Memory Hierarchy Design

Chapter 5 - Computer Architecture : *A Quantitative Approach* - Hennessy & Patterson

Cache Memory

- Direct mapped, **Fully associative**, set associative cache.
- Write-through and write-back cache. Write buffer.
- Cache replacement - Random, **LRU**, FIFO algorithms.
- Data fetch and tag checking done in parallel.
- Write allocate and no-write allocate.

Cache Equations

$$2^{\text{index}} = \frac{\text{Cache Size}}{\text{Block Size} \times \text{Degree of Associativity}}$$

CPU time = (CPU clock cycles + Memory stall cycles) $\times$ Clock period

Memory stall cycles = IC $\times$ $\frac{\text{Misses}}{\text{Instruction}}$ $\times$ Miss Penalty

Memory stall cycles = Number of misses $\times$ Miss penalty

Cache Equations

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory access}}{\text{instruction}}$$

Average memory access time = Hit time + Miss rate × Miss penalty

$$\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{execution}} + \frac{\text{Memory stall cycles}}{\text{Instruction}}) \times \text{Clock period}$$

$$\text{CPU time} = \text{IC} \times (\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}) \times \text{Clock period}$$

Cache Equations

CPU time = IC × (CPI$_{\text{execution}}$ + Miss rate × $\frac{\text{Memory access}}{\text{Instruction}}$ × Miss penalty) × Clock period

$\frac{\text{Memory stall cycles}}{\text{Instruction}}$ = $\frac{\text{Misses}}{\text{Instruction}}$ × (Total miss latency − Overlapped miss latency)

Average memory access time = Hit time$_{L1}$ + Miss rate$_{L1}$ × (Hit time$_{L2}$ + Miss rate$_{L2}$ × Miss penalty$_{L}$2)

Cache Equations

$$\frac{\textbf{Memory stall cycles}}{\textbf{Instruction}} = \frac{\textbf{Misses}_{L1}}{\textbf{Instruction}} \times \textbf{Hit time}_{L2} + \frac{\textbf{Misses}_{L2}}{\textbf{Instruction}} \times \textbf{Miss penalty}_{L2}.$$

$$\boxed{\textbf{Average Memory Access Time}}$$

$$\boxed{\textbf{is}}$$

$$\boxed{\textbf{Hit time} + \textbf{Miss rate} \times \textbf{Miss penalty}}$$

Reducing Cache Miss Penalty

- **Multilevel Cache.**
- **Critical word first and early restart.**
- **Giving priority to read miss over write miss.**
- **Merging write buffer.**
- **Victim cache.**

Multilevel Cache

- Fast and Large.
- **Average memory access time = Hit time$_{L1}$ + Miss rate$_{L1}$ $\times$ (Hit time$_{L2}$ + Miss rate$_{L2}$ $\times$ Miss penalty$_{L}2$)**
- Local miss rate - same as Miss rate$_{L1}$ for level-1, also same as Miss rate$_{L2}$ for level-2.
- Global miss rate - same as Miss rate$_{L1}$ for level-1 but is Miss rate$_{L1}$ $\times$ Miss rate$_{L2}$.

Multilevel Cache

- Global cache miss rate is very similar to single cache miss rate of the L2 cache - from experimental data.
- Local miss rate is not a good measure for the L2 cache, as it is dependent on L1 cache.
- How big the L2 cache should be and what should be the set associativity?

Multilevel Cache

- Multilevel inclusion of data - data in L1 should also be present in L2.
- Consistency of data between IO and cache and among caches is easier to maintain in case on multilevel inclusion.
- Block sizes at different levels - inclusion and second level miss.
- Multilevel inclusion is bad for smaller L2 cache - multilevel exclusion.

Critical Word First

- Cache block/line is multi-word.
- Often the CPU needs one word.
- Send the critical word first without waiting for the full cache line to be loaded - critical word first (wrapped or requested word first fetch).
- Fetch in normal order, but as soon as the requested word is available, send it to the CPU.

