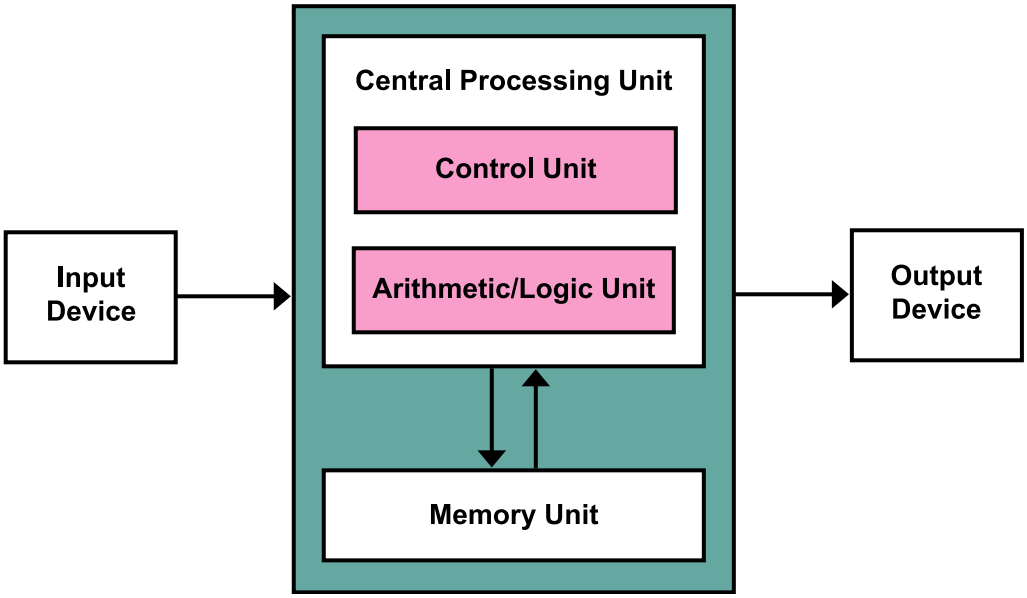
데이터 지향적 디자인[1]

Go 프로그래밍 언어의 기본 데이터 구조인 배열, 슬라이스, 그리고 맵을 살펴보기 전에 데이타 지향적 디자인이라는 개념에 대해 먼저 알아 보자

데이터 지향적 디자인은 CPU 캐쉬의 사용을 극대화하는 방향으로 프로그램의 디자인을 최적화하여서 고성능을 성취하는 것을 말한다.

오랜 세월동안 CPU의 발전이 클럭 사이클을 위주로 진행되는 동안 전통적인 컴퓨터 아키텍쳐인 폰 노이만 아키텍쳐는 데이터 처리에 있어 심각한 단점을 가지고 있다는 것이 증명되었다. 폰 노이만 병목현상(Von Neumann Bottleneck)이라고 부르는 이 문제는 CPU의 클럭 사이클이 짧아지면서 데이터 처리 속도는 높아지는 반면 메모리를 접근하기 위해 사용된 데이터 버스의 공유는 제자리 걸음을 하고 있어 데이터 처리에 많은 지연이 생기는 현상을 말한다.

