Cache

메모리 시스템즈를 아무리 빠르게 하더라도, CPU 가 항상 더 빠르고, 절대 메모리가 CPU 속도를 따라 잡지 못한다. 메모리 성능은 항상 전체 시스템 성능을 느려지게 한다. 소스-동기 클로킹(source-synchronous clocking), 8-level prefetch buffers 같은 뛰어난 엔지니어링이 있다고 해도, CPU 는 항상 메모리가 제공하는 것보다 더 빠르게 데이터를 원한다. 지난 30 년 동안에 걸쳐 메모리 스피드가 굉장히 인상적으로 빨라지기는 했지만, 시스템 메모리 속도는 CPU 와 그의 data 사이에 전반적인 상호작용(interaction)의 속도를 올리는 주요 수단은 아니다. 주요 수단(방식)은 아마도 항상 앞으로도 그렇겠지만, data caching 이 될 거다.

Data cache 는 CPU 와 system memory 사이에 있는 빠른 메모리 블락이다. cashing 의 장점은 cache memory 는 system memory 보다 더 빠르다는 것이다. CPU 가 처음으로 메모리로부터 하나의 데이터 블락을 읽고나면, 그 데이터는 data cache 에 있게 된다. CPU 가 이후에 메모리로부터 뭔가를 읽어야 할 필요가 생길때에는 CPU 는 먼저 필요한 것이 이미 cache 에 있는지 확인을 한다. 거기에 있다면, cache hit(캐쉬 적중)이 된 거다. CPU 는 그리고 나서 그 캐쉬로부터 데이터를 얻게 된다. CPU 가 필요로 하는 데이터가 cache 에 없다면 cache miss 가 발생한 것이다. 요청한 데이터는 메모리로부터 캐쉬로 이동하게 되고 그리고 나서 CPU 까지 이동한다(방금 fetch 한 데이터가 곧 다시 필요할 것이라는 기대를 가지고).

Locality of Reference (참조 지역성)

CPU 는 자신이 필요로 하는 데이터가 이미 캐쉬에 있다는 것을 얼마나 자주 발견하게 될까? 답이 굉장히 놀랍다: CPU 는 필요로 하는 데이터를 거의 항상 캐쉬에서 발견한다. 컴퓨터 사이언스에서는 “locality of reference”라고 불리는 일반원칙이 하나 있다. 이것은 컴퓨터 오퍼레이션들은 함께 모여있는 경향이 있다는 것이다. 참조지역성은 3 가지 면이 있다.

- 지금 access 되는 데이터는 아마도 가까운 미래에 금방 다시 access 가 될 거다(시간적 지역성)
- 짧은 시간동안, 데이터 엑세스(쓰기/읽기)는 메모리의 동일한 일반 영역에 모여 있는 경향이 있다.(공간적 지역성)

- 메모리 위치(locations)는 시퀀스 순서(순차로) 읽어지거나 쓰여지는 경향이 있다.

한마디로 간단히 말해서, 컴퓨터가 특정 태스크를 실행할 때, 컴퓨터의 메모리 접근들은 여기저기 흩어져 있지가 않다. 그 접근들(accesses)은 대개 메모리의 하나의 일반적 영역에나란히 모여있는 경향이 있다. 그것을 고려하면, 현재 시스템 메모리의 working area 에 있는 데이터를 CPU 에 가까운(access 시간 관점에서) 어딘가로 이동시키는 것은 나름 일리가 있는 거다.

Cache Hierachy (캐쉬 계층구조)

현대 캐쉬 기술은 이것을 극단까지 가져간다: 이 기술은 캐쉬를 CPU 자체와 있는 동일한 실리콘(chip)까지 이동시킨다. 캐쉬 메모리는 우리의 오래된 친구 SRAM(staic RAM)이다, SRAM 은 모든 세대의 DRAM 보다 훨씬 빠르다. 캐쉬는 물리적(거리)으로 CPU 에 가까울 뿐만 아니라 우리가 만들 수 있는 가장빠른 종류의 RAM 이다.(엑세스 속도도 빠르다)

Cache 가 빠른 이유중 하나는 작기 때문이다. 시스템 메모리는 아마도 수 기가바이츠(gigabytes) 일 것이다. Cache 는 비교적 작다, 그리고 아주 드물게 1M byte 이상을 저장한다(인텔 i7-855U 이 8MB cache 다). 작은 것이 빠르다 그 이유는 처리해야할 주소 비츠(fewer address bits)가 더 적기 때문이다. 그리고 또 다른 이유는 CPU 가 필요로 하는 데이터가 이미 캐쉬에 있는지 없는지 아는 것이 더 쉽다(간단)하기 때문이다. (이것에 대해서는 잠시 후에 더 보겠다) 캐쉬 메모리를 크게 만드면, 캐쉬 오퍼레이션은 느려진다.

