CS2101: Foundations of High-performance Computing

Marc Moreno Maza

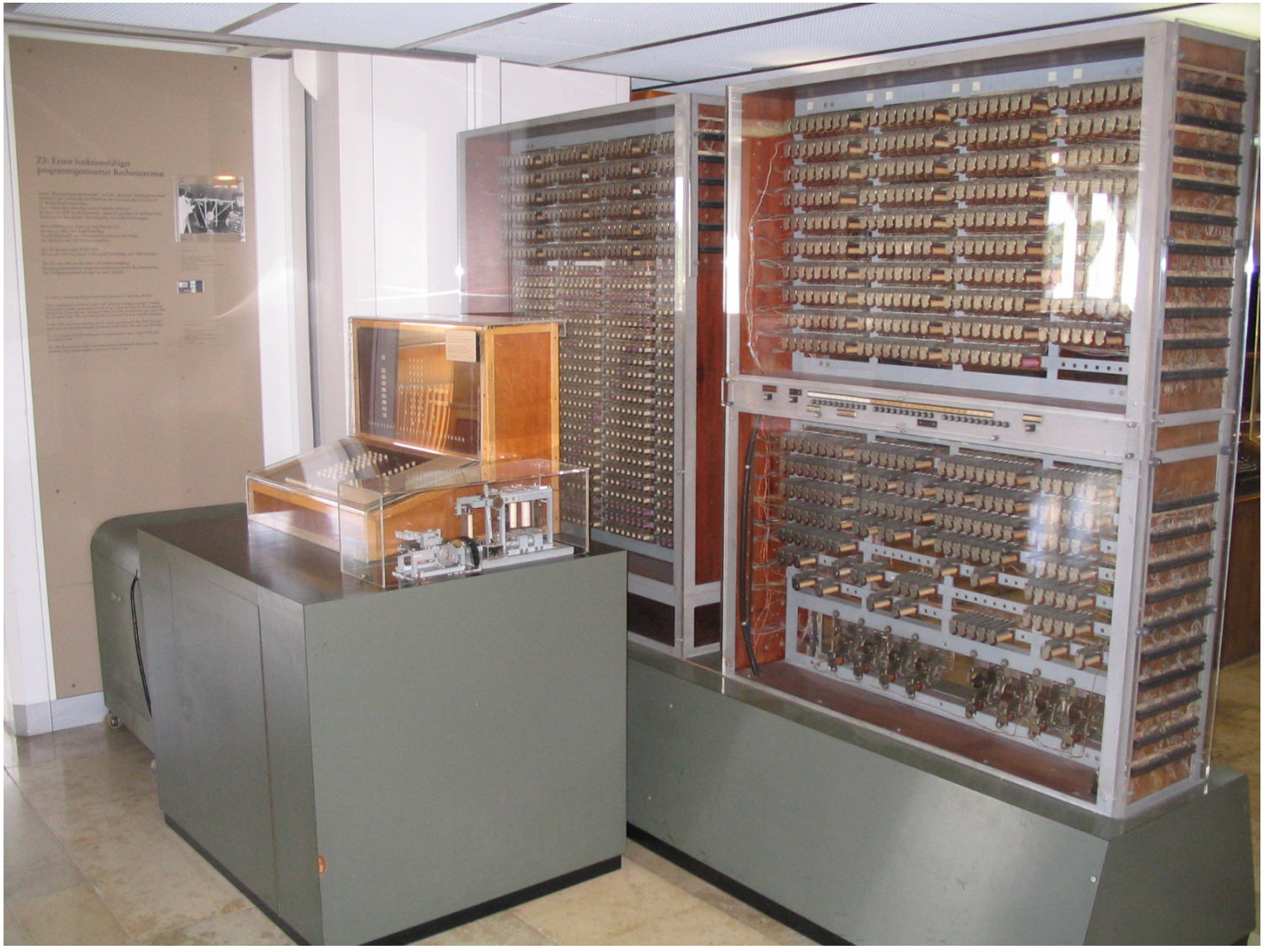
University of Western Ontario, London, Ontario (Canada)

CS2101

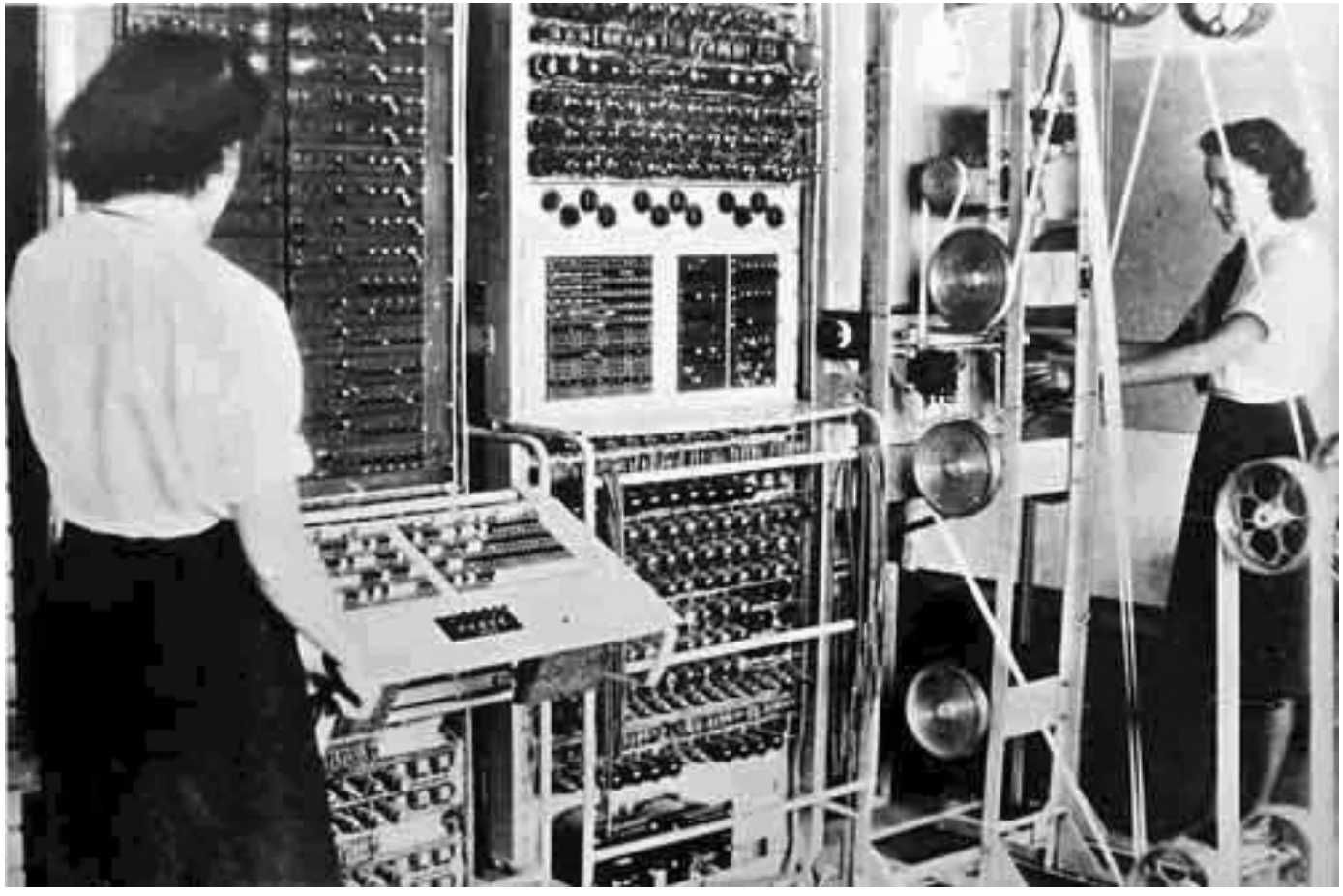
Plan

1. Hardware Acceleration Technologies
2. Software Performance Engineering
3. Cache Memories
4. A Case Study: Matrix Multiplication
5. Multicore Architectures
6. Multicore Programming
7. CS2101 Course Outline

