-VirtualBox:~/CAassi2$ g++ -o runfile runfile.cpp
```

2. 과제 pdf에도 나왔던 것처럼, 기본 실행을 실시한다. 과제에서 명시되었던 방법과 완전 동일한 방법으로 실행 결과를 볼 수 있게끔 하였다.

$ ./runfile <-c capacity> <-a associativity> <-b block_size> <-lru 또는 -random> < tracefile >

이런 방식으로 하면, tracefile과 c,a,b 값에 따라 tracefilename_c_a_b.out 파일을 output으로 내뱉게 된다.

2. Code flow

전체적인 흐름

1. **main 함수 실행**
    - main 함수가 실행된다.
    - 명령줄 인수를 파싱하여 캐시 설정(CAPACITY, ASSOCIATIVITY, BLOCK_SIZE, IS_LRU, TRACEFILE)을 초기화한다.
    - 트레이스 파일을 열어 읽을 준비를 한다.
2. **캐시 초기화**
    - L1 과 L2 캐시를 초기화한다.
    - CacheBlock 객체를 초기화하여 emptyBlockPointer 를 설정한다.
    - L1MemoryCache 와 L2MemoryCache 객체를 생성한다.
    - initializePointer 함수를 호출하여 L1 캐시와 L2 캐시 간의 상호 참조를 설정한다.
3. **트레이스 파일 읽기**
    - 트레이스 파일을 한 줄씩 읽는다.
    - 각 줄을 tokenize 함수로 분리하여 메모리 접근 정보를 추출한다.
    - totalOperations, readOperations, writeOperations 등의 통계를 갱신한다.
    - L1 캐시에 접근하여 handleAccess 함수를 호출한다.
4. **캐시 접근 처리 (handleAccess 함수)**
    - parseRealMemTagIndex 함수를 호출하여 메모리 주소에서 태그와 인덱스를 추출한다.
    - 해당 인덱스에서 태그를 비교하여 캐시 히트를 확인한다.
    - 히트 시:
        - 읽기 또는 쓰기 히트를 증가시키고, 마지막 접근 시간을 업데이트한다.
    - 미스 시:
        - handleMiss 함수를 호출하여 블록을 L2 캐시 또는 메모리에서 가져온다.
        - insertBlock 함수를 호출하여 블록을 L1 캐시에 삽입한다.
        - 읽기 또는 쓰기 미스를 증가시킨다.
    - 접근된 블록의 복사본을 반환한다.
5. **블록 삽입 처리 (insertBlock 함수)**
    - parseTagIndex 함수를 호출하여 블록의 태그와 인덱스를 추출한다.
    - 해당 인덱스에서 빈 슬롯을 찾아 블록을 삽입한다.
    - 빈 슬롯이 없을 경우 evictBlock 함수를 호출하여 블록을 퇴출한 후 삽입한다.
6. **블록 퇴출 처리 (evictBlock 함수)**
    - LRU 또는 랜덤 정책에 따라 퇴출할 블록을 선택한다.
    - 선택된 블록이 더티인 경우, 상위 캐시에 해당 블록을 쓰기(Write)한다.
    - 선택된 블록을 삭제하고, 빈 블록 포인터로 교체한다.

- 클린 퇴출 또는 더티 퇴출 통계를 갱신한다.

7. **결과 저장**
   - 트레이스 파일을 모두 처리한 후, 결과를 출력 파일로 저장한다.
   - L1 캐시와 L2 캐시의 설정과 각종 통계를 출력한다.
8. **자원 해제**
   - delete 를 통해 동적 할당된 L1 캐시와 L2 캐시 객체를 해제한다.
   - 프로그램을 종료한다.

2. Two Level Cache 결과 분석

이번 분석에서는 작성된 캐시 시뮬레이션 코드를 통해 캐시 크기, 연관도(associativity), 블록 크기(block size)가 변화할 때, 미스율(miss rate)이 어떻게 감소하는지 경향성을 분석하였다. 여기서 사용한 값은 L2의 읽기(read)와 쓰기(write) 미스율이다. Project4 pdf의 값을 보면, L2 Cache (read 또는 write) miss 횟수를 (read 또는 write) 횟수로 나눈 값과 비슷해 보이는데, 따라서, project 예상 결과와 비슷하게끔 전체 Cache의 miss rate를 비교할 것이다.

실험 설정

1. **캐시 크기 변화 분석**
   - 캐시 크기를 4KB에서 1024KB까지 증가시키며 그린 그래프이다.