뭘 해야 하나? 캐쉬를 여러 개의 레이어로 나눈다, 그리고 레이어들을 하나의 계층구조로 만든다. 현대 마이크로프로세서들은 적어도 2 레이어의 캐쉬를 가진다,(종종 3 개의 layer). 첫번째 레이어(L1) 캐쉬는 CPU 에 가장 가깝게 있다. L2 캐쉬, L3 캐쉬 죽 이렇게 나간다. L1 캐쉬는 L2 캐쉬보다 더 빠르고(또한 더 작다), L2 은 L3 캐쉬보다 더 빠르다(또한 더 작다). 캐쉬 계층구조의 하단에는 시스템 메모리가 있다. 이곳은 CPU 에 의해서 직접 접근이 되는 데이터를 저장하는데 가장 크고 또한 가장 느린 곳이다. 당연히 시스템 메모리내의 데이터는 하드 디스크나 SSD 스토리지(여전히 속도가 느리며, CPU 가 메모리 주소로 접근할 수 없다)로 쓰여질 수 있다.

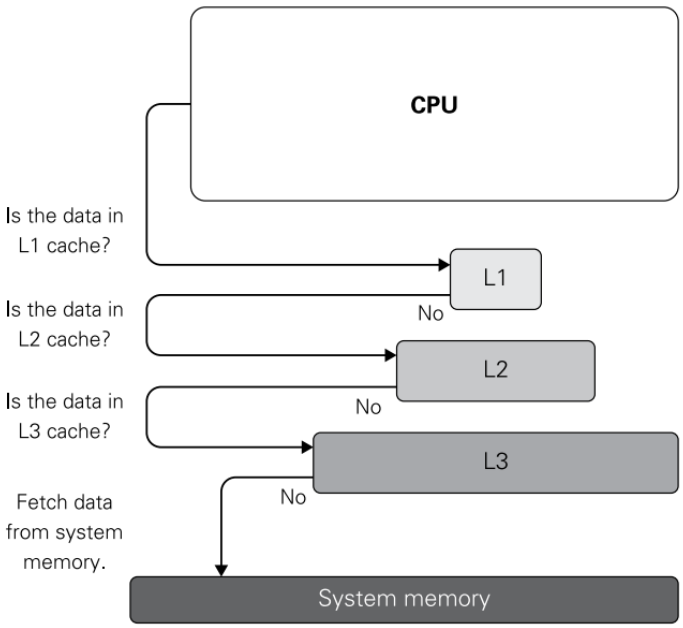

FIGURE 3-8: A multi-level cache

캐쉬 레이어의 수와 각 레이어의 사이즈는 마이크로프로세서에 따라서 다 다르다. Intel Core i7 family 는 각 core 마다 32KB L1 cache, 각 core 마다 256KB L2 캐쉬, 그리고 모든 core 들 사이에서 공유가 되는 하나의 L3 캐쉬를 가지고 있다. L3 cache 사이즈는 4MB~8MB 이다(마이크로프로세서 모델에 따라 다르다). 예전 라즈베리파이 모델들에서 사용된 ARM11 프로세서는 16KB L1 캐쉬 두개를 가지고 있었다. 하나는 instruction 용, 다른 하나는 data 용. 128KB L2 캐쉬는 ARM11 CPU 를 둘러싸는 SoC 실리콘(System-on-a-chip silicon)에 있다. 그러나 숨겨진 문제가 있다. L2 캐쉬가 ARM11 CPU 와 Video Core IV 그래픽스 프로세서(GPU) 사이에서 공유(그래픽스 프로세서가 우선 순위를 가지고)가 된다.

라즈베리 파이는 더 이상 L3 캐쉬를 가지고 있지 않다.

Cache Lines and Cache Mapping

Figure3-8(그림 3-8)는 약간 프로그래밍 순서도 같이 보인다, 그리고 결정해야할 것들이 많아서 프로세스(과정)는 아마도 느릴 것이라고 여러분은 생각할 거다: 근데 느리지가 않다. 주어진 일련의 연속된 메모리 로케이션들이 이미 캐쉬에 있는지 결정하는 것은 번개와 같이 빠르다. CPU 칩에 전용 로직이 빌드되어 있다.

주어진 메모리 위치가 캐쉬에 있는지 알아내는 2 개의 일반적인 매카니즘이 있다. 하나는 계산(calculation)에 의존하고 다른 하나는 검색(searching – 시퀀셜 검색)에 의존한다. 둘 방식 모두 심각한 단점들이 있다. 대부분의 현대 컴퓨터들이 사용하는 방식은 두 개의 접근방식을 가만한 일종의 하이브리드 방식이다. 각각의 순수한(Pure) 접근방식(절충이 아닌 각각의 방식)이, 칩에 구현이 실제로 되는 것은 매우 드물겠지만, 우리가 사용할 하이브리드 절충방식에 대해서 이해를 하기위해서는 두 방식이 모두 어떻게 작동하는지 아는 것이 좋다.

