Memory access timing

Michal Sojka
ČVUT v Praze, FEL

sojkam1 (at) fel.cvut.cz

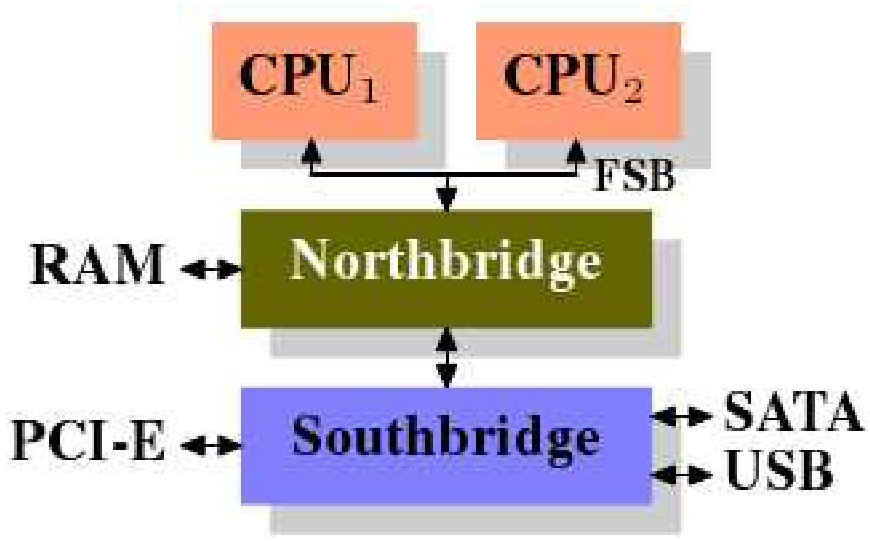
24. 10. 2012

Introduction

- In the early days computers were much simpler.
  - All parts had quite balanced performance
- Today, basic architecture is stabilized and individual subsystems are being optimized.
- Mass storage and memories were improved slower than others (CPU) because of their higher cost.
- Slowness of mass storage subsystems has been solved by caching data in memory.
- Removing the main memory as a bottleneck has proven much more difficult and almost all solutions require changes to the hardware.

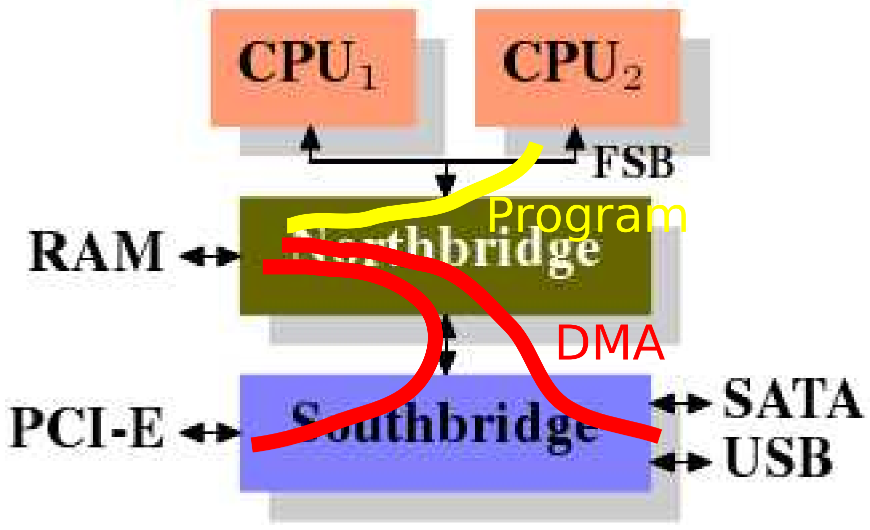
Yesterday's Hardware

- Multicore and multi-processor systems are quite common
- The Northbridge contains the memory controller
  - Its implementation determines the type of RAM chips used for the computer
  - Different types of RAM, such as DRAM, Rambus, and SDRAM, require different memory controllers.
- Southbridge (I/O bridge)
  - Connects other peripherals

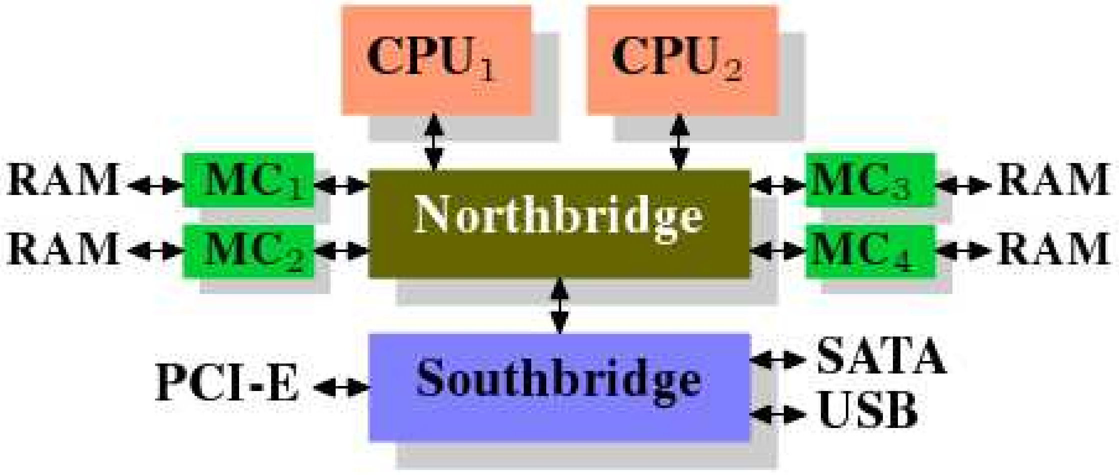


Properties of this architecture

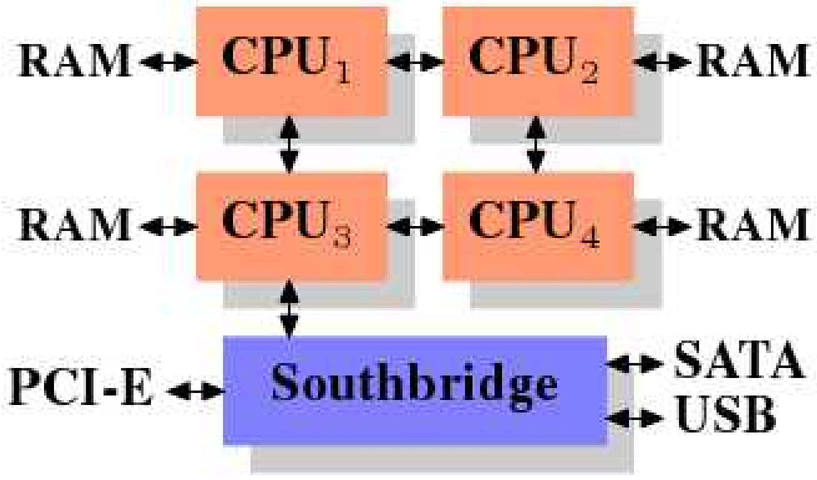
- All data communication from one CPU to another must travel over the same bus used to communicate with the Northbridge.
- All communication with RAM must pass through the Northbridge.
- The RAM has only a single port.
- Communication between a CPU and a device attached to the Southbridge is routed through the Northbridge.
- Bus between Northbridge and memory is clear **bottleneck**



Increasing Memory Bandwidth



External Memory controllers



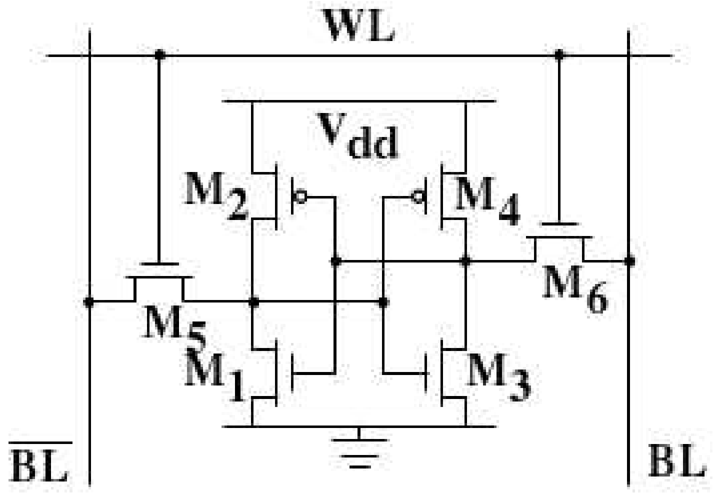
Non-Uniform Memory Architecture (NUMA)

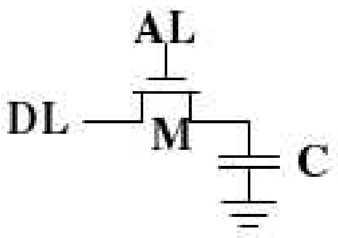
AMD Opteron
Intel Core i7

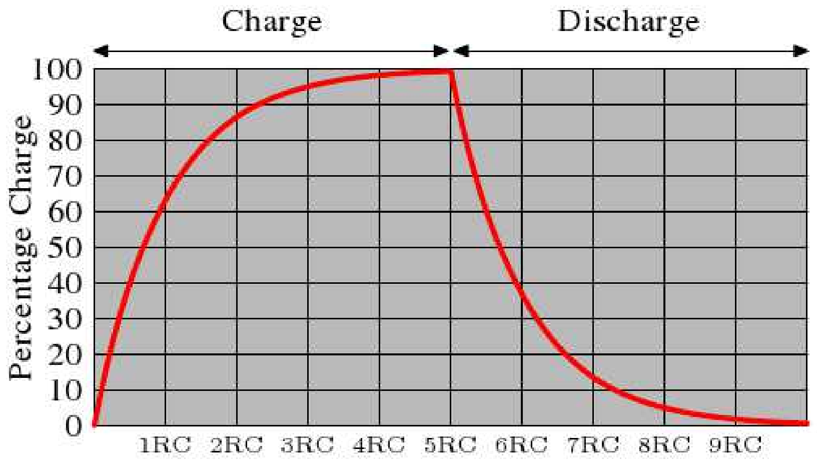
- NUMA will be more and more common in future