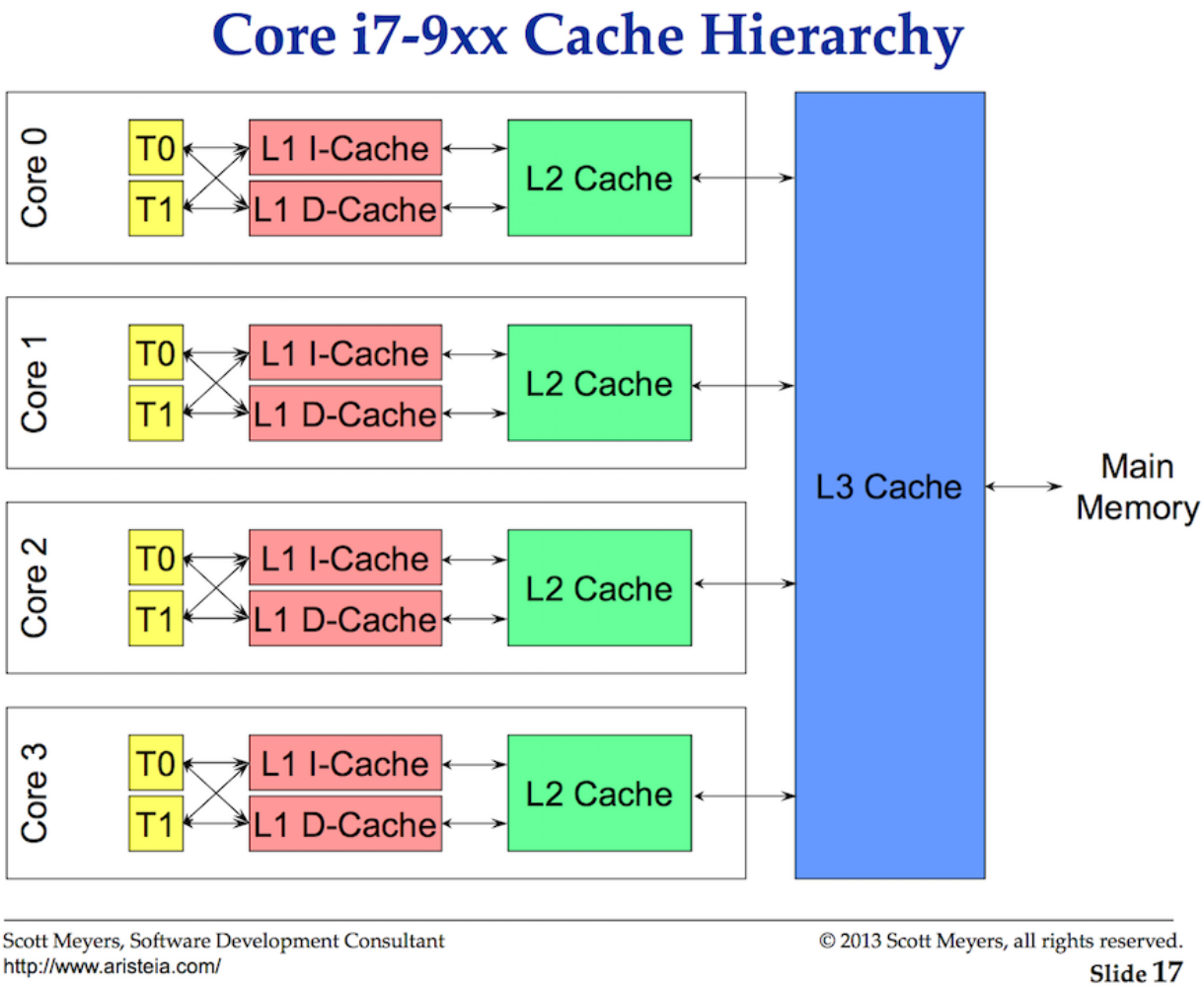


이런 문제점들은 극복하려는 시도로 널리 채택된 것들 중에는 CPU와 메모리 사이에 cache를 두고 반복되는 메모리의 접근을 제거하는 식의 temporal locality를 적용한다던지, 일정한 패턴으로 인접한 메모리들에서 데이터를 읽어들이는 경우, 예측할 수 데이터 접근을 cache에 미리 담아 두는 식의 spatial locality를 데이터 처리의 최적화로 사용할 수도 있다. 또한 명령어 루프내 선택할 수 있는 가능성이 작은 수로 제한되어 있는 경우 브랜칭에 대한 로켈리티(branch locality)가 발생하여 곧 접근할 필요가 생길 메모리나 코드를 복사해 캐쉬에 가져다 놓는 pre-fetching이 사용될 수도 있다. 결국 컴퓨터 프로그램의 고성능을 결정하는 요소로 예측가능한 형태의 데이터 처리가 얼마나 로켈리티를 발생시키느냐가 중요한 요소로 부각된다. [2]