   - 연관도(associativity)와 블록 크기는 각각 4와 32B로 고정하였다.
2. **연관도 변화 분석**
   - 연관도를 1에서 16까지 증가시키며 그린 그래프이다.
   - 캐시 크기와 블록 크기는 각각 256KB와 32B로 고정하였다.
3. **블록 크기 변화 분석**
   - 블록 크기를 16에서 128까지 변화시키며 그린 그래프이다.
   - 캐시 크기와 연관도는 각각 256KB와 4로 고정하였다.

분석에 사용된 tracefile

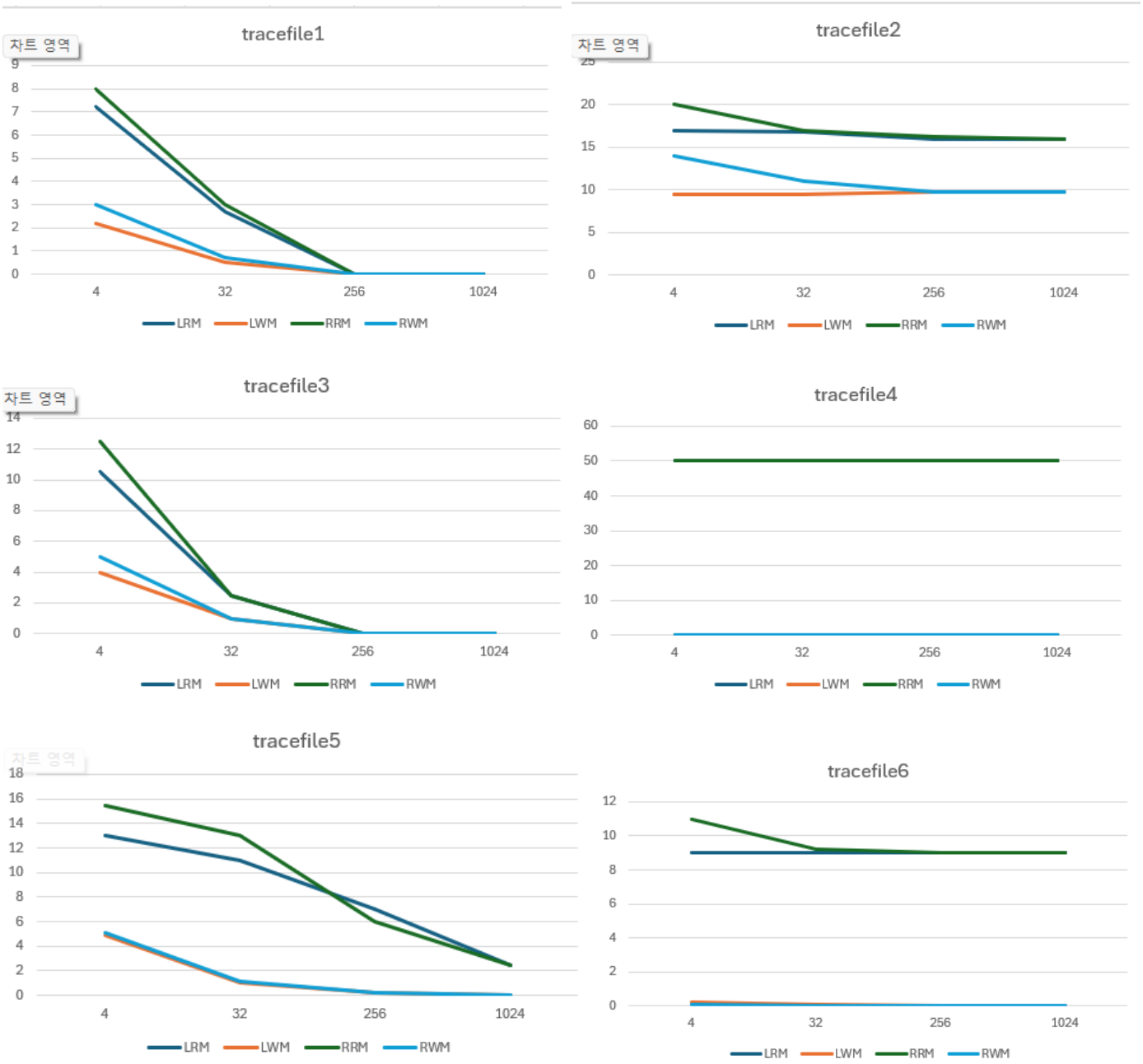
- **tracefile1**: 400_perlbench.out
- **tracefile2**: 450_soplex.out
- **tracefile3**: 453_povray.out
- **tracefile4**: 462_libquantum.out
- **tracefile5**: 473_astar.out
- **tracefile6**: 483_xalancbmk.out

그래프 설명

- **LRU READ MISS RATE (LRM)**: LRU 정책을 사용할 때의 읽기 미스율이다.