RAM Types

- Static RAM (SRAM)
  - Fast, expensive
  - 6 transistors per bit
- Dynamic RAM (DRAM)
  - Capacitor charge must be refreshed
  - Reading discharge capacitor (write after read)
  - (Dis)Charging is not instantaneous
  - Compromise: capacity/size/power consumption

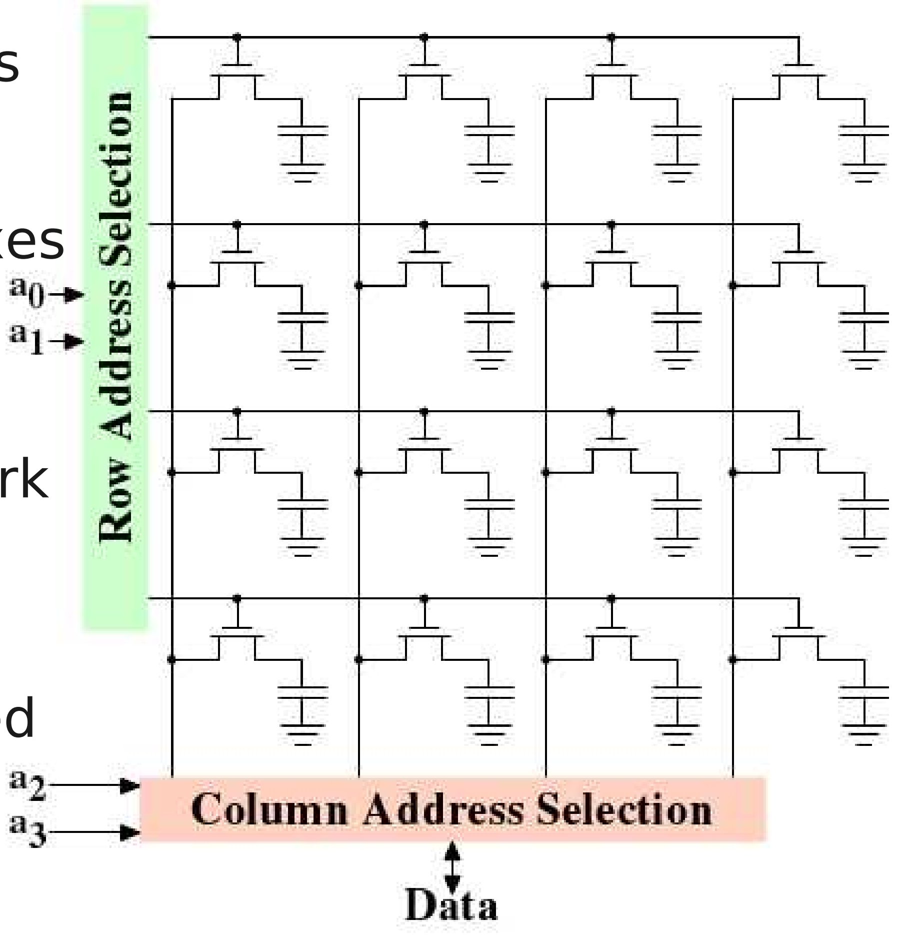






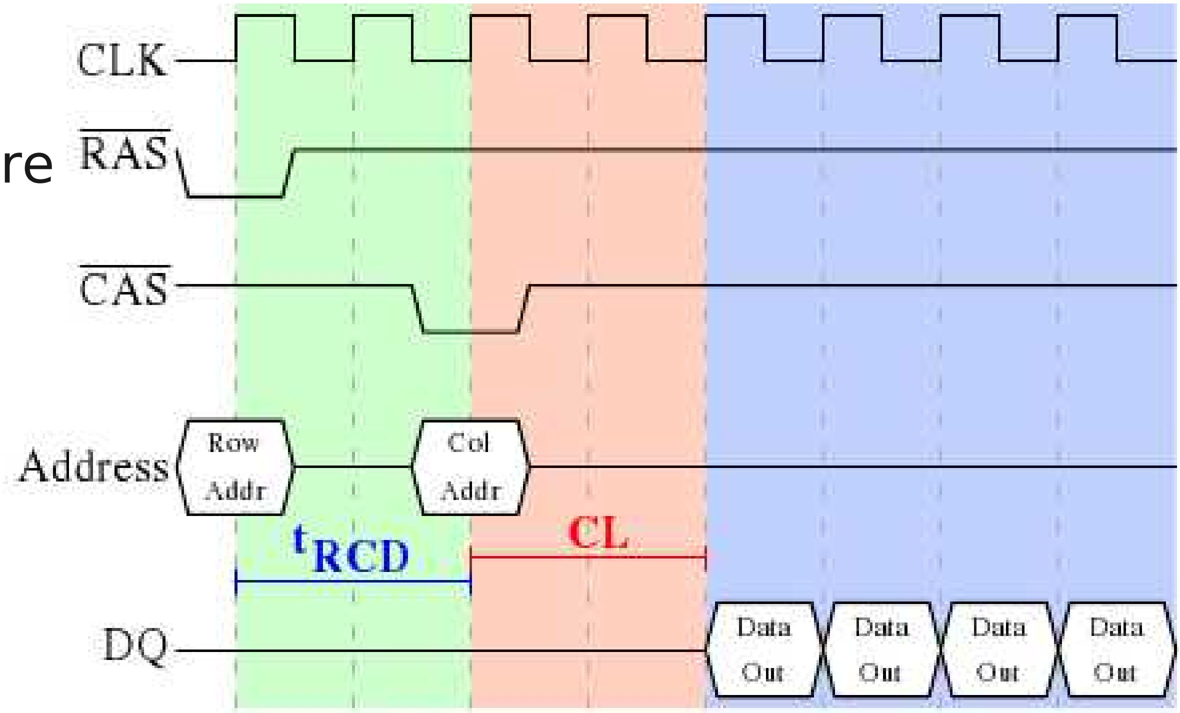
DRAM Access

- Addressing individual cells is impractical (many wires)
- Chip is organized in rows and columns (and banks), address is multiplexed
- In the chip, row and column multiplexes select the proper lines according to address bits
- Happens parallel in many chips to work with the whole data word (64 bits)
- Writing: New value is put on data, stored when RAS and CAS are selected
    - It takes some time to charge the capacitors

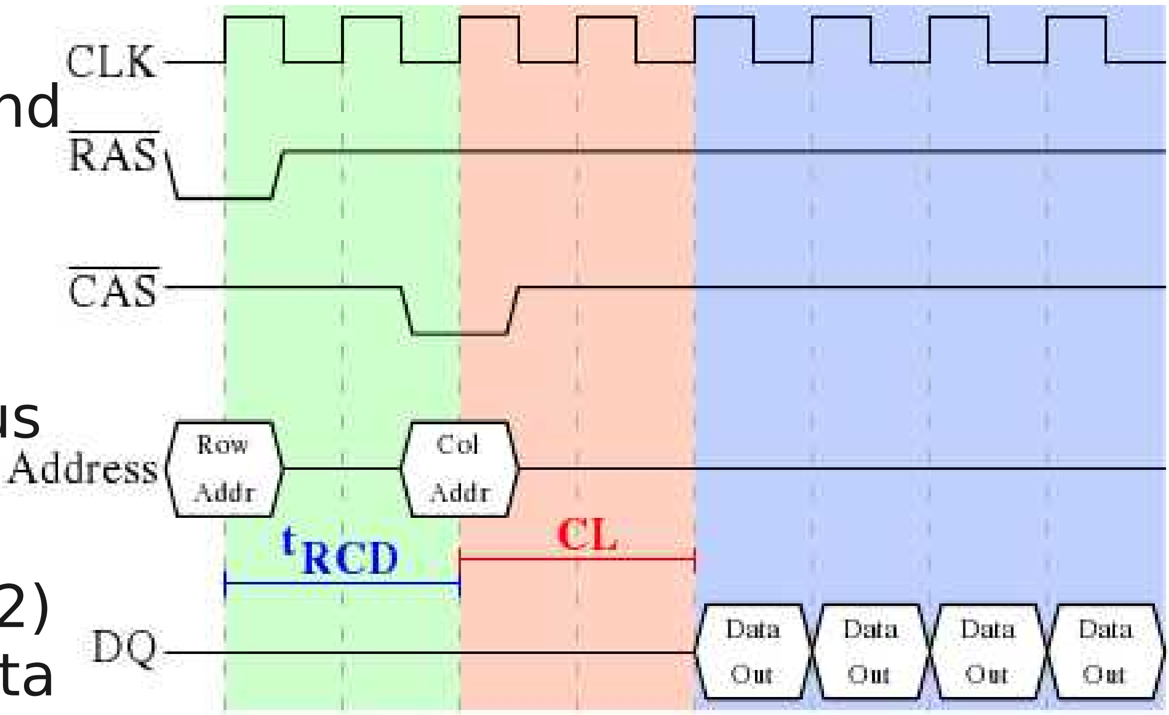


DRAM Access Details

- Access protocol is synchronous (SDRAM) – there is a clock signal,
- CLK provided by memory controller (FSB frequency – typ. 800 – 1600 MHz)
  - Double/Quad-pumped
- Max. speed: 64 bit * 4*200 Mhz = 6,4 GB/s
  - Not reachable in reality