먼저, 캐슁(caching)에 대한 일반적인 기술 백그라운드를 보자. Caching 은 한번에 하나의 data word 만을 결코 caching 하지 않는다 그 이유중에 하나로, 참조지역성(locality of reference)을 사용하기 위해서다. Caching 은 요전에 SDRAM 에서 자세히 설명했던 memory controller feature 와 아주 잘 작동한다.: “burst-mode” 로직 – a single word 를 읽고/쓰고 하는데 걸리는 동일한 시간동안에 시스템 메모리로부터 multiple words 를 읽고/쓰고 할 수 있는 logic 이다. Cache 는 cache lines 이라고 불리는 fixed-sized block 들로 (일반적으로) 읽고/쓰여진다. Cache line 의 사이즈는 다양할 수 있다, 하지만 현대 시스템즈에서는 일반적으로 32 bytes 다. 라즈베리파이에 ARM11 프로세서 뿐만 아니라, 많은 인텔 CPU 들에서도 마찬가지다. Cache 에 저장될 수 있는 Cache line 의 개수는 cache size (in bytes)/ cache line (in bytes). 라즈베리파이 L1 cache 의 경우를 보면, 16,384 bytes / 32-byte size of a cache line, 즉 L1 cache 에 512 개의 가능한 cache lines 이 있다.

Cache memory 는 단지 CPU 내에 매우 빠른 메모리 로케이션들을 죽 나열해 놓은 게 아니다. Cache 는 그 자신만에게만 해당되는 특정 구조를 가지고 있다. 32 bytes 의 데이터 뿐만 아니라, cache 에 각 로케이션은 cache tag 라는 추가적인 필드값도 가지고 있다. 그것이 cache controller 가 시스템 메모리상 어디에서 cache line 이 오는지를 결정할 수 있도록 해준다. 또한 두개의 single-bit flag(valid bit & dirty bit)들이 각 cache line 에 있다:

Valid bit: cache line 에 valid data 가 있는지 여부를 표시. Cache 가 초기화되고 나면, 모든 cach line 들의 valid bit 이 false(0)로 설정이 된다. 그리고 메모리 블락이 cache line 으로 읽어져 들어간 경우에만 true(1)로 변경된다.

Dirty bit: cache line 데이터에 일부가 CPU 에 의해서 변경이 됐다는 것을 표시한다. 그리고 그 데이터는 시스템 메모리로 write-back (쓰여) 져야 한다는 것을 표시한다.

Cache tag 는 시스템 메모리에 주소(cache line 이 그 주소로부터 채워졌다) 주소로부터 나왔다. 메모리 주소가 읽거나 쓰기위해 주어졌을 때, 그 주소는 3 가지 부분으로 나누어 진다.

Cache tag: cache line 이 메모리에 어디서 온 것인지를 가르킨다. 이것들이 메모리 주소에서 최상위 비츠(bits)이다. 그리고 cache-line-sized and aligned block of system memory 를 유니크하게 가르킨다(refer to). Tag 는 cache line 자체와 함께 저장이 된다.

Index: 데이터가 캐쉬에 있을 경우 시스템 메모리 주소로부터의 데이터가(시스템 메모리 주소 상에 있는 데이터) 있을 cache line 을 가르킨다. Direct-mapped cache 의 경우, 비츠(bits)의 개수는 cache 내에 모든 lines(blocks)들로부터 하나의 cache line 을 지정하는데 필요한 비트 개수이다. 512-line direct-mapped cache 인 경우 9 bits 이다.

Offset: cache line 내에 어느 바이트가 시스템 메모리 주소(tag 를 생성하는 주소)에 의해서 지정된 바이트와 매칭이 되는지를말해준다. 시스템 메모리에 최하위 비츠(bits)이다. 비츠의 수는 하나의 cache line 내에 모든 bytes 로부터 하나의 바이트를 지정하는데 필요한 비트의 개수이다. 32-byte cache line 에서는 5 bits 이다.

Block field 와 word field 는 어디에도 저장이 되지 않는다. 이것들은 cache access 동안에 사용이 되지만 하나의 data word 가 cache 로부터 읽거나 쓰여진 후에는 버려진다.

Figure 3-9 에서는, Cache line structure(캐쉬라인 구조)와 시스템 메모리 주소가 cache access 를 위해서 어떤식으로 구성이 되는지를 보여준다. Cache line structure 의 일부 세부적인 부분들은 system specfics(특정 시스템에 국한되는, 예를 들어 cache 크기, cache line 크기 등등)과 caching 을 관리하기 위해서 시스템에 의해서 사용되는 특정 매커니즘에 따라서 다를 수 있다.

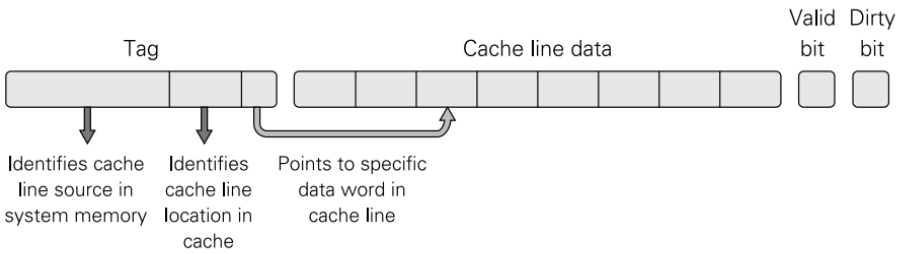

FIGURE 3-9: Cache line structure

Cache 기술에서 가장 중요한 이슈는 시스템 메모리로부터의 데이터(시스템 메모리 상에 데이터)가 cache 에 어디에 배치가 되는가 이다. 이걸 cache mapping 이라고 부른다. 그리고 이것이 CPU 가 요청된 주소(requested address)가 cache 에 있는지를 아는 방식을 결정한다. 명칭에서 의미하듯이, cache mapping 은 어떻게 시스템 메모리에 cache-line-sized data block 의 위치가 cache 내에 가능한 포지션(위치)과 연결되는지에 관한 것이다.