Priority of Read Misses

- **Write-through cache and write buffer.**
- **Write buffer may hold updated value (not yet written)**

```
SW   R3, 512(R0)
LW   R1, 1024(R0)
LW   R2, 512(R0)
```

Consider a direct mapped, write-through cache that maps 512 and 1024 in the same cache block.
Will the value of R2 be always equal to R3.

Priority of Read Misses

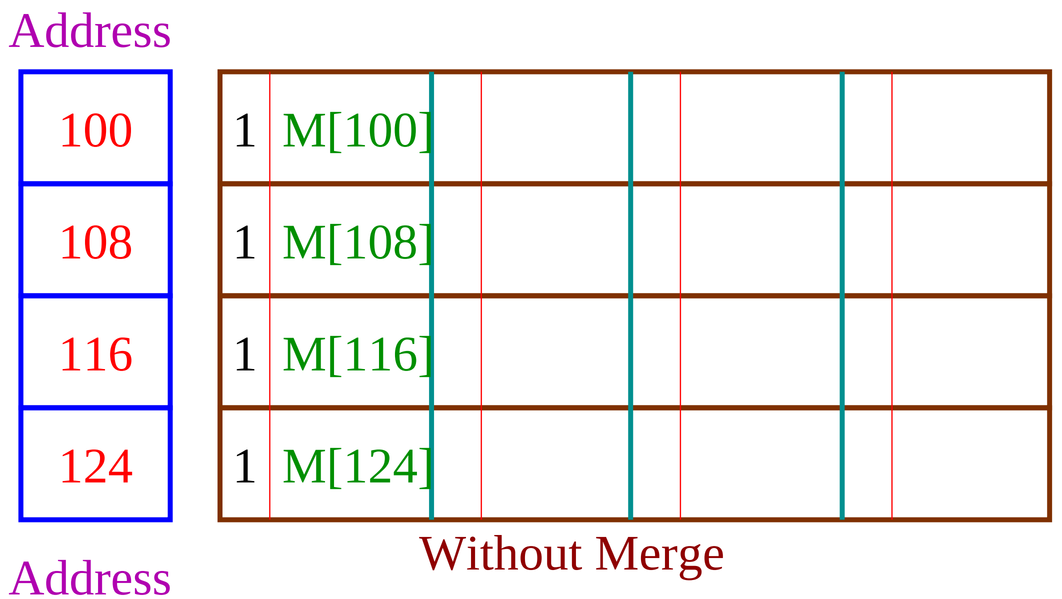
- **The store will put the data of R3 in the write buffer.**
- **There will be miss for the 1st load.**
- **There will also be miss for the second load, which will read Mem[512].**
- **It may get a wrong value if the write buffer has not yet finished writing.**

Priority of Read Misses

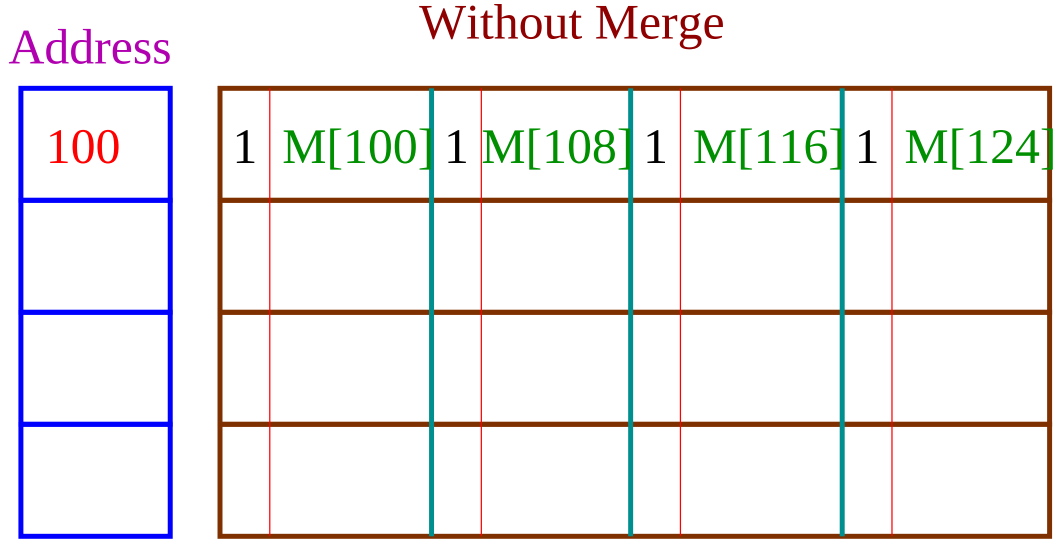
- Simple solution - read miss waits until write buffer is empty.
- Check the content of the write buffer, if there is no match and the memory is available, read miss continue.