- **LRU WRITE MISS RATE (LWM)**: LRU 정책을 사용할 때의 쓰기 미스율이다.

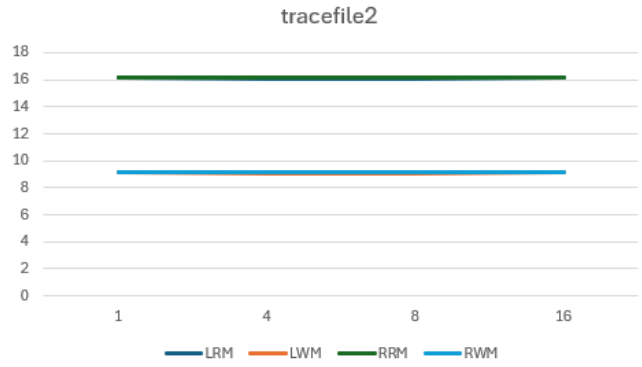
- **RANDOM READ MISS RATE (RRM)**: 랜덤 정책을 사용할 때의 읽기 미스율이다.
- **RANDOM WRITE MISS RATE (RWM)**: 랜덤 정책을 사용할 때의 쓰기 미스율이다.

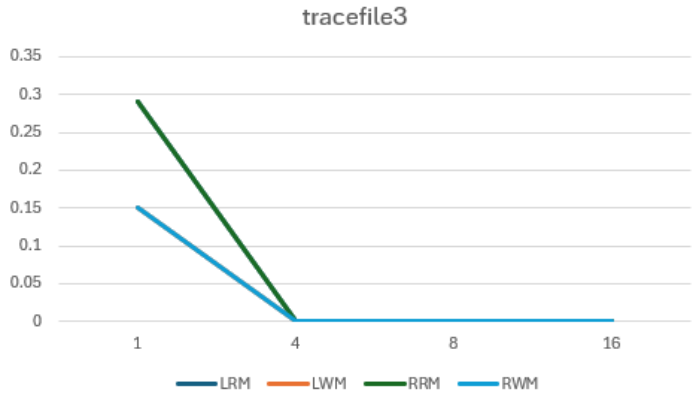
1.캐시 크기 변화 / 가로 – Cache Capacity(KB), 세로 – miss rate(%)

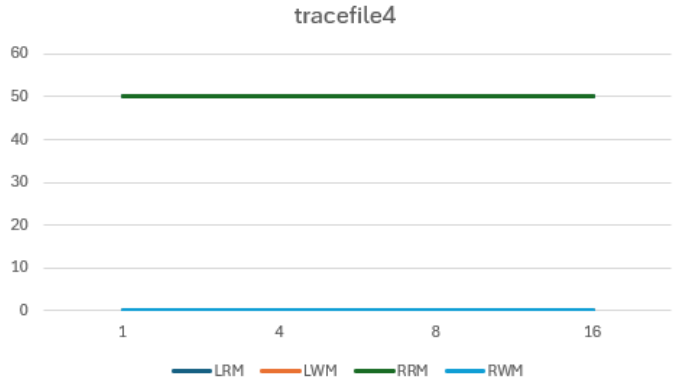


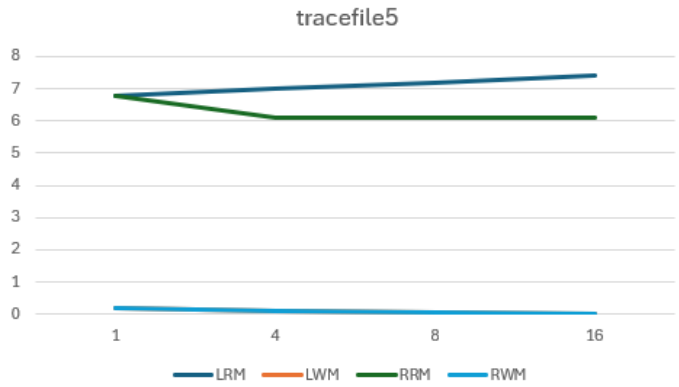
위 결과를 보면, 거의 모든 그래프 (4,6 예외사항 제외) 에서 LRU 의 miss rate가 read, write 둘 다에서 낮은 것을 알 수 있다. 또한, 거의 모든 그래프가 우하향 그래프로, capacity가 늘어남에 따라서, miss rate가 줄어드는 결과를 보이고 있다. 이는 당연한 결과다. Size가 늘어나게 되면, block을 추가적으로 저장이 가능하니, 사이즈가 커질수록 당연히 cache miss가 줄어들 것이다. 그렇다면, 2,4,6, 특히 tracefile4에서는 왜 이러한 그래프 형상이 그려질까? 이는, cache 사이즈가 거의 부족하지 않은, 극한적 spatial locality의 상황이라고 생각하면 좋을 것 같다. 이는 capacity가 꽤나 작아도, miss가 거의 그대로이기 때문이다. 즉, capacity를 늘린다고 해서 miss가 늘어나지는 않는다.