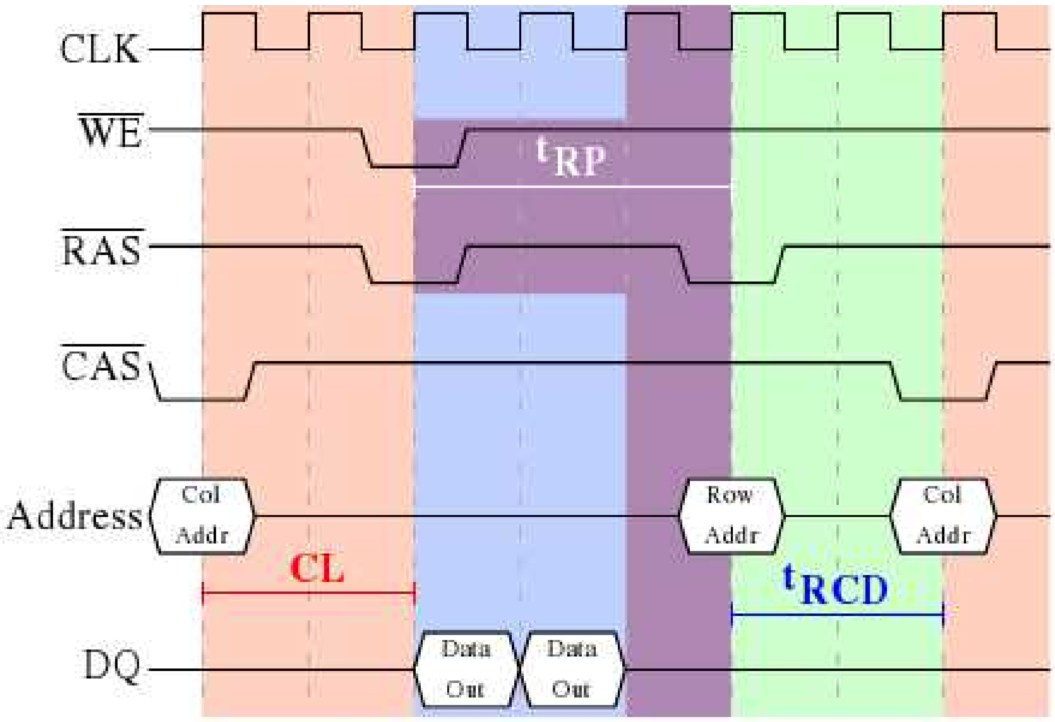


DRAM Access Details – Read

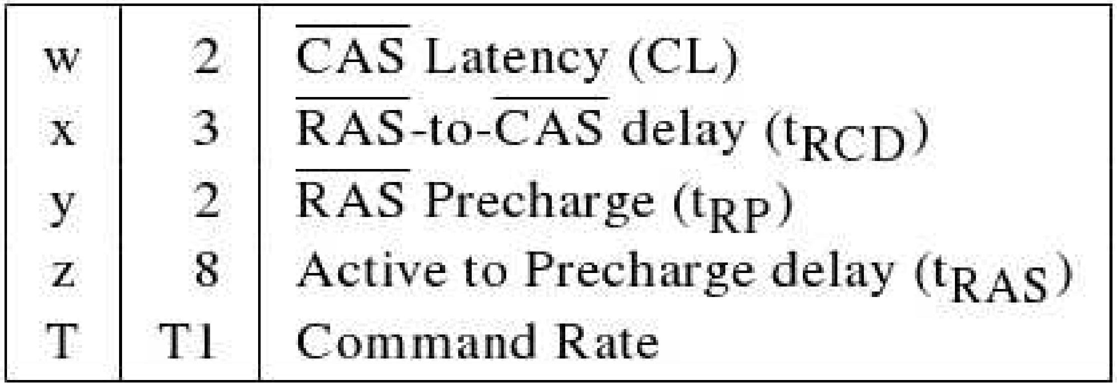


- Put row address on bus and signal \RAS
- Wait RAS-to-CAS clock
- Put column address on bus and signal \CAS
- Wait CAS latency clocks (2) to let the chip prepare data
- Read data
  - Reading only one word is wasting.
  - It is possible to specify read of multiple subsequent words
  - We can also specify only new CAS and keep RAS the same
- => Sequential access is faster than random one

Precharge and Activation



- Additional delays
  - \WE to \RAS (RAS precharge)
  - Time between subsequent \RAS signals
- Notation for SDRAM chips w-x-y-z-T (2-3-2-8-T1)

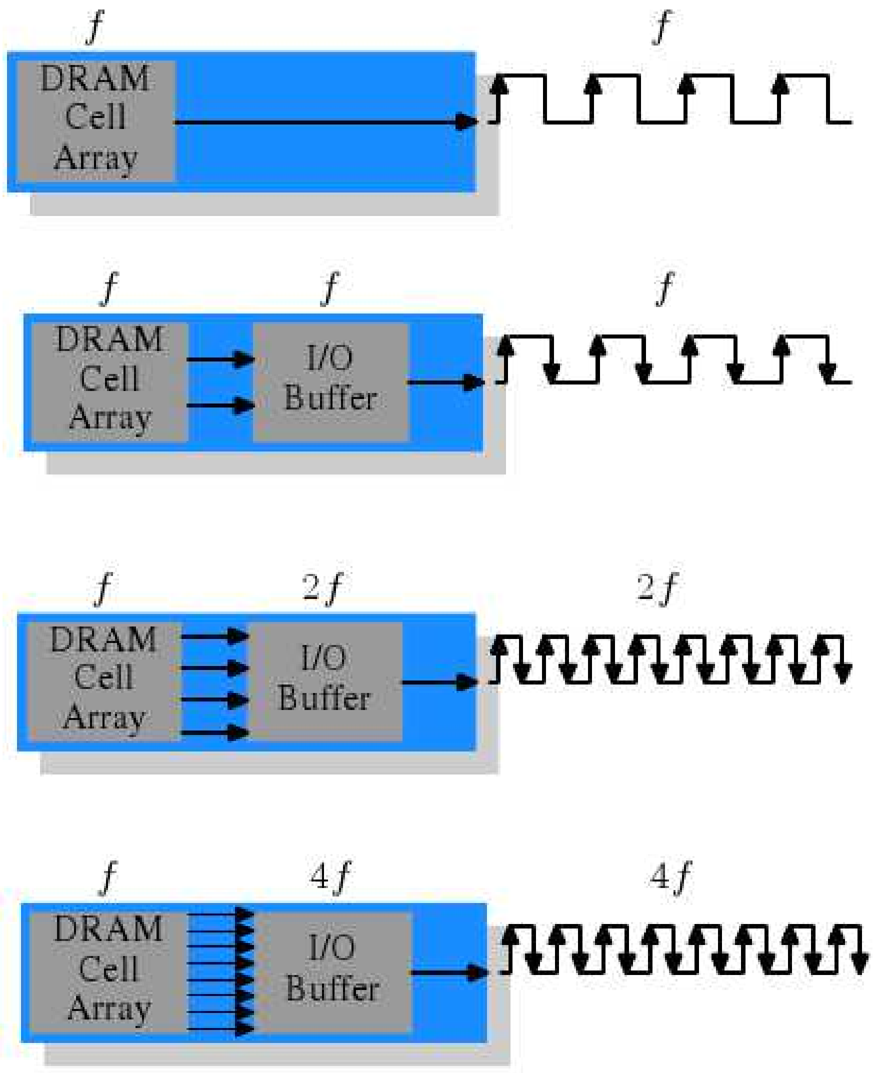


w	2	$\overline{\text{CAS}}$ Latency (CL)
x	3	$\overline{\text{RAS}}$-to-$\overline{\text{CAS}}$ delay (t_{RCD})
y	2	$\overline{\text{RAS}}$ Precharge (t_{RP})
z	8	Active to Precharge delay (t_{RAS})
T	T1	Command Rate

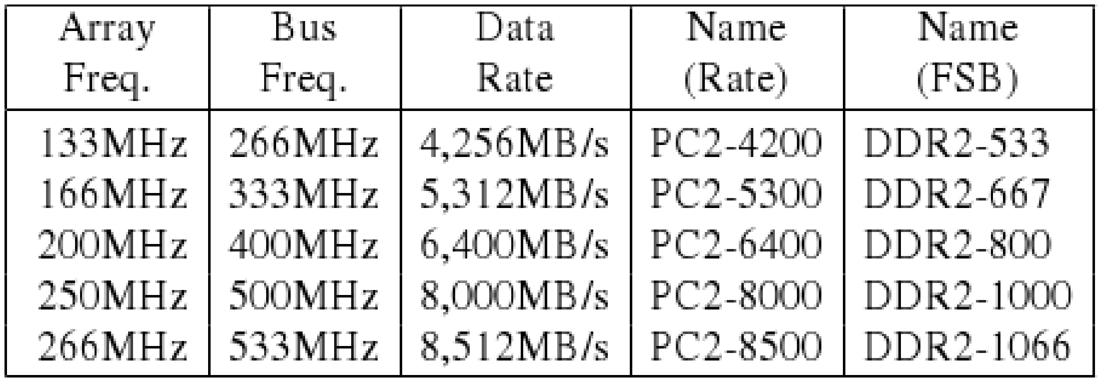
DRAM refreshing

- Completely transparent to the rest of the system
- Row cannot be accessed if it is refreshed
- Every cell needs to be refreshed every 64 ms
    - For 8192 rows, refresh happen every 7,8 µs