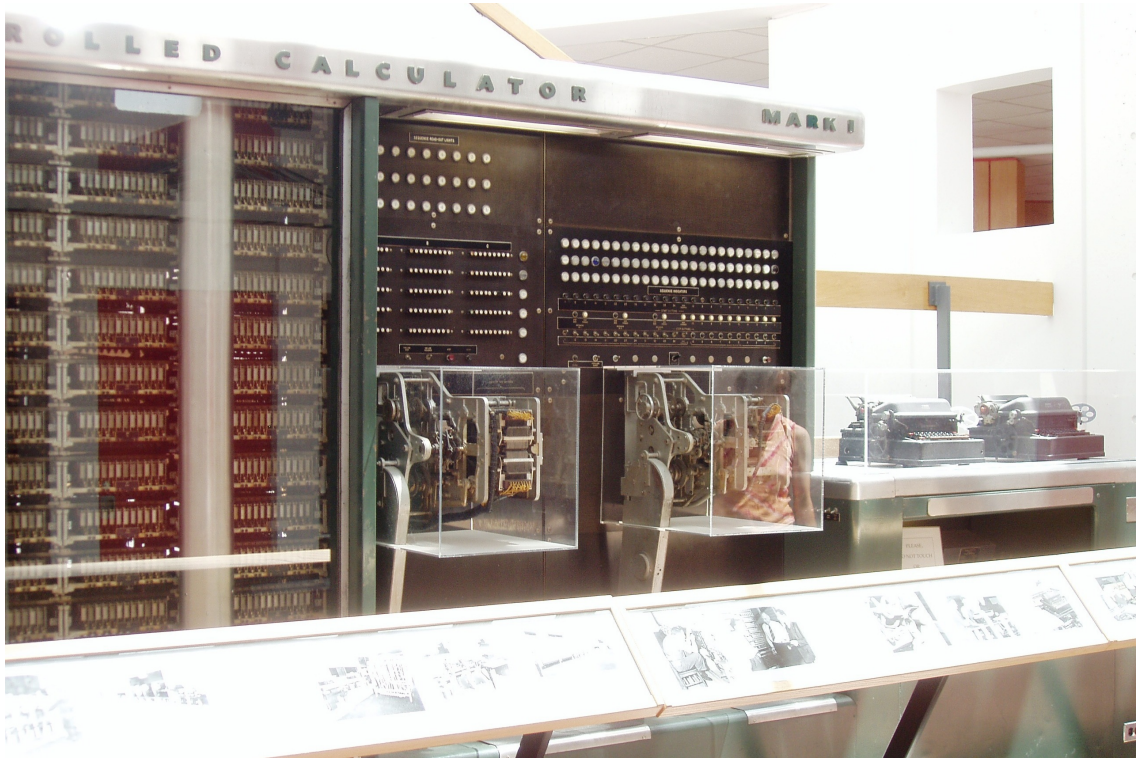
Plan



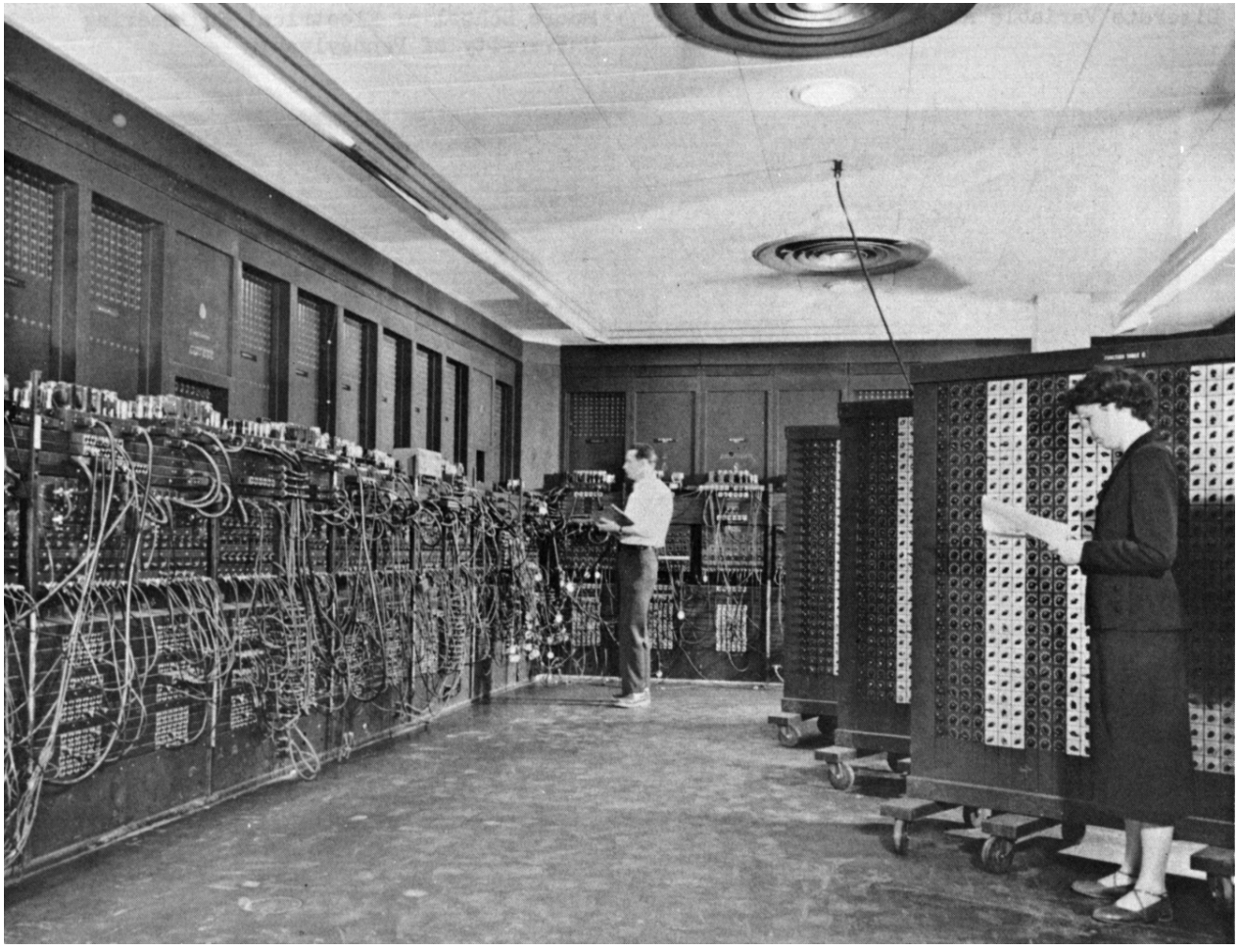
Konrad Zuse's Z3 electro-mechanical computer (1941, Germany). Turing complete, though conditional jumps were missing.



Colossus (UK, 1941) was the world's first totally electronic programmable computing device. But not Turing complete.



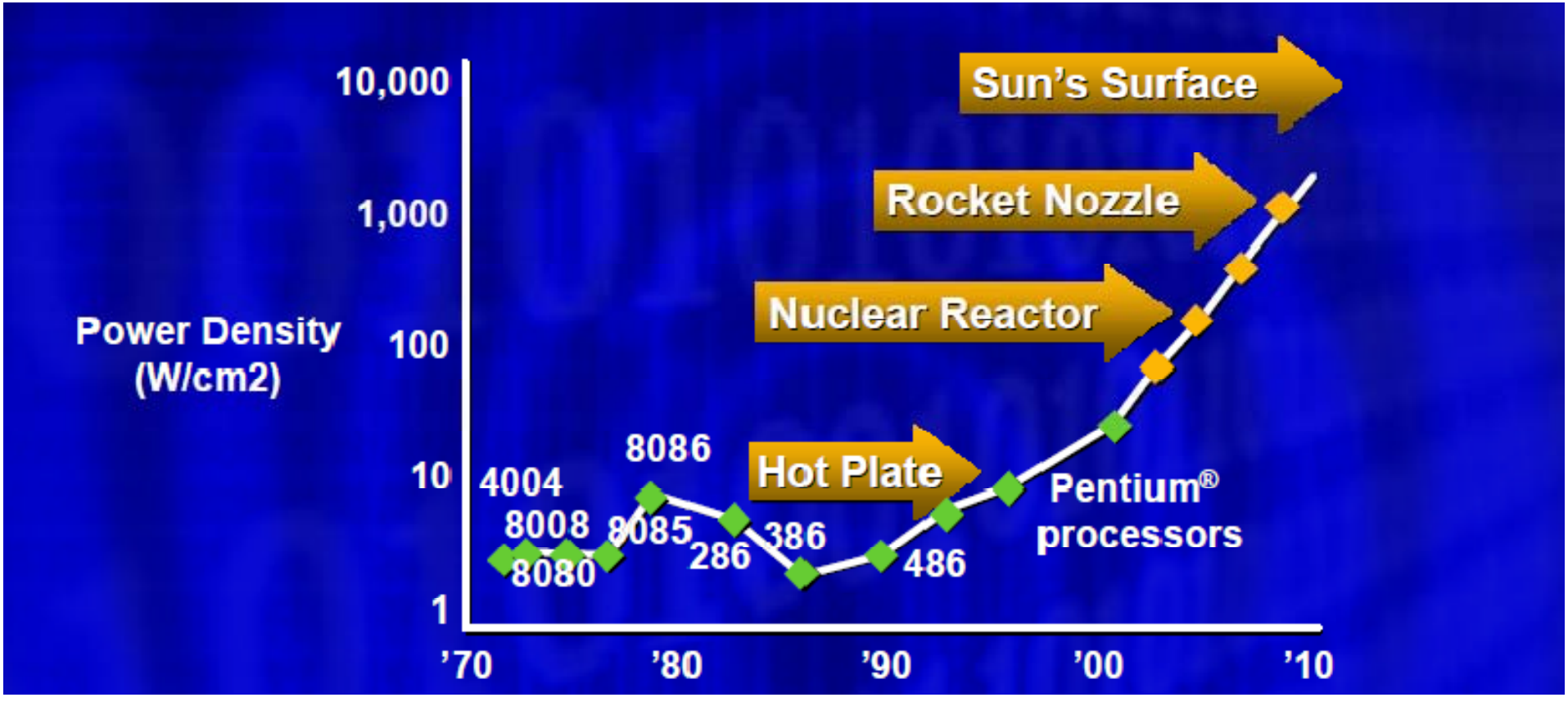
Harvard Mark I IBM ASCC (1944, US). Electro-mechanical computer (no conditional jumps and not Turing complete). It could store 72 numbers, each 23 decimal digits long. It could do three additions or subtractions in a second. A multiplication took six seconds, a division took 15.3 seconds, and a logarithm or a trigonometric function took over one minute. A loop was accomplished by joining the end of the paper tape containing the program back to the beginning of the tape (literally creating a loop).