Direct Mapping

가장 오래되고, 간단한 cache mapping 기술, 그리고 지금까지 우리가 묵시적으로 가정했던 것이 바로 direct mapping 이다. 간단히 말해서 system memory 의 첫번째 block 이 cache 에 첫번째 cache line 에만 저장될 수 있다; 시스템 메모리에 두번째 block 은 cache 에 두번째 cache line 에만 저장될 수 있다. 계속 이런식으로 간다. 물론 cache memory 보다 시스템 메모리가 훨씬 많다. 그래서 cache 가 full 되었을 때, 시퀀스가가 “wraps around(앞으로 되돌아 간다)”되고 cache 에 천번째 로케이션에서 다시 시작한다.

아래 그림이 이것을 이해하는 데 큰 도움이 된다. Figure 3-10 을 다음 설명동안 참고하기 바란다.

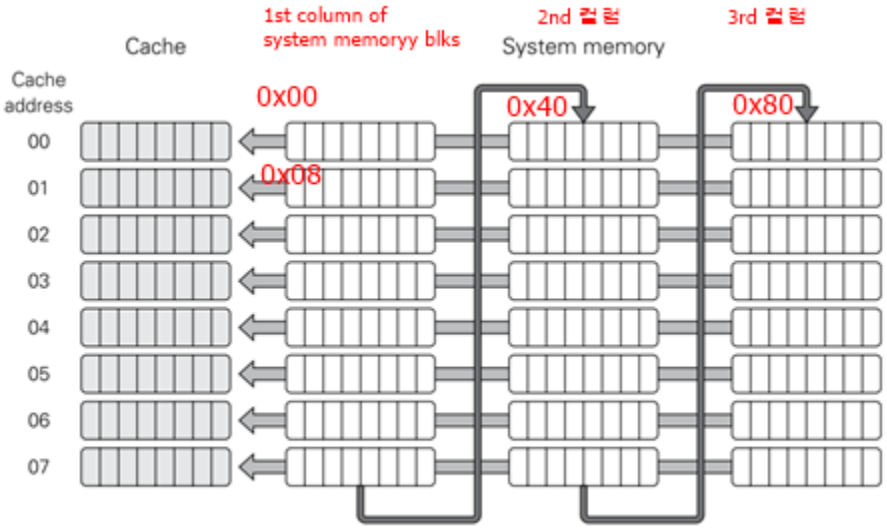

FIGURE 3-10: Direct cache mapping

Figure 3-10 에 묘사된 간단한 형태의 direct mapping 예제에는, cache 내에 8 개의 locations 들이 있다. 이들 각각은 하나의 cache line 을 저장한다.(간단하게 보여주기 위해, cache tag 들은 표시하지

않았다). 각각의 cache line 은 8 bytes 를 가진다. 시스템 메모리의 처음(앞부분) 24 blocks 이 보인다. 시스템 메모리에 각 블락은 하나의 cache line 사이즈다(즉, 여기서는 8 bytes). 모든 caching systems 에서 처럼, data 는 cache-line-sized chunk 들로 시스템 메모리에서 읽거나 쓰여진다. 시스템 메모리 블락들의 각 컬럼 위에 16 진수 숫자들은 각 컬럼 시작점의 byte address(*모든 메모리 같은 것은 byte addressing 을 쓴다. 하나 하나 location 이 1 byte 이고 1 byte 마다 주소가 있다. 그래서 하나의 word 의 경우 선두 byte 주소를 사용한다)이다. 각 컬럼은 64 bytes 를 나타낸다, 두번째 컬럼의 주소는 0 + 0x40 (10 진수로 64) 이고 세번째 컬럼의 시작 주소는 0x40+ 0x40, 즉 0x80 이다.(10 진수 표기로 128)

시스템 메모리 블락들을 cache line 들에 맵핑하는 것은 다음과 같이 작동한다: 시스템 메모리에 block 0 (주소 0x00 에서 시작)은 항상 cache lin 0 에 맵핑된다; block 1(주소 0x08 에서 시작)은 항상 cache line 1 에 맵핑된다; 계속 이렇게 진행된다. Cache line 들이 다 없어질 때까지는 아주 이해하기 쉽다. figure 3-10 예제에 cache 내에는 오직 8 개의 cache lines 만 있다. Cache lines 을 다 사용하고 나면, 순서가 “wraps around”되어서 다시 앞으로 간다 그리고 다시 시작한다: block 8(주소 0x40 에서 시작)은 cache line0 에 맵핑된다, block 9 (주소 0x48)은 cache lin1 에 맵핑된다, 그렇게 진행된다. 이런 것을 modulo *n* mapping 이라고 부른다. 여기서 *n* 은 cache 내에 locations 의 수(여기 예에서는 8 개) 이다. 어느 주어진 시스템 메모리 블락의 위치(location)가 cache 에 맵핑될 때 Memory block number % 8 (module 8)이 될 거다.