Memory types



- Single data rate
- Double data rate (DDR)
  - Two bits per \CAS
  - Buffer
  - 100 MHz * 64 bit * 2 = 1600 MB/s



Array Freq.	Bus Freq.	Data Rate	Name (Rate)	Name (FSB)
133MHz	266MHz	4,256MB/s	PC2-4200	DDR2-533
166MHz	333MHz	5,312MB/s	PC2-5300	DDR2-667
200MHz	400MHz	6,400MB/s	PC2-6400	DDR2-800
250MHz	500MHz	8,000MB/s	PC2-8000	DDR2-1000
266MHz	533MHz	8,512MB/s	PC2-8500	DDR2-1066

Speed comparison

- Intel Core 2 running at 2.933 GHz
- FSB 1,066 GHz (Quad-pumped)
- Ratio: 1:11
  - One clock stall on memory bus wastes 11 processor cycles
  - Processor with 3-way superscalar architecture wastes up to 33 instructions
  - Memory stalls can be longer than one clock => hundreds instructions wasted
- Sequentially accessed data:
  - Not so bad. Entire row without any stall.
  - DDR3-1600 can supply sequential data at 12,8 GB/s

Other main memory users

- High performance cards
  - Network, mass storage, USB
- Graphical systems with shared memory
  - 1024x768, 16bpp, 60 Hz => 94 MB/s
  - Intel chips can compress the image in memory
    - Less bandwidth for
      - statical images (not movies)
      - Large continuous areas (lines) of one color (text on white background, no gradients)

Real World Example

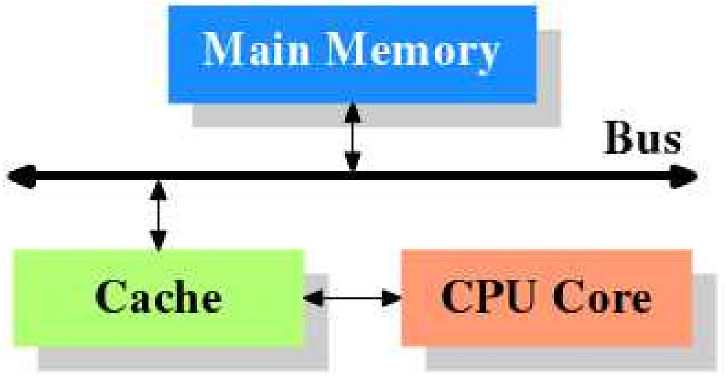
- lspci

```
00:00.0 Host bridge [0600]: Intel Corporation Core Processor DMI [8086:d131] (rev 11)
        Subsystem: ASUSTeK Computer Inc. Device [1043:8383]
```

- CPU datasheet
- DRAM datasheet

Caches

- 25 years ago, RAM access was almost as fast as registers
- RAM chips didn't increase their performance as much as processors
- It is possible to build fast RAMs, but it is not economic
- Big main memory (DRAM) and smaller but faster caches (SRAM).
- Cache is transparent to the programmer
- However, from performance point of view it cannot be seen as an array of bytes but instead as an **array of cache lines**!



Caches II.

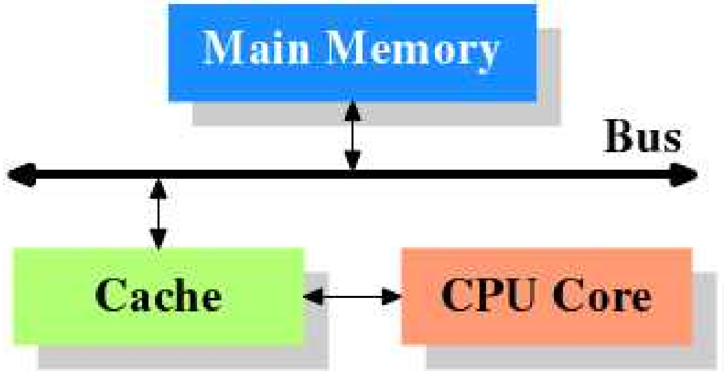
- Terminology
  - *Spatial locality*: accessed memory objects are close to each other
    - Code: inner loops
    - Data: structures (reading of one field is often followed by of other files)
  - *Temporal locality*: The same data will be used multiple times in a short period of time
    - Code: loops
    - Data: e.g. Digital filter coefficients are accessed every sampling period
- Efficiency – access to memory: 200 clocks, to cache: 15 clocks
  Accessing 100 data elements 100 times:
  - Without cache: 100*100*200 = 2 000 000 cycles
  - With cache: 1*100*200 + 99*100*15 = 168 500 cycles (91% better)

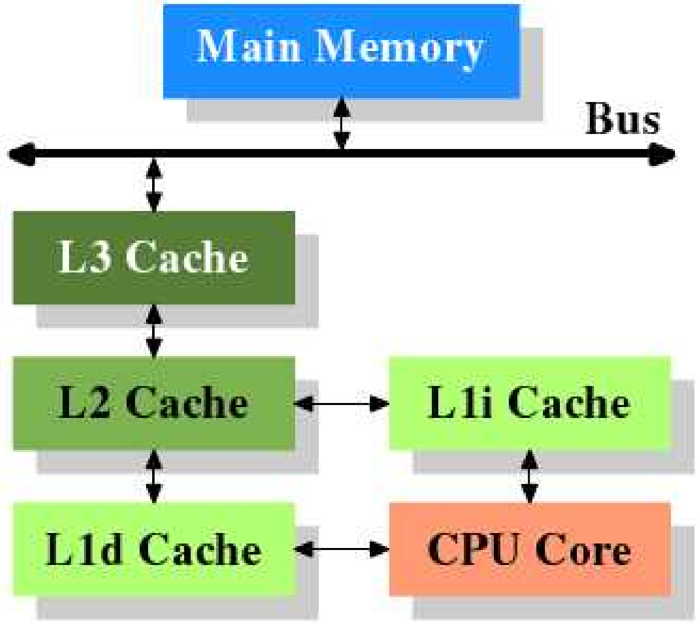
Caches III.

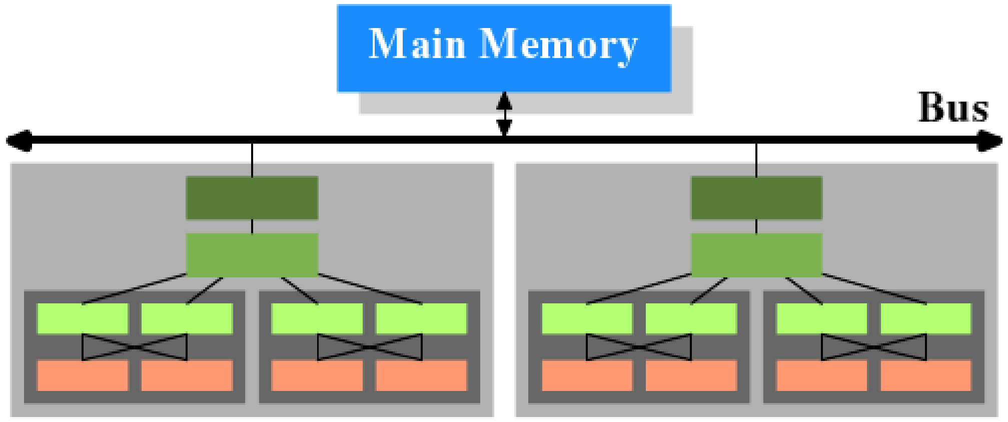
- Size of caches
  - Typically 1/1000 of main memory
  - 4 MB cache / 4 GB main memory
- Working sets are often larger than cache memory
  - True color image 1024x768 = 3,1 MB; source + destination memory: 6,2 MB
  - Good strategy what should be cached at any given time
    - Determined by hardware
    - Programmer may help a lot

CPU Caches in the Big Picture

- All loads/stores go through cache
- CPU <-> Cache: fast connection
- Cache <-> Main memory: FSB Bus + North bridge
- It is advantage to have separate caches for **instructions** and **data**





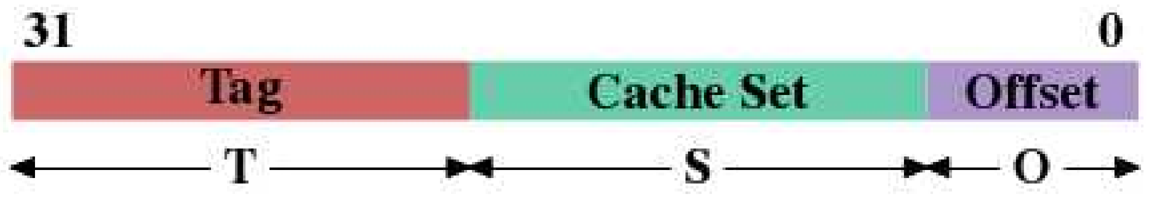


Multi processor, multi-core, multi-thread

Cache Operation at High Level

- By default read or written data are stored in the cache
- If CPU needs a data word, caches are searched first
    - Each cache entry is **tagged** using the address of the data word in the main memory
    - The address can be either physical or virtual
- Storing the tag requires memory space (typ. 32 bits)
    - There is one tag for larger block of memory (**cache line**)
    - It is efficient to read/write the whole cache line from/to DRAM
    - Space locality is the main principle of caching
    - Typical cache line size: 64 bytes (32 bytes earlier)

Cache Operation at High Level II.