Merging Write Buffer

- Write-through cache sends all store to the next level through a write buffer.
- If the write buffer contains other blocks, the new data address can be checked to see whether they can be merged.
- Multi-word memory write is faster.



Without Merge



With Merge

Victim Cache

- Remember what was discarded in case it is needed again.
- Discarded data on miss are saved in a small fully associative cache called victim cache.
- It is present on the refill path of the cache.

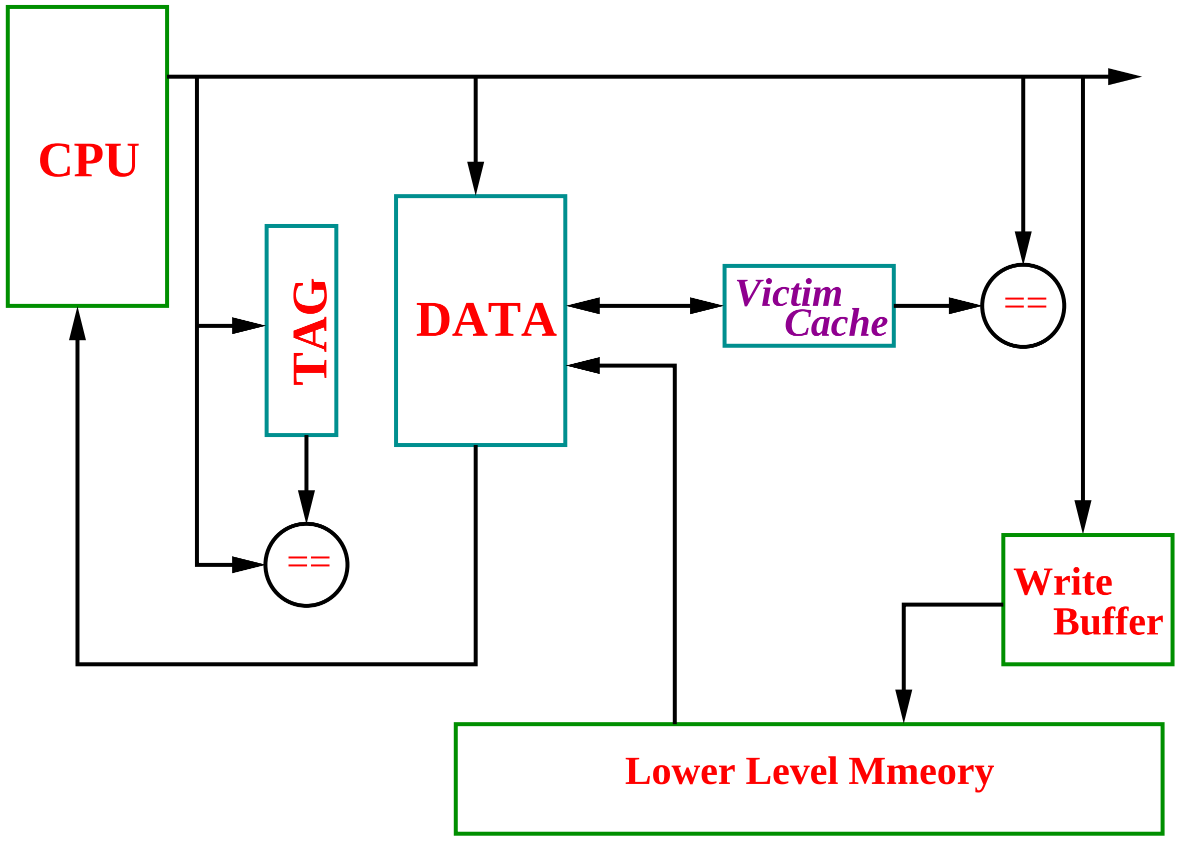


Figure 1: Victim Cache

Reducing Miss Rate

- **Compulsory miss - misses in an infinite cache.**
- **Capacity miss - misses in a fully associative cache.**
- **Conflict miss - misses in direct and set associative cache.**

Different Associativity and Sizes

- **Compulsory misses are independent of sizes.**
- **Capacity misses decreases with the increase in size.**
- **Conflict misses decreases with the increase in associativity.**

Rule of Thumb

- **For cache size less than 128KB, the miss rate for a direct mapped cache of size N is same as the miss rate of a 2-way set associative cache of size N/2.**

What to Do?

- Fully associative cache - no conflict miss, large hardware cost, slower clock rate resulting larger hit time.
- Larger cache memory with lower associativity -
- Larger block size will reduce compulsory misses - more miss penalty, more hit time.
- Miss rate versus miss penalty and hit rate.

Large Block Size

- Reduce compulsory misses - spatial locality.
- Increase miss penalty - time of transfer from the lower level.
- May increase in conflict as well as capacity misses for small cache -

Large Block Size

- **Optimum block size for a cache size - depends on the lower level memory subsystem - bandwidth and latency.**
- **32 - 64Bytes is often used.**

Large Cache Size

- Reduces capacity misses.
- High cost and higher hit time.
- Used as L2 or L3 cache.

Higher Associativity

- **8-way set associativity is as good as fully associative cache - change in capacity miss is negligible**
- **For cache size less than 128KB, the miss rate for a direct mapped cache of size N is same as a same as the miss rate of a 2-way set associative cache of size N/2.**