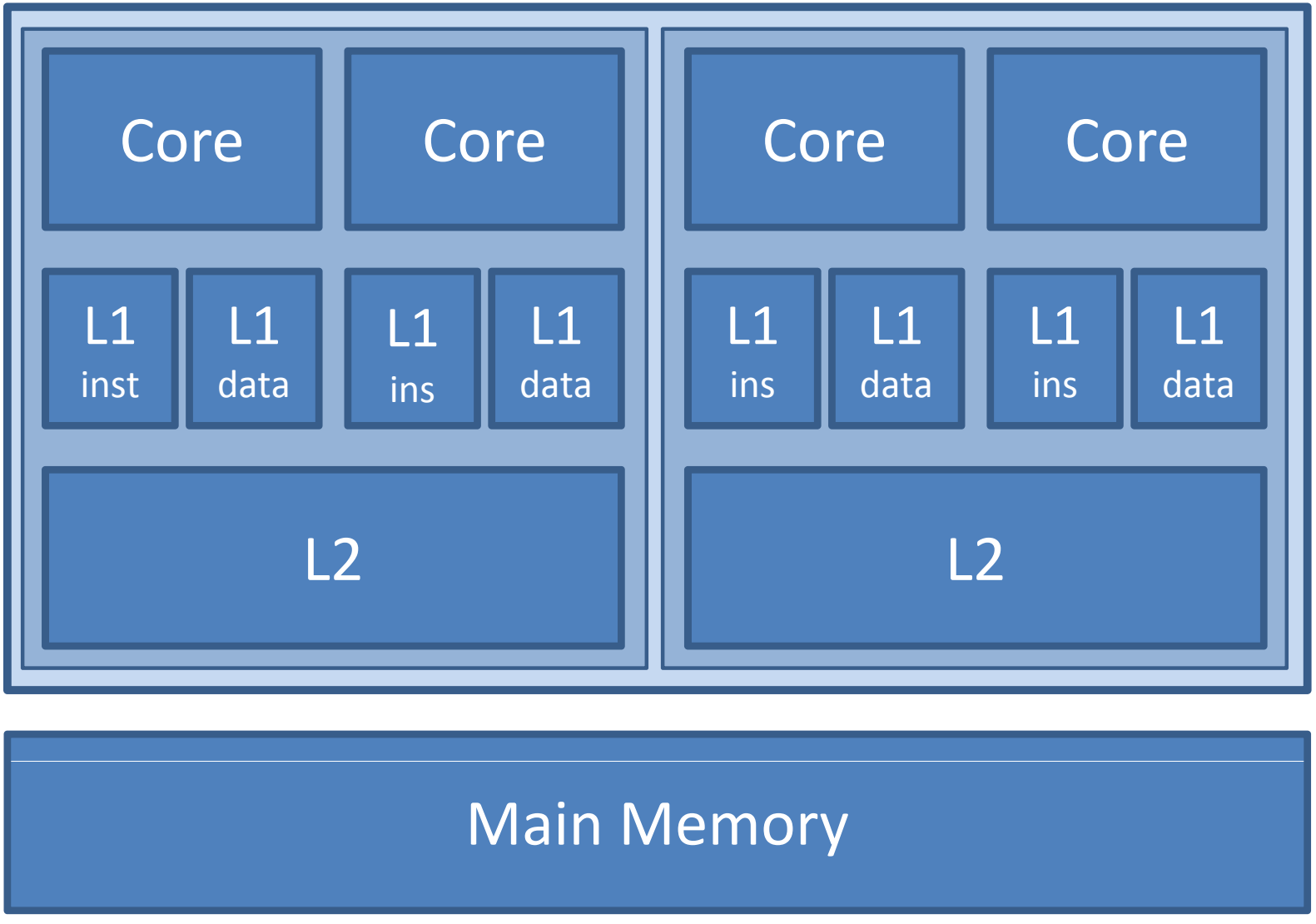
Electronic Numerical Integrator And Computer (ENIAC). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.

The IBM Personal Computer, commonly known as the IBM PC (Introduced on August 12, 1981).

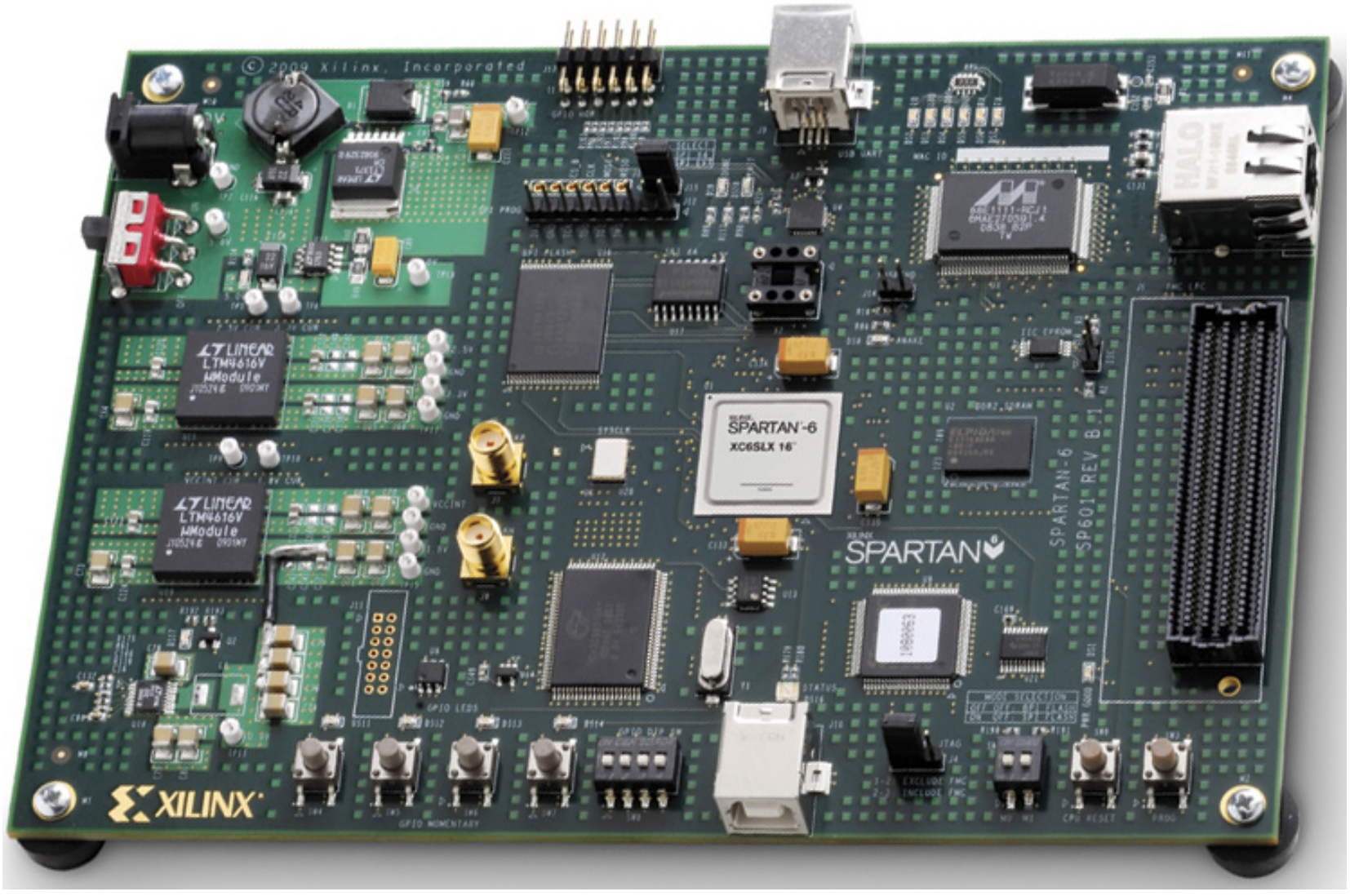
The Pentium Family.