“modulo”용어는 나누기한 후에 나머지 계산을 의미한다. 초등학교 아이들은 64/10 이 나머지가 4 라는 것을 배운다. 그래서, 64 % 10 은 간단하게 4 가 된다. 여기 예제에서 System memory block21 이 어느 cache line 에 맵핑되는지 알고 싶다면, 21 % 8 을 계산하면 된다. 답은 5 이다. 메모리 블락 21 은 항상 cache line 5 에 맵핑된다. Figure 3-10 에서 메모리 블락 21 이 cache line 5 에 맵핑되는지 확인하기 위해서 메모리 블락들을 세어 보아라(물론 0 부터)

System memory blocks to cache lines 의 direct mapping 은 수학적으로 정확하다: 시스템 메모리의 주어진 block 은 항상 cache 내에 동일한 위치(location)에 저장이 된다. CPU 가 fetch 할 필요가 있는 메모리 주소가 cache 에 있는지를 결정한다. 그 결정하는 방식은, 그 메모리 블락이 항상 캐쉬의 어느 포지션(위치)로 가는지를 계산(calulation)하고 나서 cache tag(시스템 메모리 주소 내에 매칭되는 비츠(bits)를 가진)의 tag field 에 값과 비교(comparison)을 해서 cache 에 있는지를 결정 해 낸다. 이게 매치가 되면, cache hit 이 된 거이다. 이게 매치가 안되면 cache miss 가 있는 거다.

CPU 는 계산(calculation)과 비교(comparison)를 굉장히 잘한다. 그리고 direct cache mapping 은 현재 있는 것 중에서 가장 빠른 캐시 매커니즘이다. 하지만, 단점도 있다. 시스템 메모리로부터 blocks 들이 cache 내에 저장되는 위치(포지션)에 대한 유연성이 전혀 없다. 이것이 CPU 가 번갈아 가면서 특정 메모리 블락들을 읽는 메모리 읽기들(reads)를 하고 있는 소프트웨어를 돌리고 있을 때 문제가 될 수 있다.

Direct mapping 예제에서 보면, system memory blcok 4 은 같은 block 12, block 20, … (modulo 8)와 같은 cache location(cache line 4)에 맵핑된다. Software 가 (system memory) block 4 에 있는 주소를 읽는다고 가정하면; cache line 4 는 그게 거기에 없으면 그 블락을 받게 된다. 그리고 나서 그 소프트웨어는 block 12 으로부터 데이터가 필요할 수 있다. Block 4 가 cache 내에 있다면, block 12 는 캐쉬에 없다. 그 이유는 이 블락들은 항상 같은 cache location 으로 맵핑된다. 그래서 block 12 가 적재되고 block 4 를 overwrites(이걸 evicts 라 말한다 – 쫓아낸다라고 한다) 한다.

그 이후로 곧, 아마도 어떤 프로그램 루프가 실행될 때, 그 소프트웨어는 다시 (system memory) blcok4 로부터 데이터가 필요하다, 그래서 block 12 는 반드시 evict 한다. 그 루프가 계속 이런 식으로 진행이되면, cache 내에 thrashing 이 발생한다.(즉, 시스템 메모리로부터 반복되는 feches 가 발생). 이 thrashing 은 caching 에 의해서 얻은 속도의 이득을 없애 버린다. 사실 cashing machanism 의 오버헤드 때문에 cashing 을 전혀 하지 않는 경우 보다 thrashing 상황에서는 memory access 가 더 느려지게 된다.

(* 소프트 웨어는 메모리 주소로 data 를 읽는다)

Associative Mapping (연관 맵핑)

Direct mapping 이 제공하는 것 보다 cache mapping 에 있어서 더 많은 유연성이 필요하다. 이상적으로는, 여러분들은 접근이 되는 주소들과 관계 없이, 소프트웨어가 사용하는 가능한한 많은 시스템 메모리 블락들이 cache 내에 있기를 원할 것입니다. 여러분들이 주어진 블락을 cache 내에 가용 가능한 (cache) line 에 적잭할 수 있다면, cache space 를 더 잘 사용할 수 있게 하는 replacement policy(한마디로 간단히 말해서 어느 새로운 메모리 블락을 cache 에 쓸 때 어떤 cache line 을 쫓아(evict) 낼 것인지를 결정하는 것을 말한다)를 구현할 수 있을 겁니다.

Replacement policy 가 하는 일은 대개 cache thrashing 을 피하는 것이다. 이 일은 매우 어렵다, 그리고 replacement policies 들은 어떤 새로운 메모리 블락이 cache 에 들어갈 필요가 있을 때 어떤 cache line 을 쫓아낼(evict) 것인지를 결정하는 여러 알고리즘의 조합으로 주로 많들어 진다.