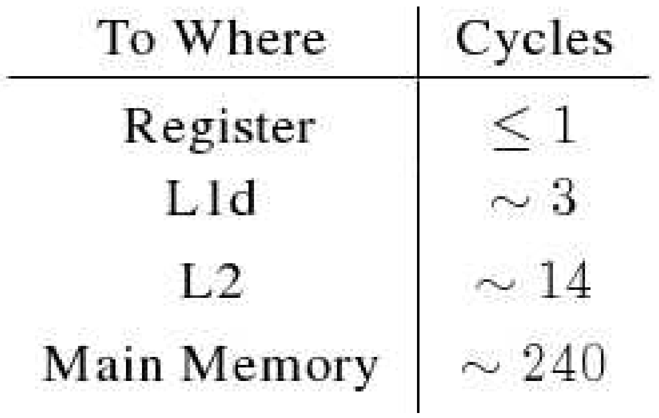
- When memory is needed the entire cache line is loaded into the L1d
- The memory address for each cache line is computed as:
  - Masking the address according to cache line size (64 -> 6 bits) These bits are discarded and used as offset to cache line
  - Remaining bits are used to locate the cache line
  - The address is split to 3 parts
    - Cache line size equals to 2^O
    - There are 2^S of sets of cache-lines
    - Tag has the size 32-S-o

Cache Operation at High Level III.

- When processor modifies memory the processor still has to load a cache line first because no instruction modifies the whole cache line
- Cache line which has been written to is marked as “**dirty**”. This flag is clear once the line is written to the main memory.
- To load a new data in a cache, it is almost always necessary to make room in the cache
  - Eviction from L1d pushes cache line down into L2
  - This means the room has to be made in L2 (*exclusive cache*)
  - The cache line is pushed to L3 and again some L3 might be freed by writing it to main memory.
  - Intel uses *inclusive cache* where L1d is also present in L2 and evicting from L1d is much faster.

Cache Operation at High Level IV.

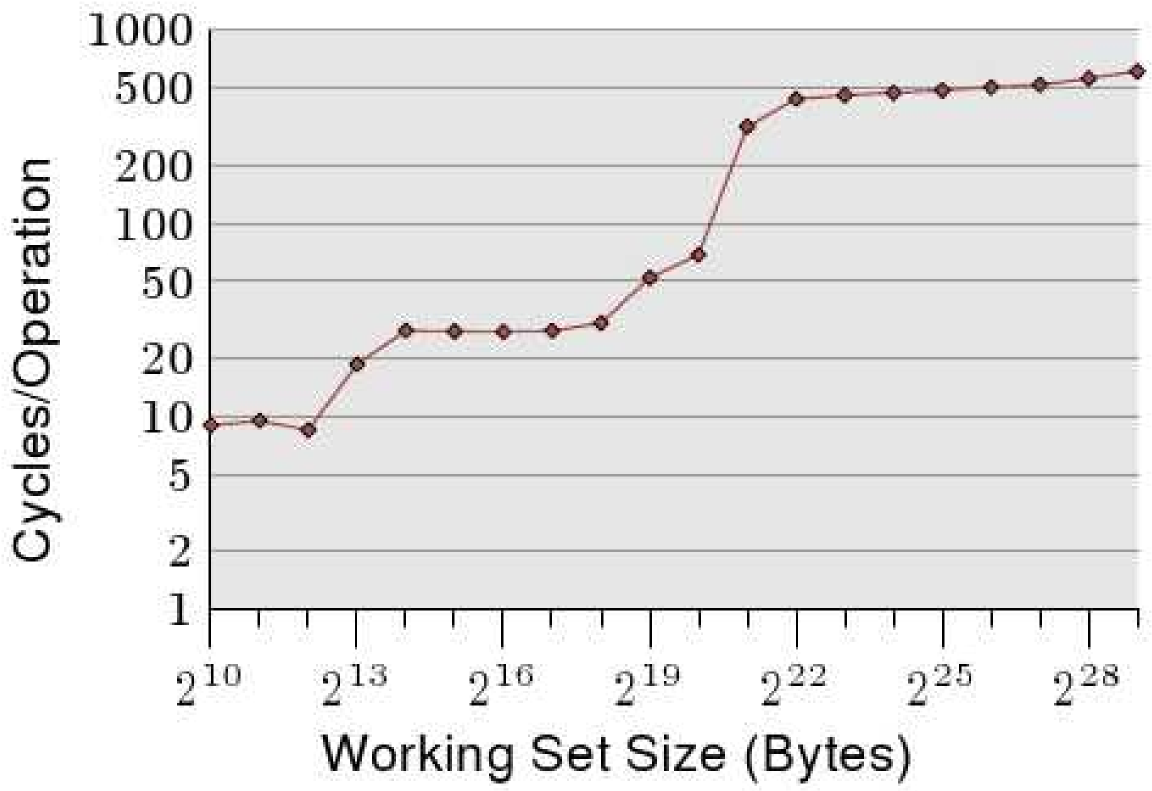
- In symmetric multi-processor (SMP) systems, caches of the CPUs cannot work independently from each other.
  - The maintaining of uniform view of memory for all processor is called "**cache coherency**"
  - If some processor writes to a cache line, other processors have to clean the corresponding cache line from their caches.
  - Cache synchronization protocol: MESI
    - A dirty cache line is not present in any other processor's cache.
    - Clean copies of the same cache line can reside in arbitrarily many caches.
- Cost of cache hit/miss (Pentium M):
  - Some delays are often hidden because of instruction pipelining inside of processor



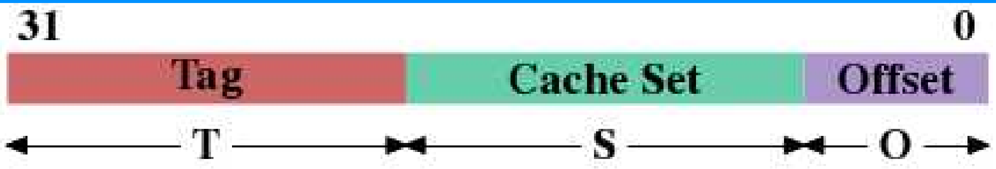
To Where	Cycles
Register	≤ 1
L1d	~ 3
L2	~ 14
Main Memory	~ 240

Access time for random writes

- L1d is 2^{13} bytes (8 kB)
- L2 is 2^{20} (1 MB)

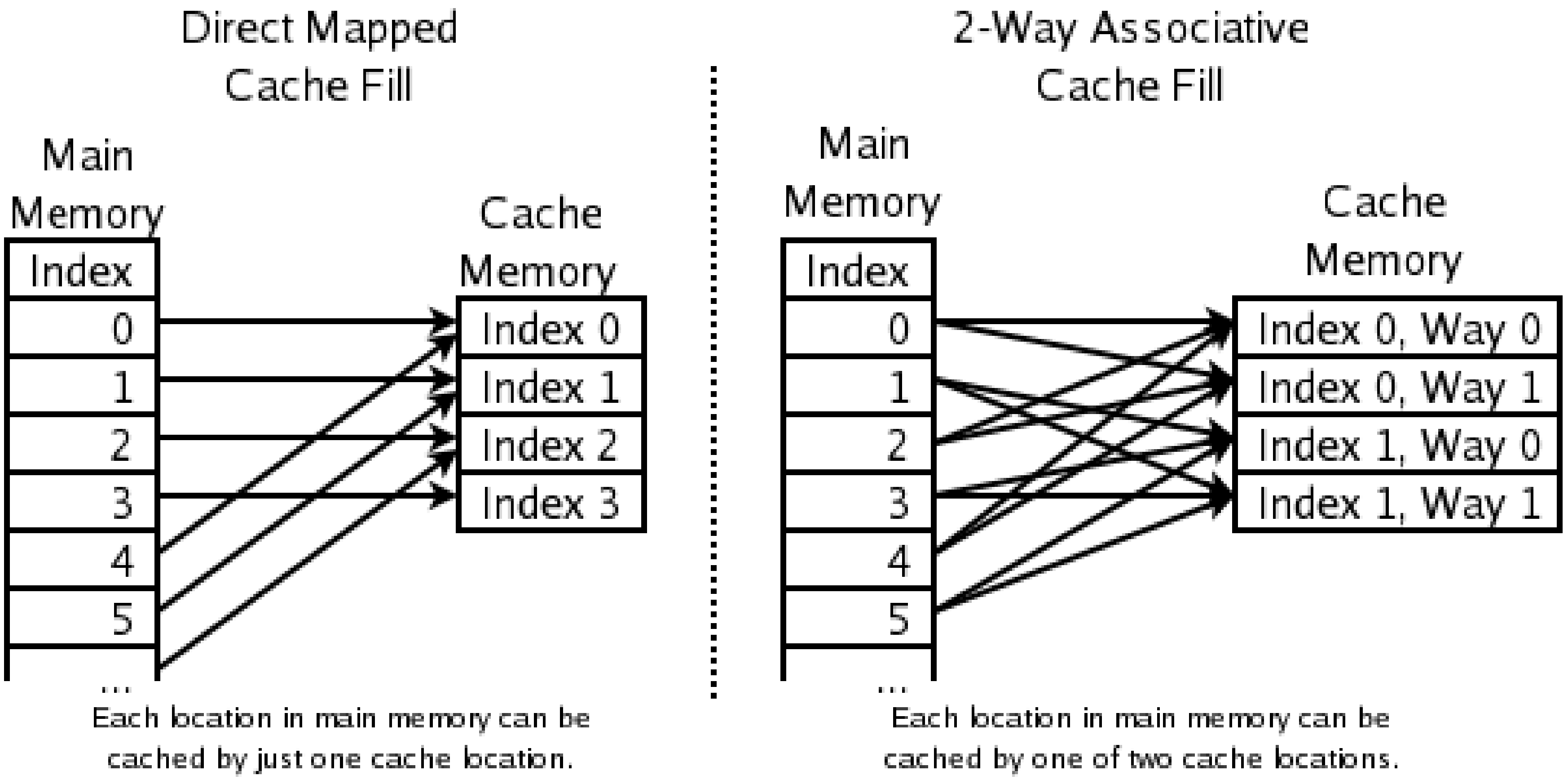


CPU Cache Implementation Details



- Associativity:
  - **Fully associative cache**: each cache line can contain arbitrary memory location
    - Tag is address but offset (no cache set part)
    - Tag comparators are complicated and expensive for large caches
  - **Direct-Mapped cache**: each entry in main memory can go in just one place in the cache
  - **Set associative caches**: compromise
    - Each location can be stored in N cache lines
    - Today
      - L2 24-way associative
      - L1 8-way associative