Power Density
(W/cm2)
10,000
1,000
100
10
1
Sun's Surface
Rocket Nozzle
Nuclear Reactor
Hot Plate
Pentium®
processors
4004
8008
8080
8085
8086
286
386
486
'70
'80
'90
'00
'10


INTEL©'05 Q6600
INTEL® CORE™2 QUAD
SLACR MALAY
2.40GHZ/8M/1066/05A
L738B526

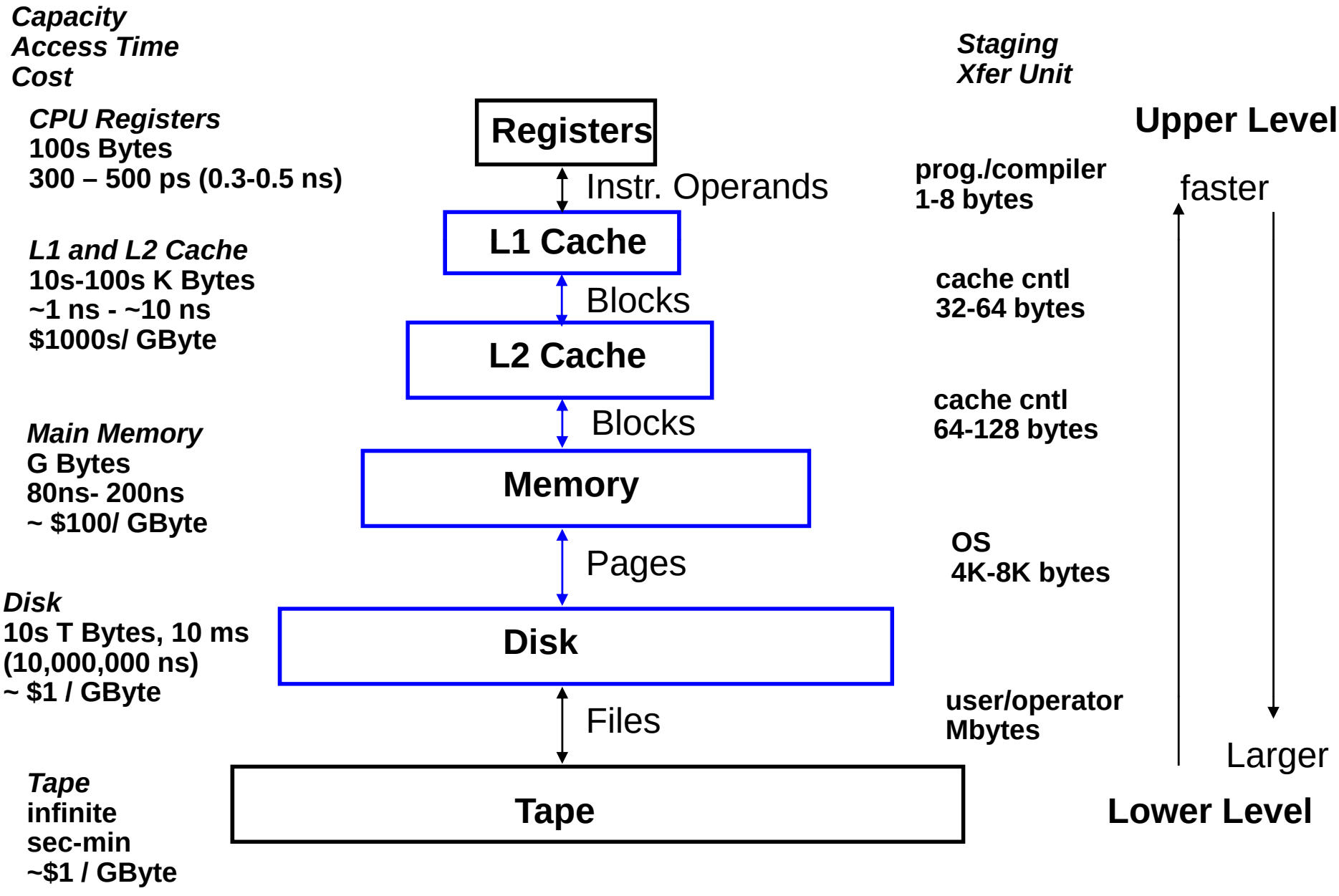

Core
Core
Core
Core
L1
inst
L1
data
L1
ins
L1
data
L1
ins
L1
data
L1
ins
L1
data
L2
L2
Main Memory


94V-0 MADE IN CHINA
NVIDIA
L7118760 0746A2
S TAIWAN
NF2043.00P
G92-270-A2
RoHS
COMPLIANT


© 2009 Xilinx, Incorporated
LT LINEAR
LTM4616V
µModule
LT LINEAR
LTM4616V
µModule
SPARTAN-6
XC6SLX 16
SPARTAN
SP601 REV B1
XILINX

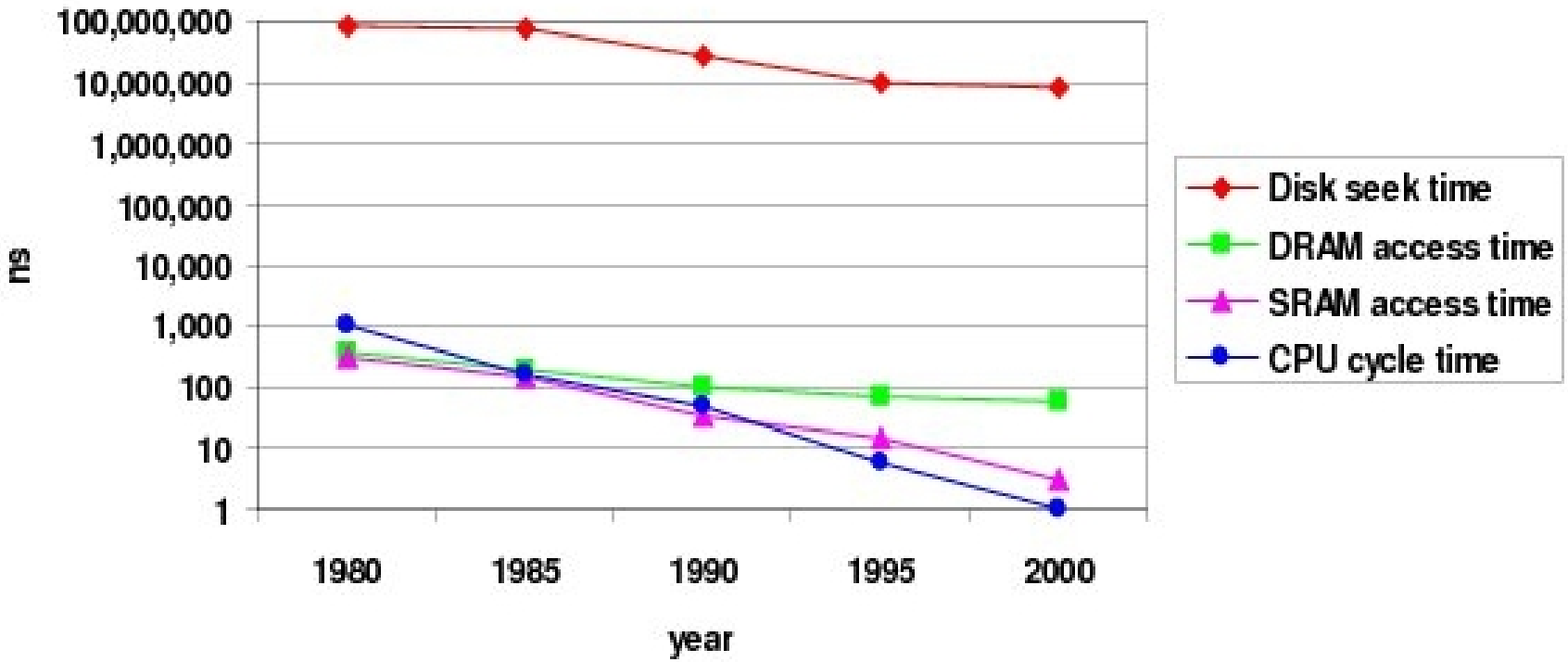
L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Typical cache specifications of a multicore in 2008.