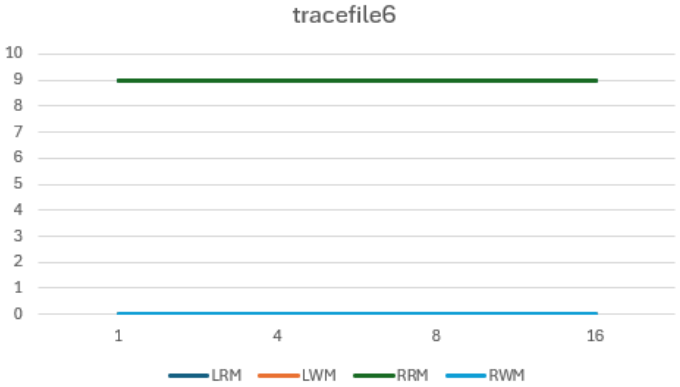
2. 연관도 변화 / 가로 – associativity, 세로 – miss rate(%)





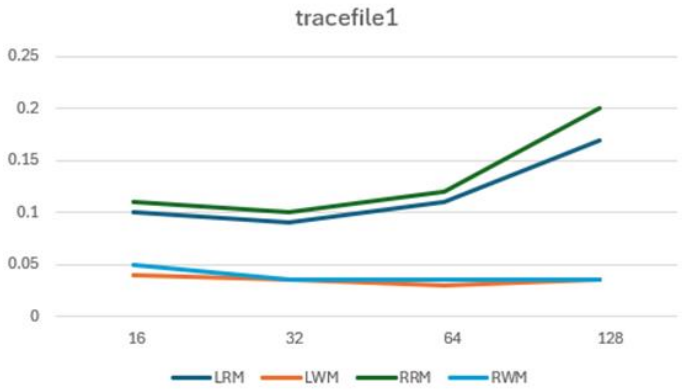


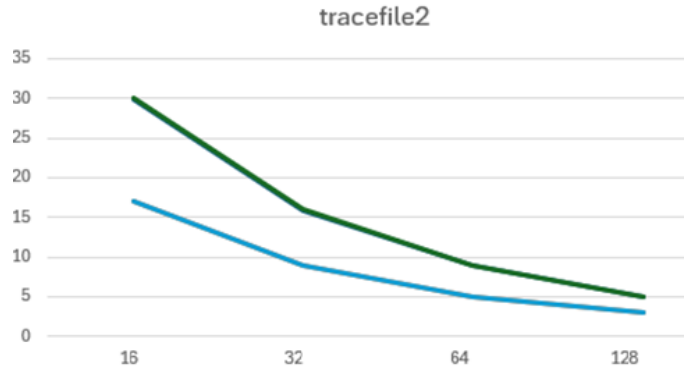


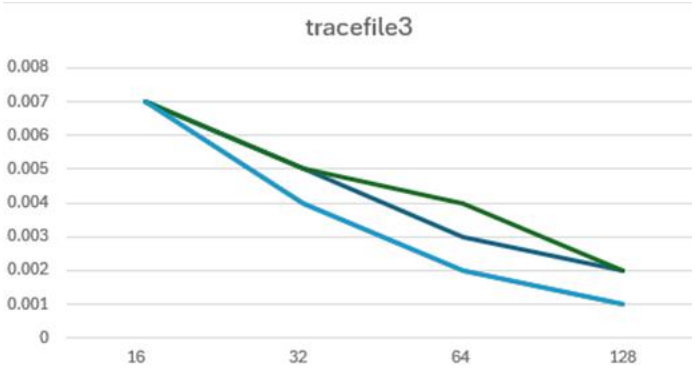


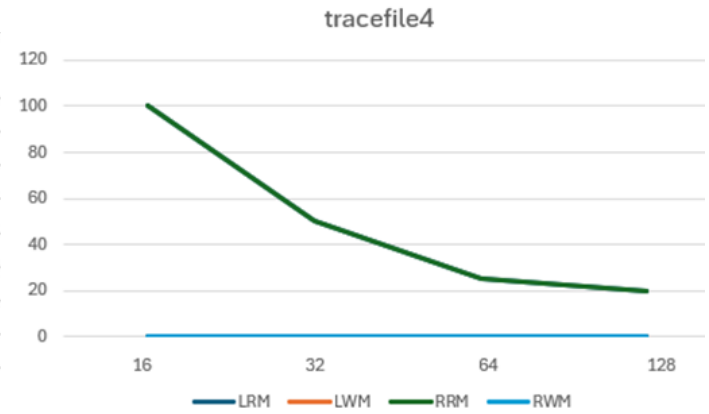
이번에는 tracefile1, tracefile3은 줄어드는 경향, 2,4,6은 그대로, 5는 조금 이상한 경향을 보인다. 이것은 1,3을 제외한 나머지 tracefile은, block 번호가 다른, 그리고 set도 다른 data가 접근이 많이 되어서, set, 즉 associativity가 늘어났다고 하더라도, 별로 영향을 받지 않는 것이다. 그리고 4,6은 위의 결과에서도, spatial locality가 높을 것이라고 판명받았으므로, 거의 여기에서도 set 자체가 필요가 없을 것이다. 