- Temporal Locality
  - 같은 데이터를 짧은 시간 간격으로 반복적으로 참조하는 경우
- Spatial Locality
  - 메모리에서 서로 가까이 있는 데이터를 참조하는 경우
- Sequential Locality
  - 메모리에서 연속적으로 나열되어 있는 데이터를 참조하는 경우
- Branch Locality
  - 실행 가능한 브랜칭의 수가 적은 경우

CPU 캐쉬

- CPU 캐쉬의 작동원리는 메인 메모리로 부터 캐쉬 라인 (혹은 캐쉬 블록)이라고 부르는 단위의 데이터를 캐쉬에 저장하는 것이다.
- 캐쉬라인은 하드웨어에 따라 다를 수 있지만 보통 32 혹은 64 바이트이다.
- CPU 코어들은 메인 메모리를 직접 엑세스하지 않고 로컬 캐쉬들을 엑세스한다.
- 데이터와 명령어들이 모두 캐쉬에 저장된다.
- 새로운 캐쉬라인들이 캐쉬들에 저장될때 L1->L2->L3의 방향으로 기존의 캐쉬라인들이 이동한다.
- 하드웨어는 캐쉬라인을 따라서 데이터와 명령어들을 선형적으로 처리하기를 좋아한다.
- 메인 메모리는 비교적 빠르고 값 싼 메모리로 만들어져 있고 캐쉬는 매우 빠르고 값 비싼 메모리로 만들어져 있다.
- 메인 메모리를 접근하는 것은 매우 느리고 CPU의 실행지연(stall)을 가져올 수 있기 때문에 캐쉬가 필요하다.
    - 메인 메모리에서 1 바이트를 읽어도 캐쉬라인 전체를 읽고 캐쉬에 저장해야 한다.
    - 캐쉬라인에 1 바이트를 쓰는 경우에도 캐쉬라인 전체가 쓰기에 사용되어야 한다.
- Small = Fast
    - 콤팩트하고, 로컬화가 이루어진 코드로 캐쉬의 크기에 맞는 경우가 가장 빠르다.
    - 캐쉬의 크기에 잘 맞는 데이터 구조가 가장 빠르다.
    - 캐쉬에 저장된 데이터를 처리하는 경우가 가장 빠르다.
- 예상 가능한 데이터의 접근 패턴은 중요하다.
    - 언제든 실용적이라면, 배열을 이용한 선형적인 데이터 처리를 채택하라
    - 일정한 메모리의 접근 패턴을 제공하라
    - 하드웨어는 요구되는 메모리에 대한 더 정확한 예측을 할 수 있다.
- 캐쉬 미스는 TLB(Translation lookahead buffers) 캐쉬 미스로 귀결될 수도 있다.
    - 가상 주소에서 실제 물리적 주소를 가리키는 번역을 캐쉬
    - OS가 어디에 메모리가 있는지 알려주기 기다리게 됨.



캐쉬 미스로 인해 지불해야 하는 클럭사이클

캐쉬 미스는 프로그램에서 처리되는 데이터가 캐쉬에서 발견되지 않을때 다른 레벨의 캐쉬나 메인 메모리에서 데이터를 읽어들여 캐쉬에 저장해야 하는 상태를 말한다.

```
3GHz(3 clock cycles/ns) * 4 instructions per cycle = 12 instructions per
ns!

L1 - 64KB Cache (Per Core)
        32KB I-Cache
        32KB D-Cache
        2 HW Threads
        4 cycles of latency
        Stalls for 16 instructions or 1.3 ns

L2 - 256KB Cache (Per Core)
        Holds both Instructions and Data
        2 HW Threads
        11 cycles of latency
        Stalls for 44 instructions or 3.6 ns

L3 - 8MB Cache
        Holds both Instructions and Data
        Shared across all 4 cores
        8 HW Threads
        39 cycles of latency
        Stalls for 156 instructions or 13 ns

Main Memory
        107 cycle of latency
        Stalled for 428 instructions or 35.6 ns
        27 times slower!
```