Capacity
Access Time
Cost
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)
L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte
Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte
Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte
Tape
infinite
sec-min
~$1 / GByte
Registers
Instr. Operands
L1 Cache
Blocks
L2 Cache
Blocks
Memory
Pages
Disk
Files
Tape
Staging
Xfer Unit
prog./compiler
1-8 bytes
cache cntl
32-64 bytes
cache cntl
64-128 bytes
OS
4K-8K bytes
user/operator
Mbytes
Upper Level
faster
Larger
Lower Level

The CPU-Memory Gap

The increasing gap between DRAM, disk, and CPU speeds.



Once uopn a time, every thing was slow in a computer . . .

Plan

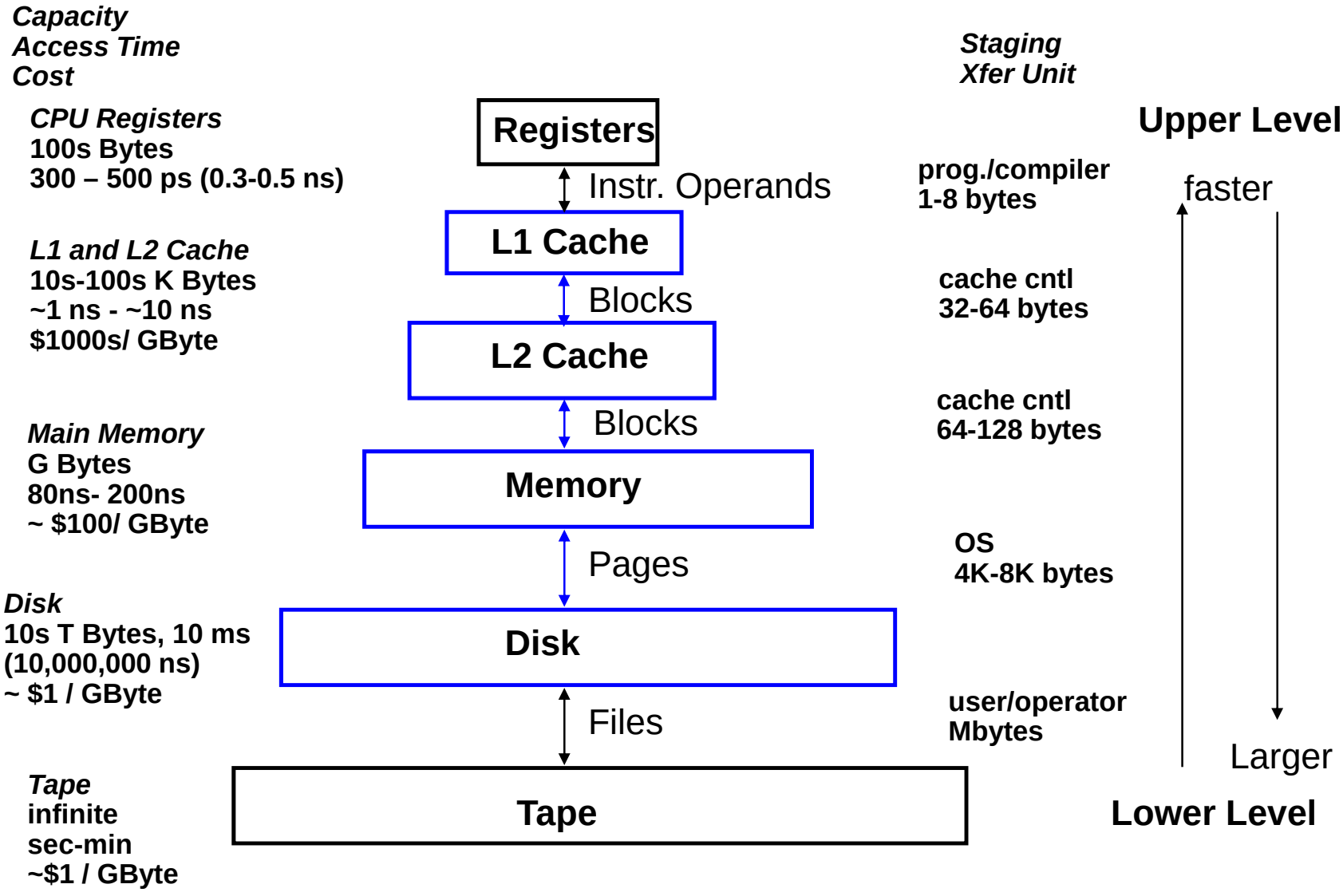
Why is Performance Important?

- Acceptable response time (Anti-lock break system, Mpeg decoder, Google Search, etc.)
- Ability to scale (from hundred to millions of users/documents/data)
- Use less power / resource (viability of cell phones dictated by battery life, etc.)

Improving Performance is Hard

- Knowing that there is a performance problem: complexity estimates, performance analysis software tools, read the generated assembly code, scalability testing, comparisons to similar programs, experience and curiosity!
- Establishing the leading cause of the problem: examine the algorithm, the data structures, the data layout; understand the programming environment and architecture.
- Eliminating the performance problem: (Re-)design the algorithm, data structures and data layout, write programs *close to the metal* (C/C++), adhere to software engineering principles (simplicity, modularity, portability)
- Golden rule: Be reactive, not proactive!

Remember that Picture!



Plan

Capacity
Access Time
Cost

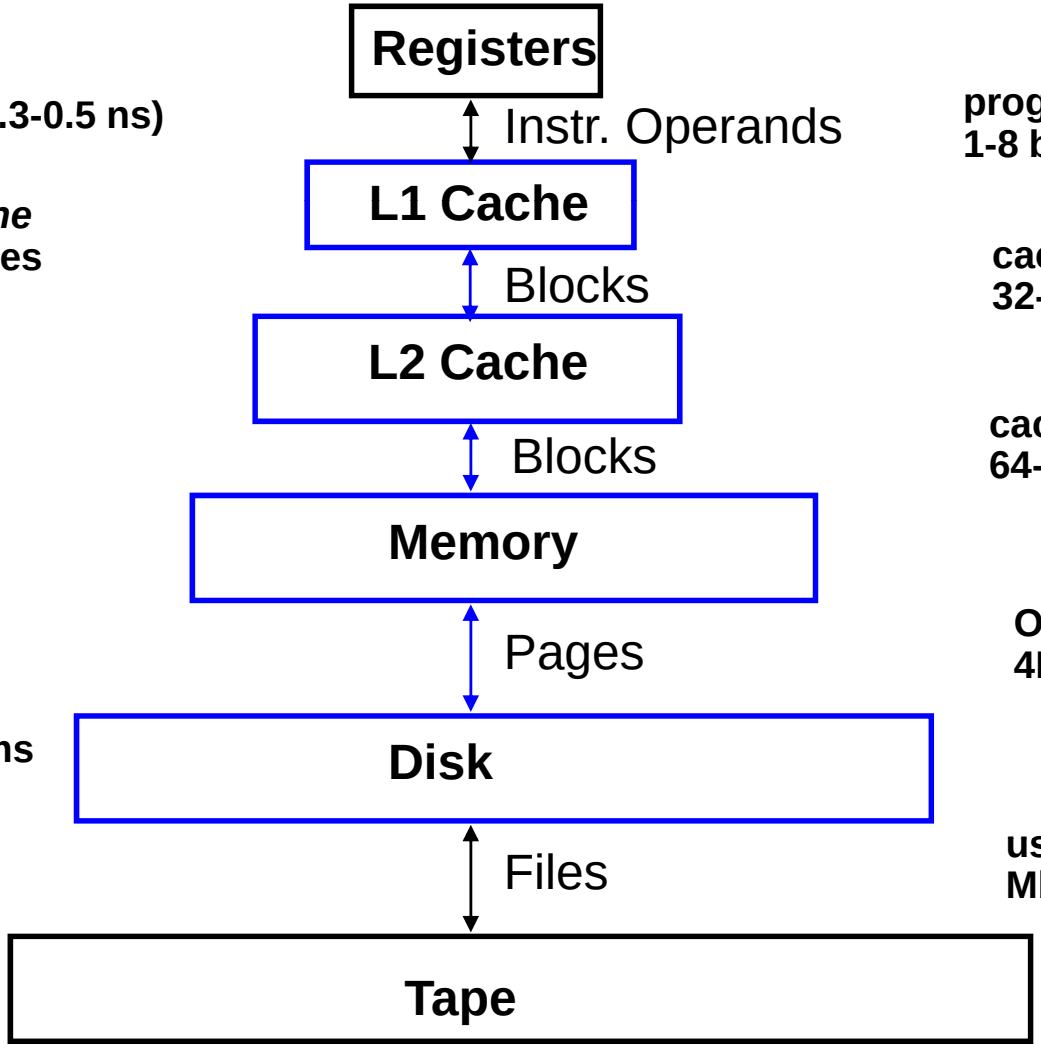
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)