또한, 여기에서는 capacity가 충분하므로, 생각보다 LRU, Random에서 큰 결과값 차이가 없는 것을 볼 수 있다. (결과 1과 비교한다면 훨씬 보기 쉬울 것이다.)

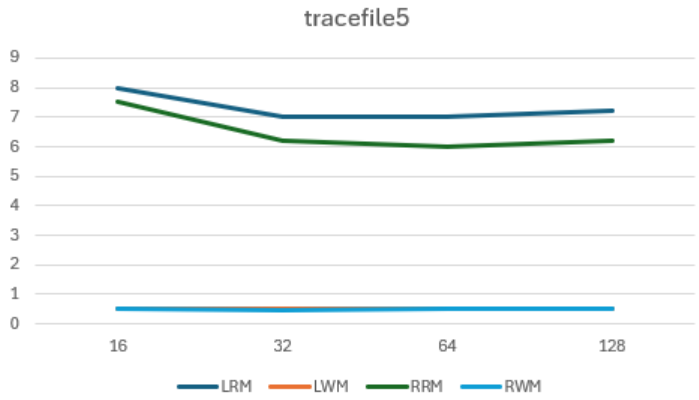
3. 연관도 변화 / 가로 - Cache Block Size(Byte), 세로 – miss rate(%)

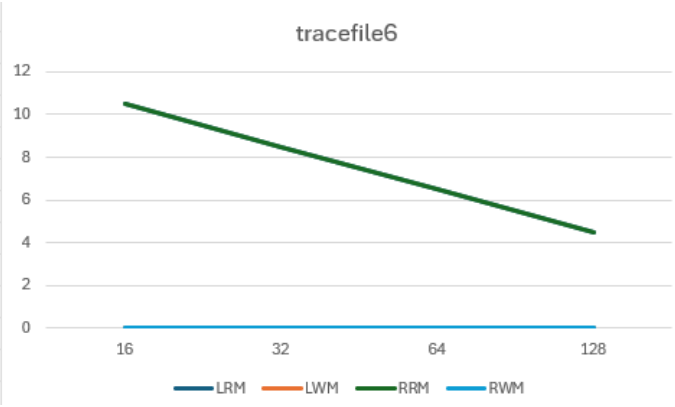












여기에서는 딱 보이는 것이 2,4,6의 변화인데, 이는 굉장히 뚜렷한 우하향 경향을 띈다는 것을 알 수 있다. 이는, spatial locality가 높았으므로, block size가 증가한다면, 당연히 block size가 커지면 miss rate가 줄어들 것이라는 것이다. 하지만, 1,5에서는 증가하는듯한 모양새도 보였는데, 이는, 이들은 spatial locality가 거의 없거나 하여서 그런 것으로 고려된다. 단위가 크게 되면, sequential locality가 주되거나, locality가 떨어지는 상황에서는 낮은 성능을 보이게 되는 것이다.