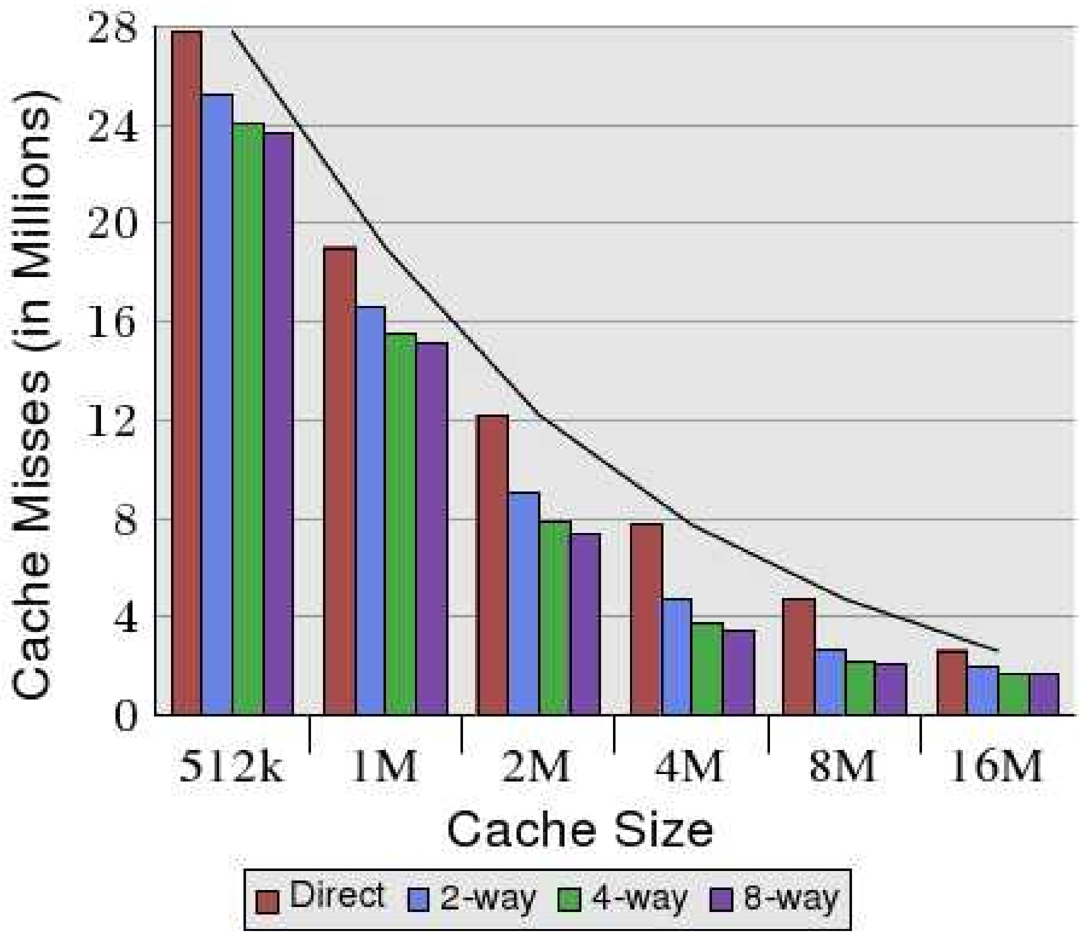
Associativity



Source: Wikipedia

Cache Size vs Associativity (CL=32)

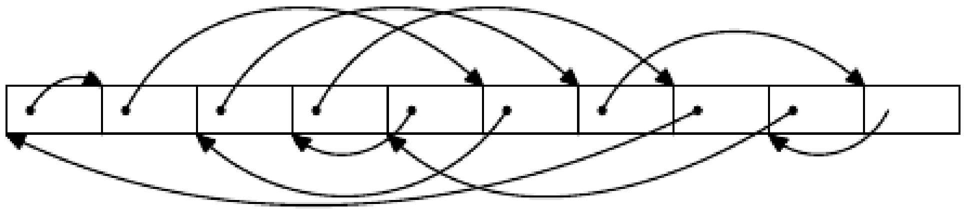
- Cache associativity virtually increase the size of cache
- Number of cache misses in gcc (while compiling a program)

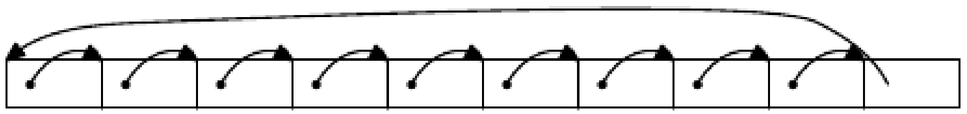


Measuring Cache Effects

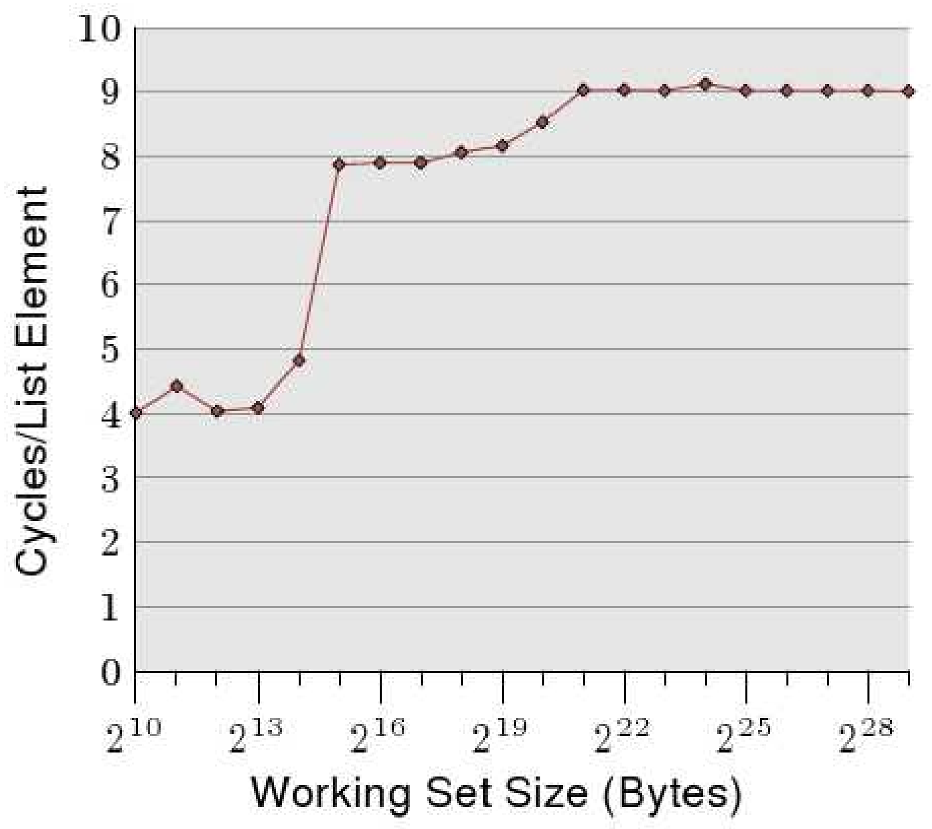
- Linked list is traversed in two ways:
  - Random acces
  - Sequential access

```
struct l {
  struct l *next;
  long int pad[NPAD];
};
```