L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte

Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte

Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte

Tape
infinite
sec-min
~$1 / GByte



Staging
Xfer Unit

prog./compiler
1-8 bytes

cache cntl
32-64 bytes

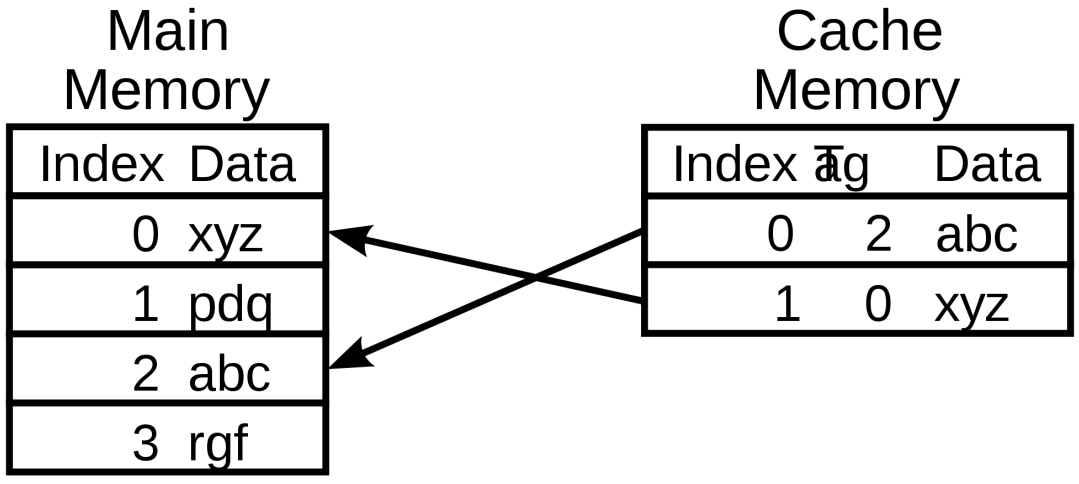
cache cntl
64-128 bytes

OS
4K-8K bytes

user/operator
Mbytes

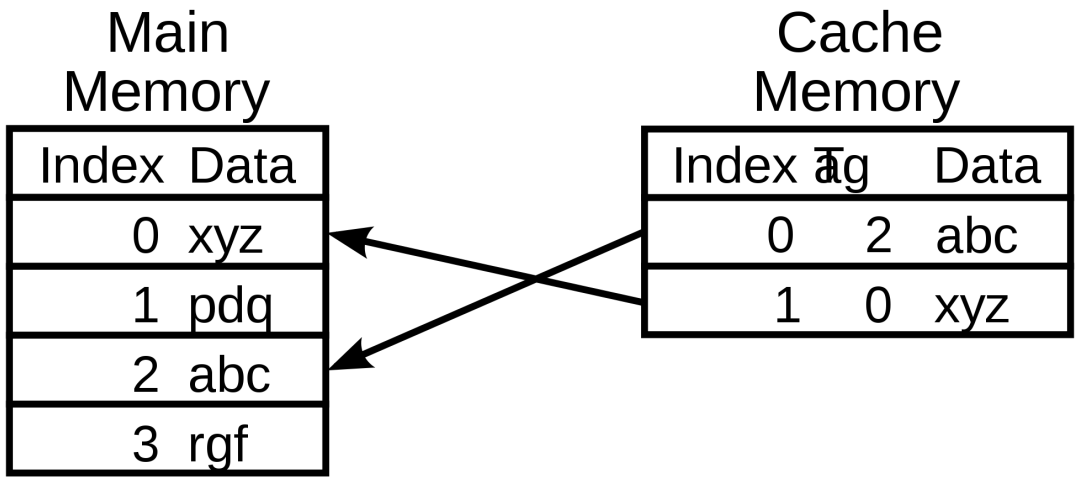
Upper Level
faster
Larger
Lower Level

CPU Cache (1/7)



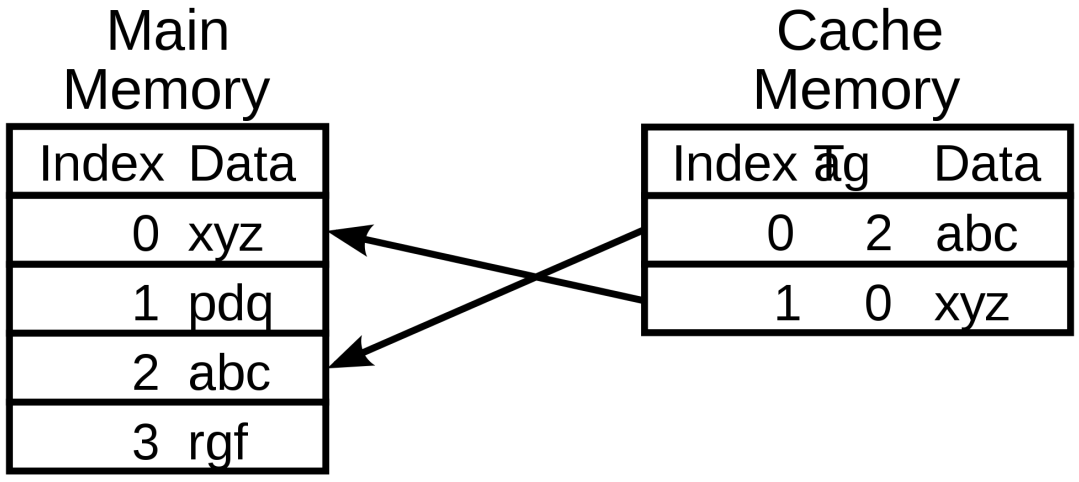
- A CPU cache is an auxiliary memory which is smaller, faster memory than the main memory and which stores copies of the main memory locations that are expectedly frequently used.
- Most modern desktop and server CPUs have at least three independent caches: the data cache, the instruction cache and the translation look-aside buffer.

CPU Cache (2/7)



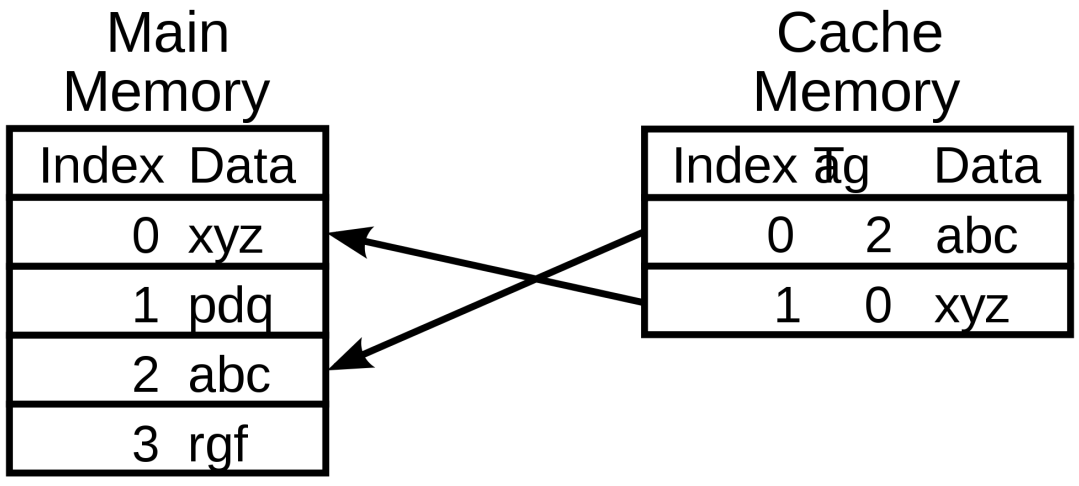
- Each location in each memory (main or cache) has
  - a datum (cache line) which ranges between 8 and 512 bytes in size, while a datum requested by a CPU instruction ranges between 1 and 16.
  - a unique index (called address in the case of the main memory)
- In the cache, each location has also a tag (storing the address of the corresponding cached datum).