여기 자주 사용되는 replacement policies(정책들) 있다:

First in first out (FIFO): cache 가 full 차면, cache 에 쓰여진 첫번째 cache line 이 쪽겨나는(evict) 대상이 된다.

Least recently used (LRU): cache lines 들에는 timpestamps(사건이 발생한 실제 시간)이 주어진다. 그리고 시스템은 cache line 이 사용된 시점을 기록한다. 하나의 새로운 cache line 이 쓰여져야만 할때는, 가장 오랫동안 엑세스가 되지 않은 cache line 이 쫓겨난다(evict). Timestamp 을 관리하는 것은 시간이 많이 걸리고 복잡하다.

Random: 이것은 반직관적으로 들릴 것이다, 그러나 logic 측면에서, 가장 싸고 가장 효과적인 replacement policies 들 중에 하나가 완전히 랜덤하게 쫓아낼(evict) 하나의 cache line 을 고른다. Random eviction 은 thrashing 이 일어나지 않도록 해준다. 이것은 또한 FIFO 나 LRU 처럼 소프트웨어에서 사용되는 알고리즘들에 민감하지도 않다.

Not most recently used(NMRU): 쫓겨날(evict) cache line 은 랜덤하게 선택이 된다. 그러나 이게 약간 변경이 되어서 가장 최근에 사용된 cache line 이 기억이 되고, 쫓겨날(evict) 대상으로 선택이 되지 않는다. 이 정책은 랜덤 정책만큼 거의 구현하는데 많은 비용(노력, 복잡성 등)이 들어가지 않는다, 그리고 그냥 랜덤인 것 보다 조금더 낫게(better) 작동한다.

라즈베리 파이에 있는 ARM processors 들은 FIFO 또는 random 정책을 사용할 수 있다. Configuration bit 에 의해서 설정 가능하다. 대부분의 경우에 replacement policy 는 random 이다.

Cache space(캐시 공간)을 사용하는 가장 유연한 방법은 새로운 cache line 을 cache 내에 어디에나 (replacement policy 가 뭘 지시하던) 배치할 수 있게 하는 것이다. CPU 는 여전히 자신이 필요로 하는 데이터가 cache 에 있는지 없는지를 결정할 수 있어야 한다. 그리고 data blocks 들이 cache 내에 어디에나 저장될 수 있다면, 하나의 계산과 비교(a single caculation and comparison)에 의해서 더 이상 그러한 결정(위에 데이터가 캐쉬에 어디에 있는지 결정)이 될 수 없다. 대신에, CPU 는 cache 내에서 주어진 블락을 검색해야(search for) 한다.

Calculation & comparision 과 비교해서, 검색(searching)이 엄청나게 computer-intensive 한 과정이다. 한번에 하나씩 Cache lines 들 검색은 가능한 성능 이득(any possible performance gain)을 다 잡아먹게 될 것이다. 해결책은 associative memory 라는 기술을 사용하는 것이다. 모든 메모리들처럼, associative memroy(연관 메모리)가 일련의 죽 연결된 storage locations(in a series of storage locations)들 내에 데이터를 저장한다. Associative memory 가 가지고 있지 않는 것은 전통적인 숫자 주소 체계이다. 대신에 storage locations 들이 그 들 안에 저장된 것(hjs:어떤 hash 값 같은 걸 key 로 해서 index 를 구하는 방식 같은 형태)에 의해서 addressed 되어 진다.(여기 빨간색 부분이 associative 의 의미를 설명하는 거다)

A full associative cache 에서는, 이전 처럼 하나의 메모리 엑세스(접근)는 하나의 cache tag 가 시스템 메모리 주소로부터 생성되어지게 한다. 그러나, 이 생성된 태그를 하나의 uniquely specified cache line 에 매칭되는 tag 와 비교하는 대신에, 이 경우에는, associative memory system (연관 메모리 시스템)는 생성된 tag 를 cache 내에 저장된 모든 tag 들과 동시에 병렬로 비교한다.
연관메모리시스템이 match 을 찾으면, cache hit 이 되고, 해당 cache line 은 CPU 에게 주어진다.
연관메모리시스템이 match 을 못 찾으면, 이것은 cache miss 이다; 하나의 cache line 은 그 cache 로부터 반드시 쫓아(evict)내어져야 한다.replacement policy 에 의해서 정해진다. 그리고 요청된 시스템 메모리 블락은 새롭게 빈 cache line 으로 읽어져 들어간다.