Compiler Optimization

- **Long cache block - aligning a basic block to the cache block.**
- **Reordering the procedures of a program reduces the instruction miss rates of conflict misses!**
- **Profiling information can determine likely conflicts between groups of instructions.**

Compiler Optimization

- McFarling[1989] - 2KB direct mapped instruction cache with block size 4Bytes - 50% reduction in miss. It was 75% with 8KB cache.
- Best performance when some instructions are prevented to enter the cache!

Optimization for Data

- A code is transformed so that the spatial and temporal locality of data access increases.

Loop Optimization

```
for(j = 0; j < 100; ++j)
   for(i = 0; i < 5000; ++i)
        x[i][j] = 2 * x[i][j] ;
```

What is wrong with the C code?

Blocking

```
for(i = 0; i < n; ++i)
   for(j = 0; j < n; ++j) {
     temp = 0 ;
     for(k = 0; k < n; ++k)
        temp = temp + y[i][k]*z[k][j] ;
     x[i][j] = temp ;
   }
```

Blocking

- **Two inner loops read $n \times n$ elements of z and n elements of y, n times each, and writes n elements of x.**
- **Number of capacity misses depends on n and the size of the cache.**
- **If the cache can hold all three matrices and there is no conflict, then there is no capacity misses.**

Blocking

- **In the worst case thee are $2n^3 + n^2$ memory access!**
- **The matrix is split into blocks of sub-matrices.**

Blocking

```
for(jj = 0; jj < n; jj = jj + b)
for(kk = 0; kk < n; kk = kk + b)
for(i = 0; i < n; ++i)
   for(j = jj; j < min(jj+b, n); ++j) {
     temp = 0 ;
     for(k = kk; k < min(kk+b, n); ++k)
        temp = temp + y[i][k]*z[k][j] ;
     x[i][j] = x[i][j] + temp ;
   }
```

Reducing Cache Miss Penalty or Miss Rate

via

Parallelism

Non-blocking or Lockup-free Cache

- A blocking cache can handle one request at a time. If the request is a miss, the cache waits for the next level memory to supply the value. It is blocked until then.
- A non-blocking cache can work on more than one requests, while waiting for the memory to supply previous misses.
- The L2 cache of Pentium II processor can pipeline four (4) requests.
- It is necessary to keep track of dependencies for precise exception.