Location	Latency in Clockcycles	Stalls
레지스터	0	0 ns
L1 캐쉬	4	1.3 ns
L2 캐쉬	11	3.6 ns
L3 캐쉬	39	13 ns
메인 메로리	**107**	35.6 ns

CPU 캐쉬 데모

`/examples/caching` 패키지의 코드를 통해 CPU 캐쉬 사용에 대한 3가지 벤치마킹을 한다.

1. 어떤 데이타 구조에 따라 성능에 차이를 보인다.
    - `BenchmarkLinkListTraverse`의 경우 `linked list`로 400만개(4194304)의 값을 연결한다. 값들의 연결이 포인터에 의해 구성되어 있어 연속적인 메모리의 사용을 하지 않은 이유로 성능의 불이익이 있다.

2. 같은 데이타 구조라도 어떻게 사용하느냐에 따라 성능의 차이를 보인다. (Predictable access patterns matter.)
    - `BenchmarkColumnTraverse`의 경우 2048마다 나타나는 값을 읽어 들이고 있다. 이 말은 각 데이터 엑세스마다 캐쉬 미스 (cache miss)가 일어나 메인 메모리에서 값을 다시 가져와야 하기 때문에 성능에 불이익이 생긴 것이다.
3. `BenchmarkRowTraverse`의 경우는 서로 바로 연결되어 있는 캐쉬에서 값을 읽어들이고 매우 규칙적이고 예상가능한 데이터의 엑세스를 하고 있기 때문에 CPU 캐쉬의 사용이 극대화된 예이다.

```
$ go test -run none -bench . -benchtime 3s
Elements in the link list 4194304
Elements in the matrix 4194304
goos: darwin
goarch: amd64
pkg: gitlab.com/jhonghee/gostudy/examples/caching
BenchmarkLinkListTraverse-4         1000           5941196 ns/op
BenchmarkColumnTraverse-4            300          17323137 ns/op
BenchmarkRowTraverse-4              1000           4208087 ns/op
PASS
ok      gitlab.com/jhonghee/gostudy/examples/caching    19.938s
```

참고할 소스코드

- examples/caching/caching.go
- examples/caching/caching_test.go

앱을 개발하면서 CPU 캐쉬를 신경써야 하나?

실제로 봉착하는 성능의 문제는 다음에 열거된 바와 같이 항상 프로그램의 실행에 국한된 것은 아니다.

- 네트워킹
- 데이터 베이스 혹은 외부의 서비스
- I/O
- 가비지 컬렉션
- 부족한 병렬처리

그럼에도 불구하고 만약 해결하고자 하는 성능의 문제들이 순수히 코드의 실행에 대한 것이라면 CPU 캐쉬가 동작하는 원리를 숙고하고 있을 때 최적화를 진행할 수 있다.

Go가 제공하는 기본적인 데이터 구조

- CPU 캐쉬의 효율적인 이용을 자연스럽게 유도하는 데이터 구조는 당연히 배열(Array)이다.
- 배열(Array)는 하드웨어에 최적화된 데이타 구조이다. 하지만 프로그래머의 생산성이나 개발의 효율성을 고려하면 사용하기 편한 데이터 구조는 아니다.
- 슬라이스는 배열을 내부적으로 사용하며 순차적인 데이터에 대해 좀 더 일반적이고, 파워풀하고, 편리한 인터페이스를 제공한다. 슬라이스는 개발자에게 최적화된 데이터 구조이다.

참고 자료

1. Data Oriented Design

2. A curated list of awesome data oriented design resources
3. Structure of arrays
4. Data-Oriented Demo: SOA, composition
5. Locality of reference
6. Caching In: Understand, Measure, and Use Your CPU Cahce More Effectively