전통적인 어드레싱(hjs: associative memory 다른 어드레싱 방식)과 시퀀셜 검색에 익숙한 사람들에게는, 이것이 약간 매직 처럼 들릴 수가 있다. 그러나, 비록 병렬검색이 빠르기는 하지만, associative memory 는 CPU 위에 많은 양의 die space 를 필요로 하는 많은 전용 로직을 요구한다. 가장 작은 또는 가장 performance-critical caches 들 제외하고 거의 모든 caches 들에 있어서, pattern-matching logic 는 실제로 사용되기에는 너무나 비싸다.(트랜지스트 개수, 그리고 결국은 time delays 측면에서 너무 비싸다)

++

Die space(다이 공간)는 하나의 실리콘 칩(a silicon chip-반도체 제조 공정 중에는 die 라고 부름)의 공간(area)이다. 이 공간은 트랜지스터들을 제조하는데 사용이 된다. 이 트랜지스터들로부터 chip's digital logic 이 만들어진다. 주어진 die 에는 트랜지스터들에 사용할 영역이 많지 않다(공간에 한계가 있다)

그래서 chip designers 들은 자신이 가진 그 공간을 사용함에 있어서 매우 주위를 기우려야 한다. Die space 와 chip functionality 사이에 trade-off(공간에 한계로 어떤 것을 빼고 넣고 결정하는거)는 large-scale chip design 에서 가장 오래된 하나의 어려운 문제(challenge)다.

++

Set-Associative Cache (세트 연관 캐시) – 제일 많이 쓰는 방식

한쪽 극단에는 번개처럼 빠르고 컴팩트한 direct cache mapping(새로운 cache line 이 저장되는 위치측면에서 전혀 유연성이 없는) 있다. 반대쪽 극단에는 완전히 유연한 associative cache mapping(구현하기에는 너무 많은 on-chip logic 이 필요하다) 있다. 이런 경우 늘 해결책은 중간 어딘가에 있다.

그 절충이 set-associative cache 라고 부르는 것이다. Set-associative caching system 은 cashe lines 들을 sets 들(그룹)으로 reorganise(재조직)한다. 각 세트는 2, 4,8, 또는 16 cache lines(각 cache line 은 data block 과 tag 를 가진다.)들을 가지고 있다. Figure 3-11(그림 3-11)은 세트 당 4 개의 cache lines 를 가진 하나의 set-associative cache 의 심플한 다이어그램을 보여준다. 세트당 4 개의 (cache) lines 을 가지고 있기 때문에, 4-way set-associative cache 라고 부른다. 이게 오늘날 많은 랩탑과 데스크탑 컴퓨터들에서 사용되는 것 뿐만 아니라 라즈베리파이에서도 사용되는 cache scheme(캐시 방식)이다.

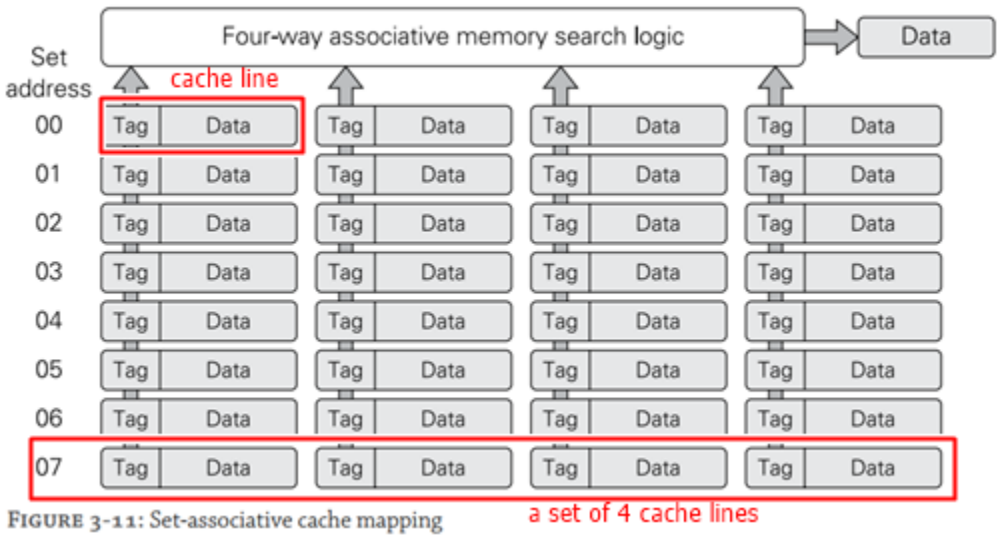

FIGURE 3-11: Set-associative cache mapping

주어진 하나의 세트에 맵핑되는 memory locations 들은 여전히 direct mapping 에 의해서 결정이 된다. 이것은, 하지만 우리는 하나의 incomming block 를 어디에 배치 시킬 것인가 측면에서는 약간의 유연성을 이제는 가지게 됐다는 것을 제외하고는, 시스템 메모리 주소들과 cache 위치들(positions)의 modulo 관계가 여전히 유지가 된다는 것을 의미한다. 앞에서 주어진 예제인 eight-line direct-mapped cache(순수한(pure) diret-mapping scheme(방식) 아래에서 blocked 되 듯이 시스템 메모리로부터 2, 10, 18 and 26 를 block 한다)를 상기해 보세요.

하지만 문제는 여전히 남는다: 하나의 세트내에 cache lines 들에 저장된 4 개의 시스템 메모리 블락들이 있다. 컴퓨터는 주어진 메모리 주소가 어느 세트에 해당이 되는지 쉽게 계산 할수 있다. 하지만, 컴퓨터는주어진 세트 내에 어느 cache line 이 해당 요청한 주소를 가지고 있는지 간단한 계산으로(쉽게) 결정할 수가 없다.