Non-blocking Cache to Reduce Stall on Miss

- The pipeline need not be stalled on a cache miss.
- A data cache miss should not affect the instruction cache fetch.
- In a pipeline with out-of-order completion, a non-blocking cache can supply more than one cache hit during a cache miss - hit under miss.
- This is more beneficial for floating-point operations.

Non-blocking Cache

- The miss penalty is reduced by overlapping memory access with execution of other instructions.
- This makes sense if the processor allows out-of-order instruction execution.

Non-blocking Cache

- Difficult to judge the **impact of any single miss** and hence the calculation of **average memory access time** due to the overlap in CPU stalls.
- In case of **multiple outstanding misses**, it is possible that **more than one misses** are for the **same cache block**.

Prefetching Instruction and Data

- **Prefetch** data and instruction **before the request.**
- Prefetch may be done in the cache or in separate buffer.
- On a cache miss, the processor may **fetch two blocks**, the requested one in the cache, and the next one in the **instruction buffer**.
- On a cache miss the data is first searched in the buffer, if found, it is loaded in the cache and a new block is prefetched.

Compiler-Controlled Data Prefetching

- Compiler may insert **prefetch instruction** to request the data before the need.
- Two essential types - **register prefetch**, data is loaded in a register and **cache prefetch**, data is cached.
- A prefetch instruction may be **faulting** or **non-faulting** i.e. **does** or **does not cause an exception** for virtual address fault and protection violation.

Faulting and Non-faulting Prefetch

- **Non-faulting prefetch** should behave like a **no-op on exception.**
- **A normal load** may be considered as **faulting register prefetch** instruction.
- A prefetch should be **semantically invisible** to the program - **does not change register or memory** and **does not cause memory fault.**
- Many modern processors offer **non-faulting cache prefetch** (non-binding prefetch).

Non-Faulting Cache Prefetch

- The **execution is overlapped** with **data prefetching in cache.**
- The **cache should be non-blocking**.
- Loops are important target for prefetching.
- For small miss penalty, **compiler unrolls the loop once or twice**, and **schedules prefetch with execution.**
- Unnecessary prefetch is to be avoided as it increases the instruction cost.

Compiler Prefetching: An Example

```
for(i=0; i<3; ++i)
   for(j=0; j<100; ++j)
        a[i][j] = b[j][0] * b[j+1][0] ;
```

- **8-KB direct-mapped write-back and write-allocate cache with 16-byte block.**
- **Each element of array takes 8-bytes.**
- **The array are a[3][100] and b[101][3].**

Compiler Prefetching: An Example

- Determine the data access that are likely to cause cache miss.
- Insert prefetch instructions to reduce miss.
- Calculate the number of pefetch instructions executed and misses avoided.

Analysis

- **The array a gets the benefit of spatial locality - accessed along the row.**
- **The values of j runs through - 0, 1, 3, 4, $\cdots$, 99. The sequence of hit/miss is - m, h, m, h, $\cdots$ Miss for even value of j and hit for odd value of j.**
- **Misses are** $3 \times \frac{100}{2} = 150$.

Analysis

- **The array b does not get the benefit of spatial locality - accessed along the column.**
- **But it gets the benefit of temporal locality - same element is accessed for every value of i.**
- **If we ignore the conflict misses, there are 101 misses for b - the value of j runs through - (0,1), (1,2), (2,3), $\cdots$, (99, 100) - (m,m), (h, m), $\cdots$, (h, m).**
- **Total misses are $150 + 101 = 251$.**

Analysis

To make life simple we assume the following.