Random

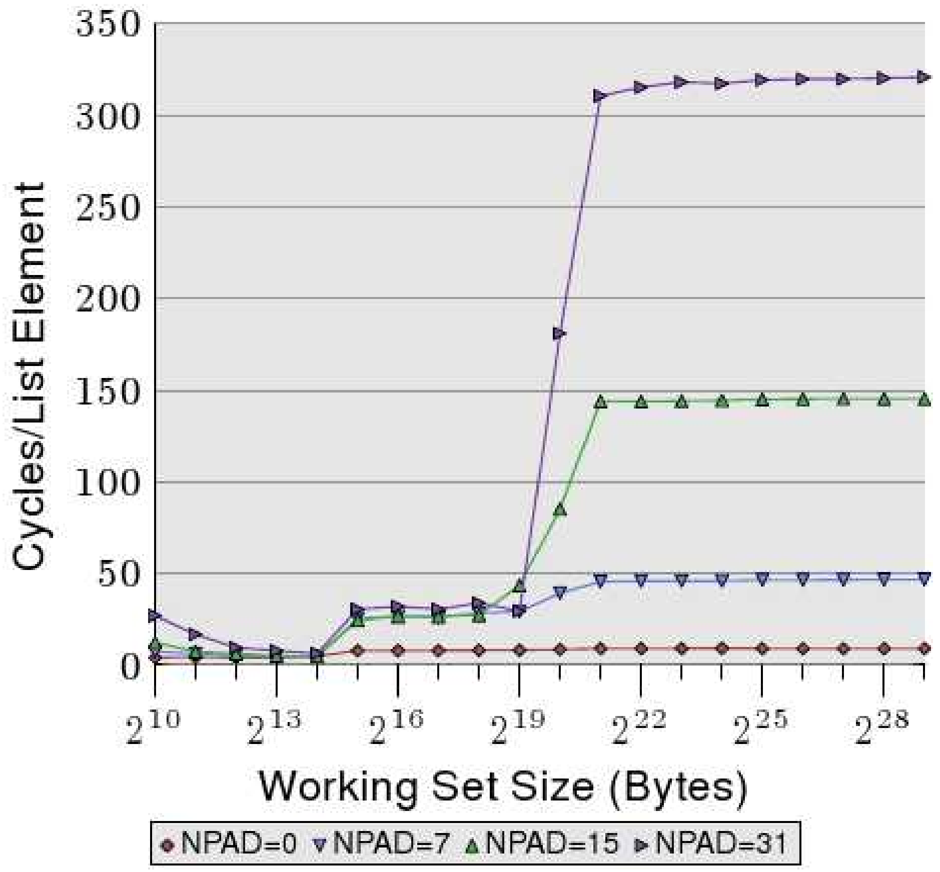

Sequential

Sequential Read Access, NPAD=0



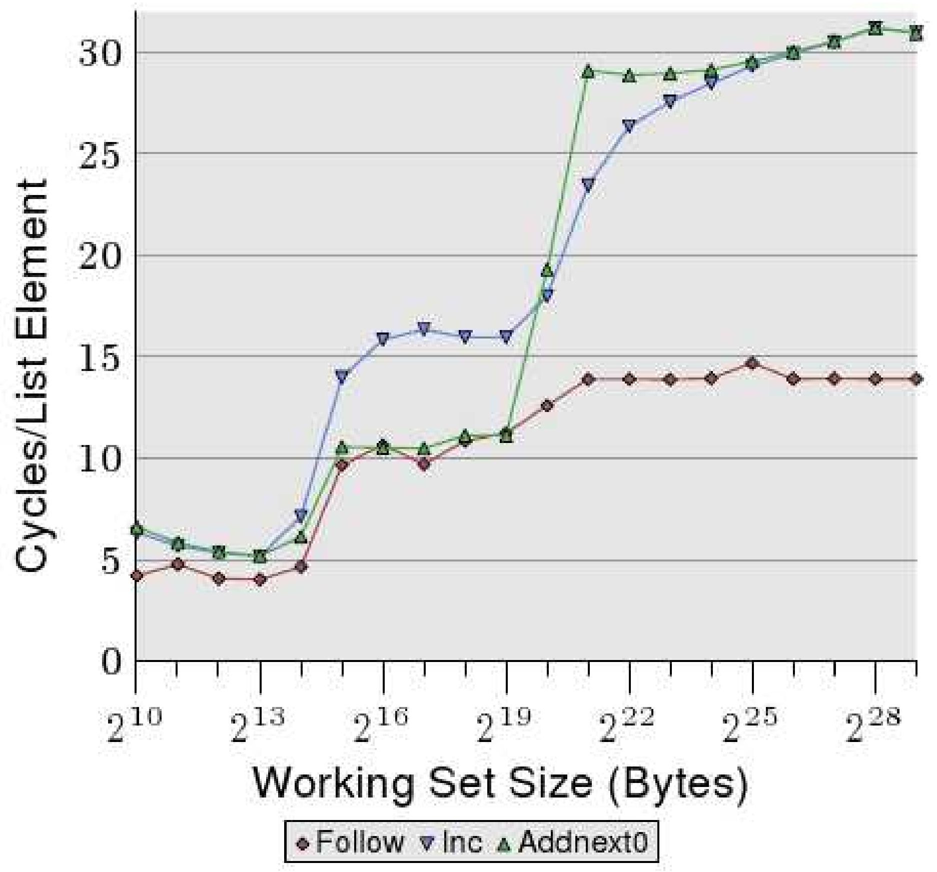
- 16 kB L1d (2^{14} bytes)
- 1 MB L2 (2^{20} bytes)
- No sharp edges:
  - Caches are also used by the rest of the system
  - L2 is used also for instructions
- Why we do not see big change for cache misses for WS > 2^{20}? We have seen times like 200.
  - CPU prefetches next cache line in advance

Sequential Read for Several Sizes



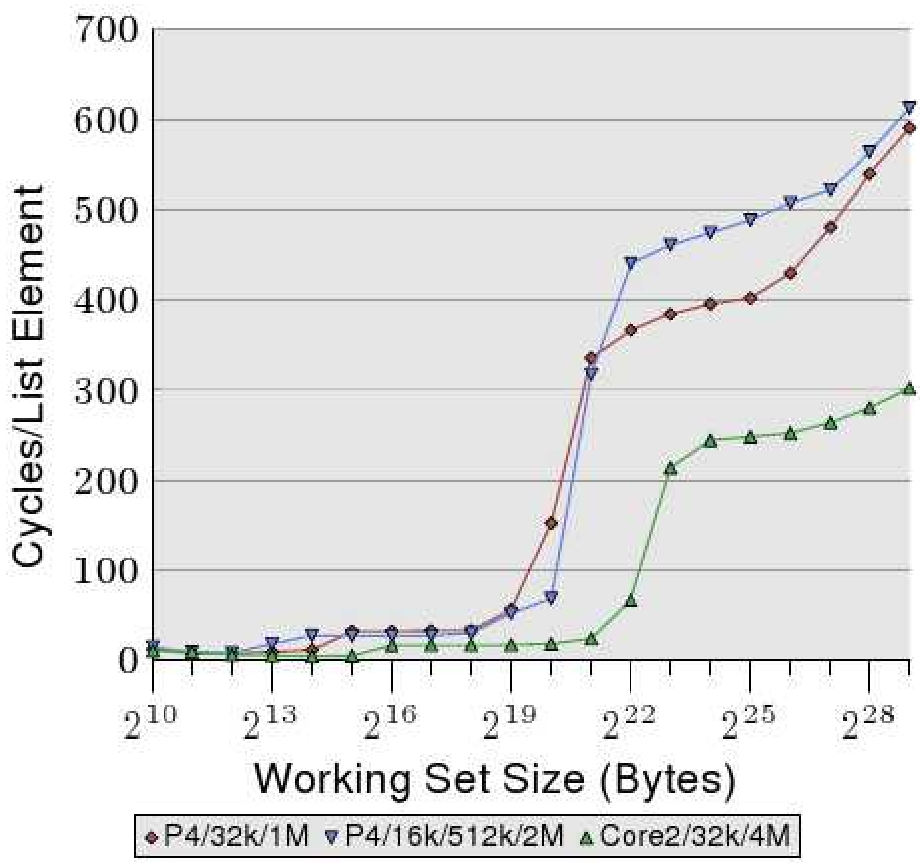
- How to see the effect of prefetching?
- Let's have fewer but larger elements in the list.
- Distance between elements 0, 56, 120, 248 bytes
- Bottom line same as in the previous experiment
- L2 access: 28 cycles (prefetching disabled)
- WS > L2: Prefetcher should recognize some cache lines are not necessary for NPAD=15, 31
- Prefetches do not happen beyond page boundary (4 kB)

Sequential Read and Write, NPAD=1



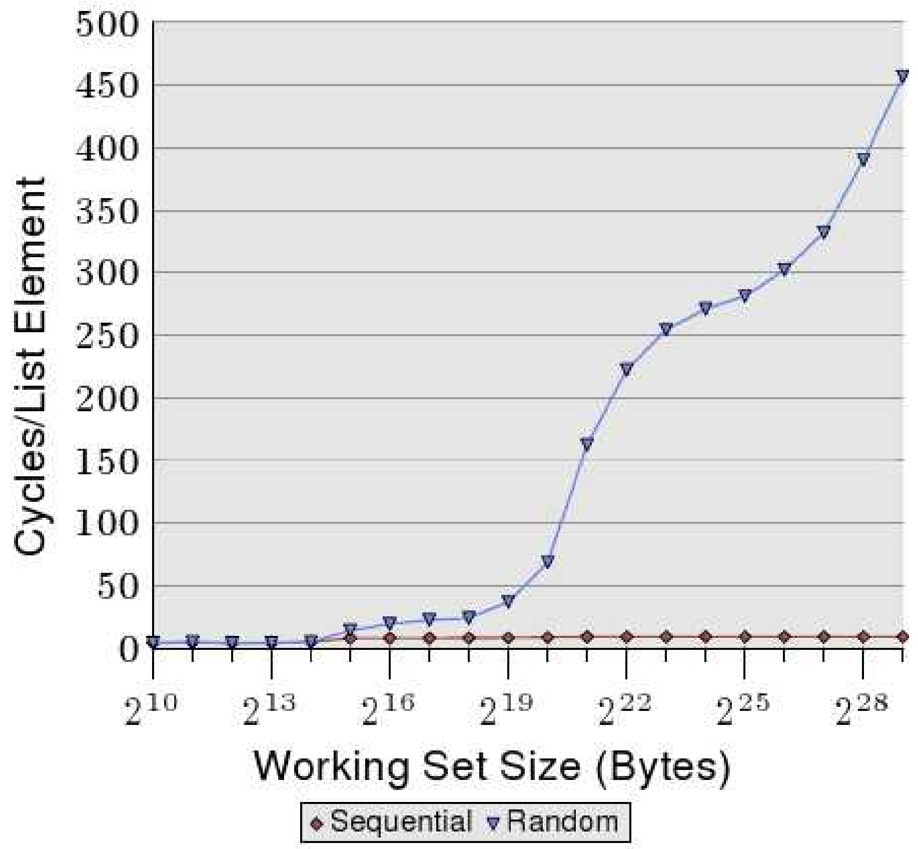
- Element size: 16 bytes
- Inc: p->pad[0]++;
- Addnext0:
  p->pad[0] += p->next->pad[0]
- Addnext0 has more work to do, but is sometimes faster!
  - Reading next actively prefetches data to L1d and these are ready next cycle (see also prefetch instruction)
- Addnext0 runs out of L2 faster. It needs more data from main memory
- For WS=2^{21}: 28 cycles is 2 times longer than Follow test: L2 cache eviction must write a dirty cache from L2 to main memory.