CPU 는 반드시 어느 cache line's tag 가 요청한 주소와 매치가 되는지 알기 위해서는 하나의 세트 내에 4 개의 cache lines 을 검색(search)해야 한다. Associative memory 가 이 검색을 한다. 이것은 각 cache tag 을 보고나서 검색이 match 를 찾았을 때 중지되는 시퀀셜 검색이 아니다. 대신에, 병렬 비교기들(parallel comparators)이 거의 동시에 그 cache line 내에 4 개의 tags 들로부터의 bits 를 (요청한 메모리 주소로부터) 생성된 태그(generated tag)내에 해당 bits 와 비교한다. 이 로직은 근데 내부적으로 복잡하다, 그러나 오직 4 개의 위치(locations)만 검색이 되기 때문에, 검색이 빠르게 될 수 있다.

그 과정은 이렇게 작동한다: CPU 가 하나의 메모리 블락이 어느 세트에 있어야 하는지를 결정 한다, 시스템 메모리 주소로부터. (이것은 direct cache mapping 에서와 같은 방식으로 된다.). 그리고 나서 CPU 는 그 주소를 associative memory logic 에 제출한다, 그리고 associative memory 는 그 세트 내에 어떤 (cache) line 이 요청한 blcok(a cache hit)을 가지고 있는지 CPU 에게 말해 주거나, 또는 cache miss 를 등록한다.

요청 블록은 그리고 나서 시스템 메모리로부터 읽어들여지고 그 세트내에 4 개의 (cache) lines 들 중에서 하나에 배치가 된다, replacement policy 에 따라서. 정리하면: set-associative cache 는 하나의 cache 를 여러 개의 sets(세트)들로 나눈다, 라즈베리파이에서 사용되는 ARM11 의 경우에는 4 개의 cache lines 들을 가진다. CPU 는 direct-mapping 방식을 통해서 주어진 주소가 어느 세트에 있어야 하는지를 계산/결정할 수 있다, 그리고 나서 CPU 는 그 세트 내에 매칭되는 cache line 으로

바로 가기 위해서 associative memory 의 pattern-matching mechanism 사용한다. – 또는 검색이 실패했다면, cache miss 를 등록한다.

Writing Cache Back to Memory

지금까지, 우리는 caching 을 메모리로부터 읽는 것으로만 완전히 가정해왔다. 물론 읽어온 데이터가 종종 변경된다. CPU 가 하나의 cache line (32 bytes)어딘가에 있는 하나의 data word 를 변경했을 때, 그 cache line 은 a single-bit flag 를 사용해서 "dirty"로 표시가 된다. 하나의 cache line 의 dirty bit 이 설정되고나면, 그 (cache) line 은 데이터가 처음에 있어던(읽어졌던 장소) 메모리내에 blcok 으로 다시 쓰여져야 만 하다.(write-back to memory 발생). 무슨 일이 발생하더라도, 시스템 메모리 블락들과 그들의 연결된(associated) cache lines 들은 반드시 일관성이 유지되어야 한다. Cache 에 변경이 시스템 메모리로 다시 쓰여(write-back)지지 않으면,만약 replacement policy 가 하나의 새로운 블락을 변경이 발생된 동일한 cache line 로 읽어들이면 이러한 변경들은 다 사라져 버린다.

Cache 와 메모리 의 일관성을 유지하기 위한 2 개의 일반적인 접근 방식이 있다. 이 두개를 합쳐서 이것들을 "cache write policies"라고 부른다.

Wrtie-through(to memory): 하나의 cache line 내에 하나의 data word 가 CPU 에 의해서 변경이 될 때 마다, 그 cache line 은 메모리로 즉시 쓰여진다. (cache) line 이 쓰여질 때 마다 이것이(메모리로 쓰는 것이) 일어난다, 쓰기들(writes – 여러 번의 쓰기들)이 모두 동일한 cache line 내에 있다고 하더라도. 예상대로, 하나의 cache line 을 메모리에 여러 번 쓰기(write –back)하는 낭비하는 시간이 있다, 하지만 메모리에 대한 CPU's view 는 메모리에 실제 있는 데이터(copy)와 일관성(consistent)을 유지한다; display controller 와 같은 주변 장치가 또한 메모리를 엑세스(접근)한다면, 이것(일관성)은 아주 중요하다.

Write-back: replacement policy 가 cache 로부터 dirty cache line 을 쫓아내기(evict)로 했을 때만 그 "dirty" cache line 이 메모리로 쓰여진다.(write-back 이 발생). 새로운 시스템 메모리 블록이 cache line 에 적재되기 전에, 그 (cache) line 의 현재 contents 는 시스템 메모리내의 그 데이터의 오리지날 블록으로 복사가 된다.(copy-back 발생). Write-back 은 많은 불필요한 시스템 메모리 쓰기들(writes)를 피한다. 대신에 덜 엄격한 개념의 일관성을 가지게 된다.