1. Introduction

본 과제의 목적은 기존에 매직 메모리 기반으로 구현했던 CPU보다 실제 환경과 더 유사하게, 접근에 일정 사이클이 소요되는 메모리를 도입하고, 쓰루풋 향상을 위해 캐시를 사용하는 CPU를 구현하는 데 있다. 이를 통해 컴퓨터 구조에서의 메모리 계층 구조와 캐시의 중요성, 그리고 프로그램 성능에 미치는 영향을 직접적으로 경험해보고자 한다.

2. Design

우리 팀은 기본적으로 4개의 set과 4개의 way를 갖는, 총 256bytes 크기의 associative cache를 설계하였다. set과 way의 수는 파라미터로 쉽게 조정할 수 있어, set당 way가 1개인 direct-mapped cache로도 간단히 변경 가능하다. 단, set이 1개뿐이고 모든 way가 한 set에 포함된 fully-associative cache는 파라미터 적용 시 set index 처리 문제로 인해 지원하지 않는다. Write 정책은 write-back과 write-allocate policy를 택했으며, replacement policy로는 LRU(Least Recently Used)를 사용하였다.

캐시의 주요 동작 방식은 다음과 같다.
Write-back: 캐시 라인이 수정될 경우 dirty bit를 설정하고, victim이 될 때만 메모리에 기록한다.
Write-allocate: miss가 발생한 write 요청에 대해 우선 메모리에서 라인을 불러온 뒤 캐시에서 write를 수행한다.
LRU Replacement: 가장 최근에 사용되지 않은 라인을 victim으로 선정한다.

3. Implementation

LRU policy의 구현을 위해 cache의 각 way마다 lru 비트를 추가하였다. 어떤 way를 실제로 접근해 이용할 경우 해당 set을 순회하며, lru 값이 접근한 way의 lru 값보다 낮다면 1 증가시키고, 그렇지 않다면 가만히 둔다. 또한 접근한 way의 lru 값은 0으로 설정한다. 이렇게 하면 항상 lru 값이 3인 way가 가장 마지막에 사용한 way가 된다. 이는 모든 way가 valid 할 경우이고, 만약 valid 하지 않은 way가 있다면 해당 way를 우선적으로 victim으로 이용한다.

Cache의 동작은 4개의 state를 가지는 FSM으로 구현하였다.
IDLE: 입력 신호를 대기하며, hit/miss를 판정한다.
WRITEBACK: victim line이 dirty하면 메모리에 writeback을 수행한다.
MEMREAD: 메모리로부터 새로운 line을 읽어온다.
REFILL/COMPLETE: 읽어온 line을 캐시에 적재하고, 필요한 경우 write 연산을 반영한다.

캐시 라인은 데이터, 태그, valid/dirty 비트, lru 정보를 갖는다. 캐시 miss 발생 시 victim way를 선정하여 필요시 writeback 후, 메모리에서 데이터를 읽어온다. 이후 write 요청이라면 즉시 데이터 갱신 및 dirty 비트 설정, read 요청이라면 cache에서 해당 데이터를 반환한다.

4. Discussion

캐시를 4-way set associative로 구성했을 때, naive_matmul 테스트 케이스에서 약 0.784의 hit rate, opt_matmul 테스트 케이스에서 약 0.904의 hit rate를 기록했다. 이에 따라 naive_matmul은 38,958 사이클, opt_matmul은 25,398 사이클이 소요되었다. 이 결과는 적절한 set-associativity를 적용하면 naive한 방식보다 최적화된 방식에서 hit rate가 높고, 사이클 수도 더 적게 필요함을 보여준다.

캐시를 16개의 set과 1개의 way를 가진 direct-mapped cache로 변경했을 경우, naive_matmul에서는 0.675의 hit rate와 72,821 사이클, opt_matmul에서는 0.642의 hit rate와 78,155 사이클이 소요되었다. 이 경우에는 특이하게 opt 테스트 케이스의 hit rate가 더 낮아 오히려 사이클 수가 더 많이 소요되는 현상이 나타났다.

2개의 set과 8개의 way로 구성할 경우 naive는 0.742의 hit rate, 44,417 사이클, opt는 0.904의 hit rate, 25,398 사이클이 소요되었다. 8개의 set과 2개의 way로 구성하면 naive는 0.755의 hit rate, 52,862 사이클, opt는 0.746의 hit rate, 55,640 사이클이 소요되었다.

Naive matmul 테스트는 두 개의 8x8 행렬 곱을 수행하며, opt matmul 테스트는 동일한 8x8 행렬 곱이지만 4x4 크기의 subblock 단위로 행렬 곱셈을 수행한다. 적절한 set-associativity를 가진 cache에서 naive하게 행렬 곱셈을 수행하면 cache conflict miss가 많이 발생하지만, subblock 단위로 계산하면 캐시에 한번 적재된 block 내에서 연산이 이루어져 conflict miss가 줄고, hit rate가 높아지는 현상을 확인할 수 있었다.

5. Conclusion

본 과제를 통해 캐시의 동작 원리를 학습하고, 실제로 캐시를 구현해봄으로써 그 원리를 보다 깊게 이해할 수 있었다. 또한 cache의 set-associativity를 조정하며 naive 행렬 곱셈과 subblock 단위 행렬 곱셈에서의 hit rate와 총 사이클을 비교·분석함으로써, 캐시 친화적인 프로그램 설계의 중요성과 캐시를 효과적으로 활용하여 쓰루풋을 높이는 방법에 대해 고민할 수 있는 계기가 되었다.

추가로, fully-associative와 direct-mapped 캐시의 장단점, LRU 정책의 구현 난이도와 실제 효과, 파라미터화된 캐시 설계의 유연성 등 다양한 캐시 구조에 대한 이해를 넓힐 수 있었고, 실험을 통해 메모리 계층 구조와 소프트웨어 접근 패턴이 시스템 성능에 미치는 영향을 체감할 수 있었다.