- No prefetching for the first few accesses in the loop.
- We shall not bother about **last few useless prefetches** (assuming no fault).
- Miss penalty is large and we start prefetching seven iterations in advance.

Analysis

```
for(j=0; j<100; ++j) {
       prefetch(b[j+7][0]) ;
       prefetch(a[0][j+7]) ;
       a[0][j] = b[j][0] * b[j+1][0] ;
}

for(i=1; i<3; ++i)
   for(j=0; j<100; ++j) {
      prefetch(a[i][j+7]) ; //  allocate
      a[i][j] = b[j][0] * b[j+1][0] ;
}
```

Analysis

- **7 misses:** `b[0][0], ···, b[6][0].`
- $\lceil \frac{7}{2} \rceil$: `a[0][0], ···, a[0][6].`
- $\lceil \frac{7}{2} \rceil$: `a[1][0], ···, a[1][6].`
- $\lceil \frac{7}{2} \rceil$: `a[2][0], ···, a[2][6].`

We have reduced $251 - 19 = 232$ misses at the cost of $2 \times 100 + 2 \times 100 = 400$ prefetch instructions. Does it make sense?

Example Extended

- Ignore **instruction cache misses**; also ignore **conflict** and **capacity misses** of the data cache.
- Prefetch can overlap with each other and with cache miss.
- The **original loop takes 7 cycles per iteration.**
- The **first prefetch loop takes 9 cycles per iterations**.
- **The second prefetch loop takes 8 cycles per iterations.**
- A miss takes 100 cycles.

Example Extended

- **The original loop takes** $300 \times 7 + 251 \times 100 = 2100 + 25100 = 27200$ **cycles.**
- **The modified code takes** $100 \times 9 + 200 \times 8 + 19 \times 100 = 900 + 1600 + 1900 = 4400$ **cycles.**

Prefetch `a[i][j+7]`

- **The only purpose of prefetching `a[i][j+7]` is to pre-allocate a cache block for the next write. But then actual data read is not required.**
- **Some processor support write hint instruction and can be used in place of prefetch. But then the whole cache block is to be written.**

Reducing Cache Hit Time

- **Cache hit time may limit the clock rate of a processor.**

Cache Hit Time

- Cache access time is mostly taken by **accessing the tag** using the index portion of the address and **comparing it**.
- The access is certainly **faster for a smaller** and **simpler (direct mapped) cache**.
- The **tag checking** can be **overlapped** with **transmission of data**.
- The L1 cache is often direct access and smaller in size - Pentium 4 has smaller L1 data cache (8KB) than Pentium III (16KB) for faster clock.

Other Hit Time Reduction Techniques

- Cache in the vartual address space - no virtual to physical address translation for cache hit.
- Pipelined cache access - effective latency is multiple cycles, fast cycle time and slow hits.
- Trace cache.

Physical to Virtual Cache

- Normally cache uses **physical address** for **indexing** and **tag**.
- **Virtual** to **physical** address translation is to be completed before cache access.
- Can the cache be in the **virtual space** i.e. the cache is accessed using the virtual address?
- Then in case of **hit** (which is more common), no address translation is required.

Physical to Virtual Cache

- In case of a miss, the virtual address will be translated and the data will be loaded from the physical memory but the tag of the cache will be that of the virtual address.
- What are the problems?

Virtual Cache

- How will the **protection** be enforced?
- What will happen at the time of **context-switch**? After all two different processes has **common virtual spaces.**
- What will happen if **two different virtual spaces** (kernel and user) are **mapped to the same physical address.** There may be two copies of the same data in the cache. This is called **synonyms** or **aliases** problem.
- IO typically uses physical address.

Virtual Cache

- Protection information may be **copied from the TLB** and **checked at every access to the cache.**
- The cache may be **flushed at every context-switch.** The other alternative is to introduce a **process-identifier tag** along with the address tag.
- The aliasing problem may have hardware or software solution.

