**2018212031 이원찬 캐시 시뮬레이터 실험 보고서**

파이썬으로 캐시 시뮬레이터를 구현하였습니다.

1. **파이썬을 선택한 이유**

* 데이터를 분석하고 마음대로 변형, 수정이 쉬운 언어는 파이썬이라고 생각했습니다.
* 실제로 시뮬레이터를 구현하면서 C언어, JAVA보다 더 쉽게 구현한 것 같습니다.
* 단점은 시간이 오래걸립니다. 실제로 파일양이 많은 gcc.trace파일이나 swim.trace 파일은 2~5초정도 걸렸습니다.

1. **실행코드 입력과 출력**

* 실행인자로 set수, block수, byte사이즈, write\_type, allocate\_type, Out\_type, filename을 받습니다.
* 실행시간이 오래걸려 set,block은 [8,16,32] 세가지 중에서 실행 시켰고

Bytesize는 [4,8,16] 세가지 중에서 실험 하였습니다.

* 출력에 Total load, Total store, Load hit, Load Misses, Store hits, Store misses, Total cycles, Total hits, Performance(본인 설정값) 가 나옵니다.

1. **Perfomance의 정의**

* 캐시시뮬레이터의 성능을 표현하는 Performance의 정의는  
  (Total\_Hits/Total\_Cycle)x1000으로 정의하였습니다.
* Hits가 많으면 성능이 좋다고 표현하고,  
  Cycle(주기)가 많으면 성능이 떨어짐을 이용한 비례/반비례 관계입니다.
* 사이클의 숫자가 hits보다 보통 크기 때문에 1000을 곱해 자릿수를 조정하였습니다.
* 소수 두 자리수 까지 표현하였습니다.

1. **Flow.py파일**

* 주석으로 캐시 시뮬레이션이 흐름을 나타낸 파일입니다.

1. **Cach\_Simulator.py파일**

* 본격적으로 캐시 시뮬레이터를 구현한 파일입니다.
* 실행인자로 위에서 말한 값들을 받고, 출력으로 각 출력값들을 출력합니다.

1. **Simulation\_Function.py 파일**

* 본인이 for문으로 쉽게 캐시 실험을 하기 위해 만들 파일입니다.

1. **구현 설명**
2. 파일내용을 읽어 명령어 한줄씩 for문을 돌며 실행되도록 구현하였습니다.
3. 캐시는 set, block, byte수를 읽어 3차원 리스트로 구현하였습니다.
4. 입력된 주소를 2진수로 변환후 Set,block 사이즈로 슬라이싱하여 각각

10진수로 변경하였습니다.

1. 예를 들어 block수가 8이라면 2^3이므로 주소의 2진수중 3자리만 block offset으로 사용하였다. Set는 block다음 왼쪽 주소를 이용
2. 직접 주소가 들어갈 3차 리스트는 빈리스트로 생성하였습니다.  
   (리스트의 길이로 cache의 자리가 있는지 없는지 판단하기 때문)
3. 빈리스트에 주소가 있다면 hit하여 분기 하였고, 주소가 없다면 miss로 분기 하였습니다.
4. Load 명령어와 store 명령어를 구분하여 Total\_load\_store값을 증가시켰습니다.
5. Hit 이며, store명령일 때 write\_through\_back에 따라 분기하였습니다.
   * White\_through 일때는 cycle을 +100하였습니다.
   * White\_back 일때는 Dirty리스트에 히트된 주소를 추가합니다.  
     (나중에 캐시에서 추방될 때 사이클을 올려야하기 때문)
6. Out\_type이 lru일 때 히트된 주소를 리스트 뒤로 옮겨 최신화 시켜 주었습니다.
7. Miss이며 no\_write\_allocate가 아니고 stroe명령이 아니라면 캐시 저장을 하기 때문에 분기 하였습니다.
8. 만약 miss된 리스트의 길이가 byte 사이즈와 비교했을 때 크거나 같다면 자리가 없고, 작다면 자리가 있음으로 분기하였습니다.
9. 자리가 있다면 리스트 끝에 주소를 추가 하였습니다.
10. 자리가 없다면 Out\_type에서 두가지로 분기하였습니다.
11. Lru 또는 fifo 는 miss된 리스트가 이미 최신화가 되어있는 상태이므로  
    마지막 요소만 삭제하는 것으로 구현했다.
12. Random 경우 리스트 길이보다 작은 자연수를 랜덤으로 생성해 랜덤인덱스를 삭제하는 것으로 구현하였다.
13. 만약 삭제되는 주소가 Dirty리스트에 있다면 사이클을 증가 시켰습니다.
14. Allocate\_type이 no\_write\_allocate인 경우 캐시에 저장하지 않는 것으로 구현하였습니다.
15. **실험 결과**
16. 정확한 실험 결과는 깃허브 링크 파일중 캐시\_결과.pdf 파일에 나와있습니다.
17. 일단 set, block, byte 사이즈가 클수록 성능이 높게 나타났습니다.
18. 그렇다면 set, block, byte 사이즈를 고정시키고 다른 인자를 변경 하며 성능을 비교해봤습니다.
19. Defalut  
    write\_type = write\_through

Allocate\_type = write\_allocate

Out\_type = fifo

1. Out\_type이 lru,fifo,random 각각 비교했을 때 lru일 때 성능이 좋은 것으로 결과가 나왔습니다.
2. 다음으로 Out\_type을 lru로 고정하고 write\_type을 변경하여 실험하였습니다.
3. Write\_through 와 write\_back 의 경우 성능이 2~3배정도 차이로 write\_back의 성능이 높은 것으로 결과가 나타났습니다.
4. Allocate\_type을 비교하려 했을 때 write\_back과 no-write-allocate는 존재하지 않으니 write\_through 와 no-write-allocate로 성능을 비교하였습니다
5. 마지막으로 성능이 가장 좋은 캐쉬는 set,block,byte사이즈가 가장 크고  
   write-back, write-allocate,lru일 때 성능이 가장 좋은 것으로 결과가 나왔습니다.

직접 실행시킨 출력 화면

텍스트이(가) 표시된 사진

자동 생성된 설명

실행한것중 가장 성능이 좋았던 조합

텍스트이(가) 표시된 사진

자동 생성된 설명