Advantage of Larger L2/L3 caches



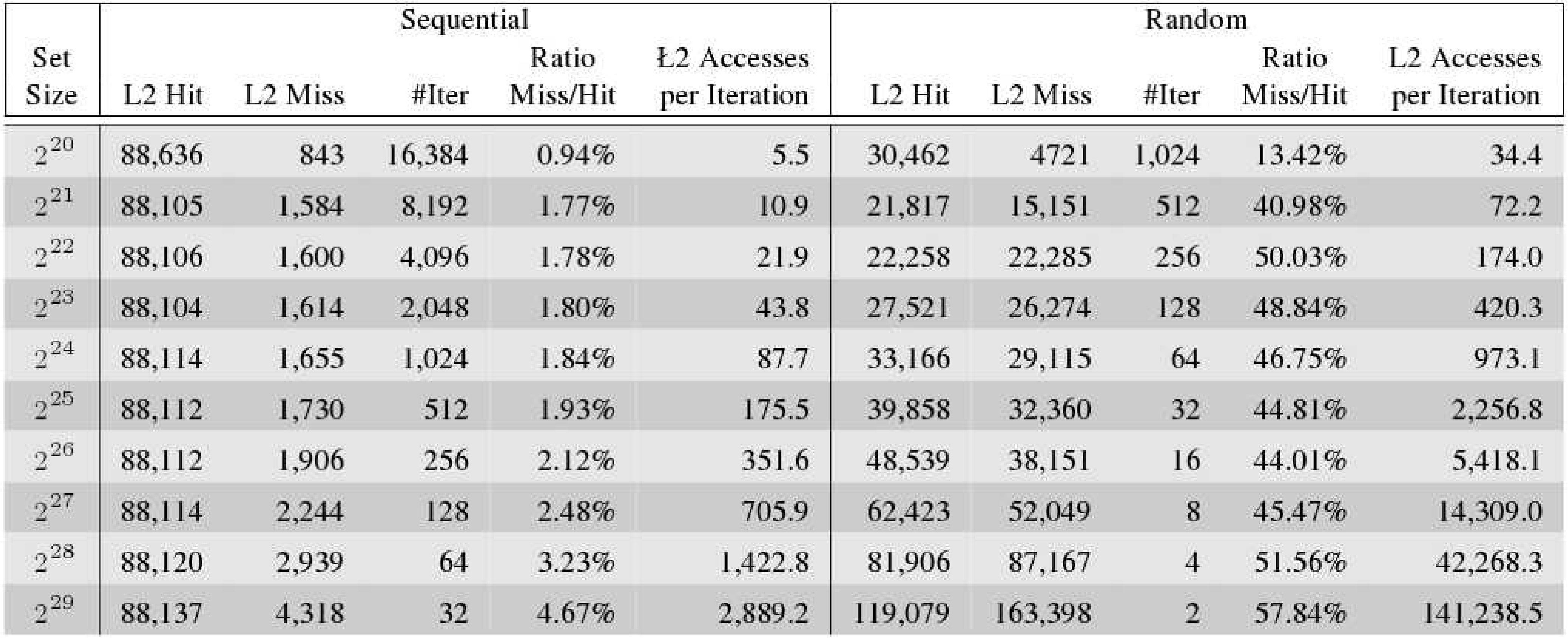
- Inc benchmark, 128 bytes per element
- 32kB L1d, 1MB L2
- 16kB L1d, 512kB L2, 2M L3
- 32kB L1d, 4M L2

Single Thread Random Access



- Prefetching cannot help here
- We have seen that data can be accessed from main memory in 200 cycles. High numbers (400) are here because automatic prefetching if now working against us.
- The curve is not flattening at various plateaus: cache miss ratio increases

L2 Hits and Misses for Sequential and Random Walks, NPAD=0



Set Size	Sequential					Random				
	L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses per Iteration	L2 Hit	L2 Miss	#Iter	Ratio Miss/Hit	L2 Accesses per Iteration
2^{20}	88,636	843	16,384	0.94%	5.5	30,462	4721	1,024	13.42%	34.4
2^{21}	88,105	1,584	8,192	1.77%	10.9	21,817	15,151	512	40.98%	72.2
2^{22}	88,106	1,600	4,096	1.78%	21.9	22,258	22,285	256	50.03%	174.0
2^{23}	88,104	1,614	2,048	1.80%	43.8	27,521	26,274	128	48.84%	420.3
2^{24}	88,114	1,655	1,024	1.84%	87.7	33,166	29,115	64	46.75%	973.1
2^{25}	88,112	1,730	512	1.93%	175.5	39,858	32,360	32	44.81%	2,256.8
2^{26}	88,112	1,906	256	2.12%	351.6	48,539	38,151	16	44.01%	5,418.1
2^{27}	88,114	2,244	128	2.48%	705.9	62,423	52,049	8	45.47%	14,309.0
2^{28}	88,120	2,939	64	3.23%	1,422.8	81,906	87,167	4	51.56%	42,268.3
2^{29}	88,137	4,318	32	4.67%	2,889.2	119,079	163,398	2	57.84%	141,238.5

Write Behaviour

- Cache policies:
  - Write-Through: Dirty cache lines are immediately written to the main memory
  - Write-Back: Dirty cache lines are written only when they are dropped from cache
  - Write-Combinig (used for video memory): Data is written only after the whole cache line is filled. Processor has a write-combine buffer with the same size as cache line. If the whole buffer is filled, there is no need to read the cache line before writing.
  - Uncacheable: typically some hardware registers

Instruction Cache

- Much less problematic than data caches:
  - The quantity of code which is executed depends on the size of the code that is needed. The size of the code in general depends on the complexity of the problem. The complexity of the problem is fixed.
  - While the program’s data handling is designed by the programmer the program’s instructions are usually generated by a compiler. The compiler writers know about the rules for good code generation.
  - Program flow is much more predictable than data access patterns. Today’s CPUs are very good at detecting patterns. This helps with prefetching.
  - Code always has quite good spatial and temporal locality.

Other Issues

- Influence of cache coherency on multi-processors and multi-core

What Programmers Can Do

- If you know the data will be used only once, bypass the cache when writing. Hopefully, write-combining will be used.
  - Non-temporal write operations (gcc)

```
#include <emmintrin.h>
void _mm_stream_si32(int *p, int a);
void _mm_stream_si128(int *p, __m128i a);
void _mm_stream_pd(double *p, __m128d a);
#include <xmmintrin.h>
void _mm_stream_pi(__m64 *p, __m64 a);
void _mm_stream_ps(float *p, __m128 a);
#include <ammintrin.h>
void _mm_stream_sd(double *p, __m128d a);
void _mm_stream_ss(float *p, __m128 a);
```

Example: Matrix multiplication

- Naive implementation

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * mul2[k][j];
```

- With transposition

```
double tmp[N][N];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    tmp[i][j] = mul2[j][i];
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
    for (k = 0; k < N; ++k)
      res[i][j] += mul1[i][k] * tmp[j][k];
```

- Performance: naive: 100%, transposed: 23,4%

Example: Matrix multiplication cont.

-
  ```
  #define SM (CACHE_LINE_SIZE / sizeof (double))
  for (i = 0; i < N; i += SM)
    for (j = 0; j < N; j += SM)
      for (k = 0; k < N; k += SM)
        for (i2 = 0, rres = &res[i][j],
             rmul1 = &mul1[i][k]; i2 < SM;
             ++i2, rres += N, rmul1 += N)
          for (k2 = 0, rmul2 = &mul2[k][j];
               k2 < SM; ++k2, rmul2 += N)
            for (j2 = 0; j2 < SM; ++j2)
              rres[j2] += rmul1[k2] * rmul2[j2];
  ```
- Performace: 17,3% (with vectorized operations: 9,47%)

What Programmers Can Do

Structure layout

- Always move the structure element which is most likely to be the critical word to the beginning of the structure.
- When accessing the data structures and the order of access is not dictated by the situation, access the elements in the order in which they are defined in the structure.
- Align structures to cache lines
  gcc: *struct strtype variable __attribute((aligned(64)));*
- ...

What Programmers Can Do

Prefetching

-
  ```
  #include <xmmintrin.h>
  enum _mm_hint
  {
     _MM_HINT_T0 = 3,
     _MM_HINT_T1 = 2,
     _MM_HINT_T2 = 1,
     _MM_HINT_NTA = 0
  };
  void _mm_prefetch(void *p,
                    enum _mm_hint h);
  ```
- Helper threads – reads memory in advance to bring it into cache

References

- Ulrich Drepper, “What Every Programmer Should Know About Memory”, 2007/11 [online], http://people.redhat.com/drepper/cpumemory.pdf
- http://en.wikipedia.org/wiki/CPU_cache
- C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, P. Dubey. "FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs." 2010 ACM SIGMOD/PODS Conference (SIGMOD '10). Indianapolis, IN, 2010. http://www.kaldewey.com/pubs/FAST__SIGMOD10.pdf