CPU Cache (3/7)



- When the CPU needs to read or write a location, it checks the cache:
  - if it finds it there, we have a cache hit
  - if not, we have a cache miss and (in most cases) the processor needs to create a new entry in the cache.
- Making room for a new entry requires a replacement policy: the Least Recently Used (LRU) discards the least recently used items first; this requires to use age bits.

CPU Cache (4/7)

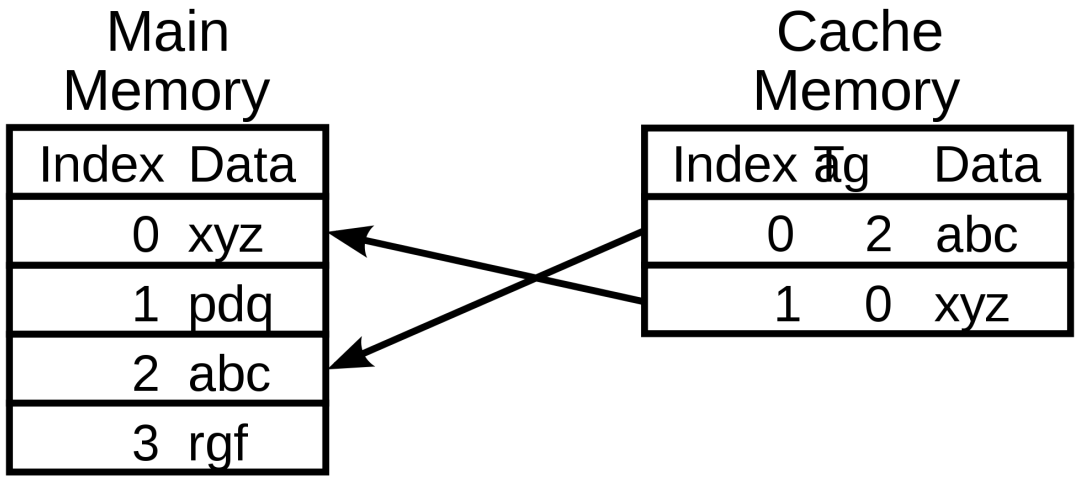


- Read latency (time to read a datum from the main memory) requires to keep the CPU busy with something else:

  out-of-order execution: attempt to execute independent instructions arising after the instruction that is waiting due to the cache miss

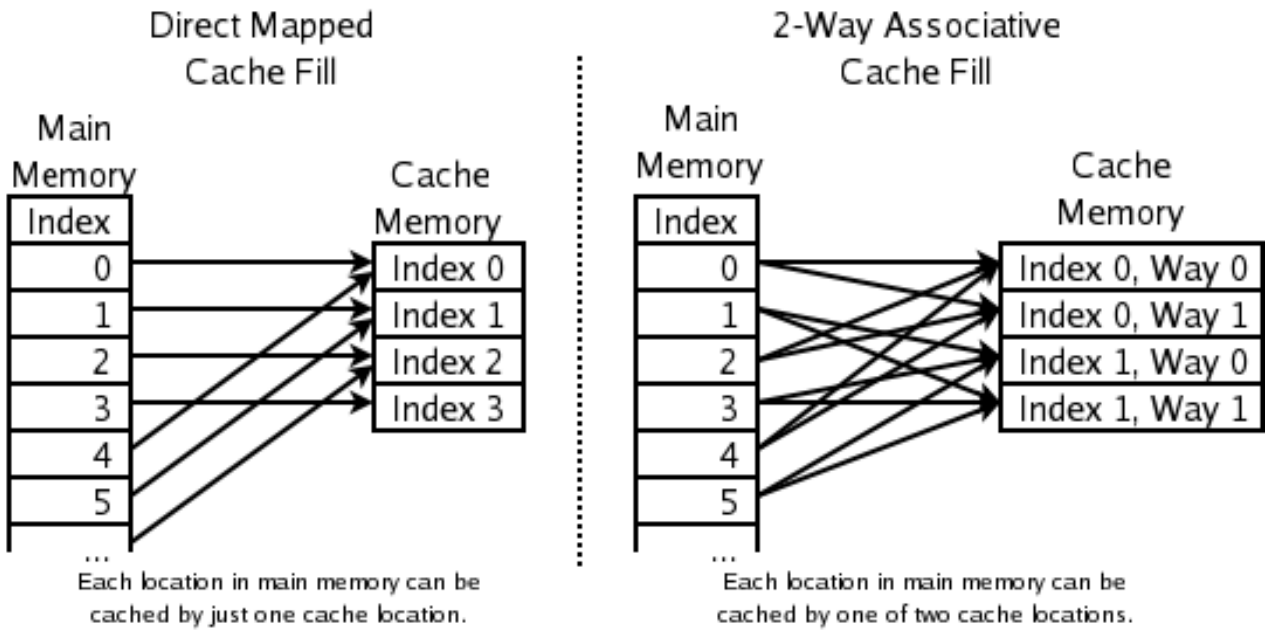
  hyper-threading (HT): allows an alternate thread to use the CPU

CPU Cache (5/7)

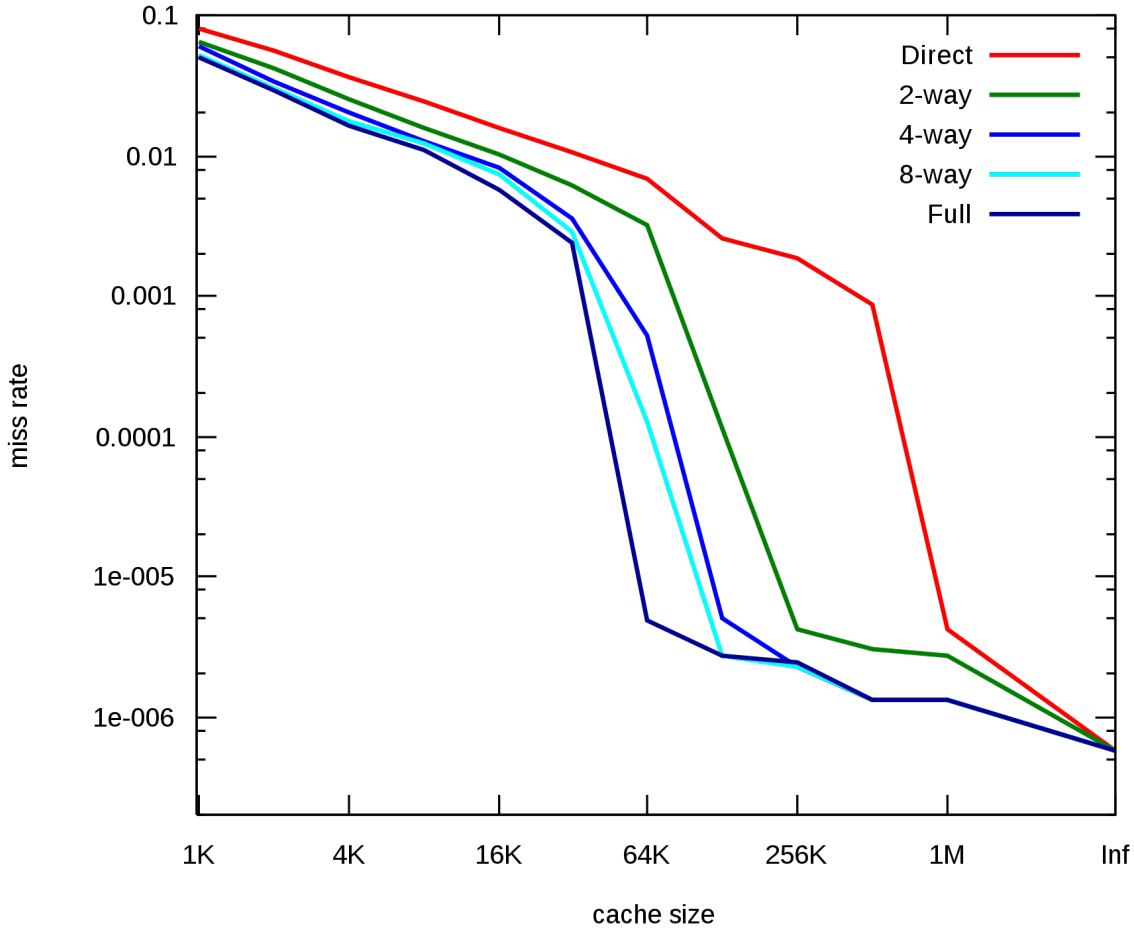


- Modifying data in the cache requires a write policy for updating the main memory
  - write-through cache: writes are immediately mirrored to main memory
  - write-back cache: the main memory is mirrored when that data is evicted from the cache
- The cache copy may become out-of-date or stale, if other processors modify the original entry in the main memory.