Synonym Problem: Antialiasing

- It guarantee that every cache block is a unique physical address.
- As an example - Alpha 21264 uses a 64KB instruction cache with an 8KB page size and 2-way associativity. There are 4 sets indexed by two virtual address bits.
- If there is a miss, the hardware checks all eight (8) possible locations. If there is already one, it is invalidated and only one data block is loaded.

Synonym Problem: Software Solution

- Software forces all aliases to share some address bits!
- As an example - UNIX for Sun Microsystem - all aliases are identical in last 18-bits - this is called page colouring.
- Let there be a direct-mapped cache of size 256KB (2^{18}).
- If there are two different virtual addresses pointing to the same physical address, their last 18-bits are identical hence will point to the same cache block.

Pipelined Cache Access

- The cache access may be pipelined and the latency of the L1 cache access is more than one clock cycle.
- L1 cache access in Pentium was one (1) clock cycle, it is two (2) clock cycles in Pentium III and four (4) clock cycles in Pentium 4.
- This increases the number of pipe stages.

Pipelined Cache Access

- Increase in the number of pipe stages increases the penalty of the mis-predicted branches, increases the load latency.
- But finally it increases the instruction bandwidth.

Trace Cache

- Additional factors in **instruction fetch bandwidth** - other than **I-cache hit rate** and **branch prediction accuracy**.
- **Branch throughput** - if **one branch is predicted per cycle**, the instruction fetch window changes at the rate of **one basic block**. The window size will be larger if **more than one branches can be predicted per cycle.**

Trace Cache

- Noncontiguous instruction alignment - due to the presence of branches and jumps, the instructions be fetched during a given cycle are not available in contiguous memory locations and therefore also not available in contiguous cache blocks.

  It is not enough that instructions are present in the cache, they must be accessible in parallel.

Trace Cache

- **Fetch unit latency - there is a startup cost of instruction fetch after a branch is mispredicted or a jump. Higher branch throughput and noncontiguous instruction alignment will increase the fetch unit latency.**

Trace Cache

- Traditionally a cache block stores a sequence of instructions from a program (static) that are local in space.
- The trace cache stores the dynamic sequence of executed instructions into a cache block.
- The content is determined not by the memory but by the CPU.

Trace Cache

- The branch prediction is part of the trace cache.
- It has more complex address mapping.
- Used in Pentium 4.

DRAM

Pronounced DeeRam - dynamic random access memory used used in most personal computers.

EDO DRAM

Extended Data Out Dynamic Random Access Memory, a type of dynamic RAM that is faster than conventional DRAM. A conventional DRAM can access one block of data at a time. EDO RAM can start the fetch of the next data block at he same time it sends the previous block to the CPU.

BEDO DRAM

Burst EDO DRAM can process four memory addresses in one burst. But a BEDO DRAM can only stay synchronized with the CPU clock for short periods (bursts). It can't work with a processor whose buses run faster than 66 MHz.

SDRAM

Synchronous DRAM is a type of DRAM that can run at a higher clock speed than conventional DRAM. It can synchronize itself with the CPU's bus and can run at 133 MHz.

RDRAM

Rambus DRAM is developed by Rambus, Inc. One of the fastest available memory technologies used by PCs. It can transfers data at up to 800 MHz. Intel licensed the Rambus technology for use on its motherboards. An alternative memory architecture called SyncLink DRAM (SLDRAM). Intel and Rambus are working a new version of RDRAM, called nDRAM, that will support data transfer speeds at up to 1,600 MHz.

SLDRAM

Synchronous Link DRAM developed by the SyncLink Consortium. SLDRAM was intended to be an enhanced version of SDRAM that used a multiplexed bus to transfer data to and from chips. It was expected to be an alternate to RDRAM as a PC memory, but has never gained good acceptance.

DDR-SDRAM

Double Data Rate-Synchronous DRAM supports data transfers on both edges of each clock cycle. Thus effectively doubles the data throughput. It also consumes less power, which makes it well-suited for power sensitive devices. DDR-SDRAM is also called SDRAM II. and DDRAM.