CPU Cache (6/7)



- The replacement policy decides where in the cache a copy of a particular entry of main memory will go:
  - fully associative: any entry in the cache can hold it
  - direct mapped: only one possible entry in the cache can hold it
  - N-way set associative: N possible entries can hold it



- *Cache Performance for SPEC CPU2000* by J.F. Cantin and M.D. Hill.
- The SPEC CPU2000 suite is a collection of 26 compute-intensive, non-trivial programs used to evaluate the performance of a computer's CPU, memory system, and compilers (`http://www.spec.org/osg/cpu2000`).

Cache issues

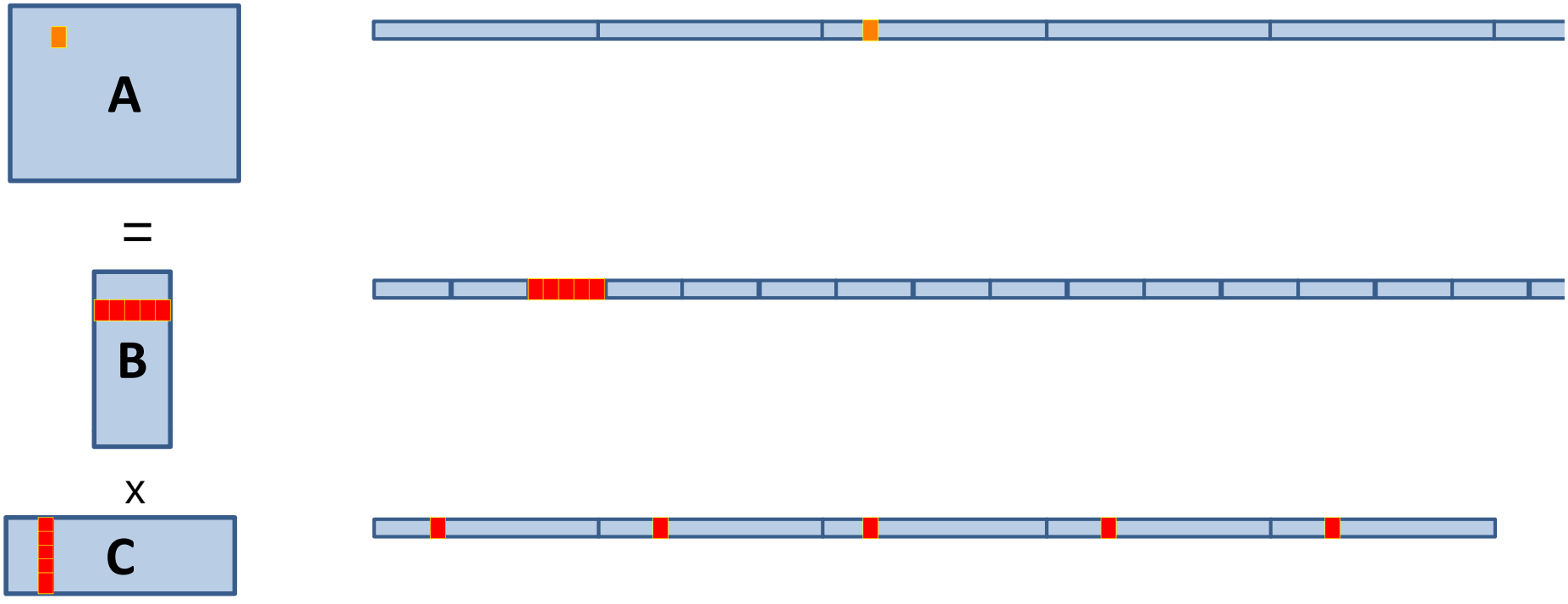
- **Cold miss:** The first time the data is available. Cure: Prefetching may be able to reduce this type of cost.
- **Capacity miss:** The previous access has been evicted because too much data touched in between, since the *working data set* is too large. Cure: Reorganize the data access such that *reuse* occurs before eviction.
- **Conflict miss:** Multiple data items mapped to the same location with eviction before cache is full. Cure: Rearrange data and/or pad arrays.
- **True sharing miss:** Occurs when a thread in another processor wants the same data. Cure: Minimize sharing.
- **False sharing miss:** Occurs when another processor uses different data in the same cache line. Cure: Pad data.

Plan

A typical matrix multiplication C code

```
#define IND(A, x, y, d) A[(x)*(d)+(y)]
uint64_t testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended;
        float timeTaken;
        int i, j, k;
        srand(getSeed());
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
                    //  A[i][j] += B[i][k] + C[k][j];
              IND(A,i,j,y) += IND(B,i,k,z) * IND(C,k,j,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with matrix representation

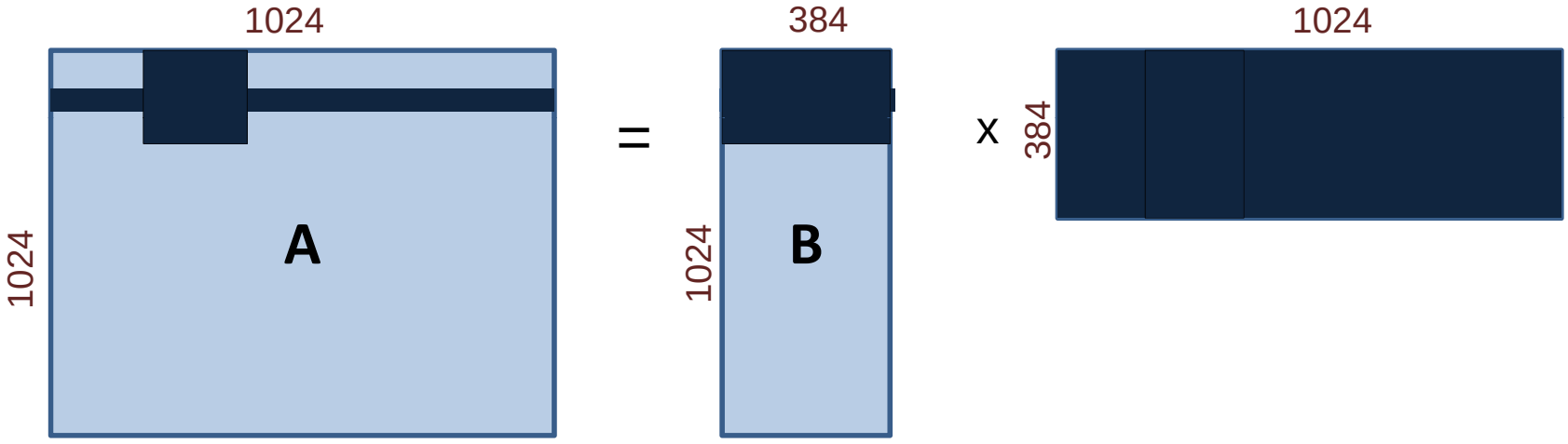


- Contiguous accesses are better:
  - Data fetch as cache line (Core 2 Duo 64 byte L2 Cache line)
  - With contiguous data, a single cache fetch supports 8 reads of doubles.
  - Transposing the matrix `C` should reduce L1 cache misses!

Transposing for optimizing spatial locality

```
float testMM(const int x, const int y, const int z)
{
  double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        Cx = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for(j =0; j < y; j++)
          for(k=0; k < z; k++)
            IND(Cx,j,k,z) = IND(C, k, j, y);
        for (i = 0; i < x; i++)
          for (j = 0; j < y; j++)
             for (k = 0; k < z; k++)
               IND(A, i, j, y) += IND(B, i, k, z) *IND(Cx, j, k, z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
  return timeTaken;
}
```

Issues with data reuse



- Naive calculation of a row of A, so computing 1024 coefficients: 1024 accesses in A, 384 in B and $1024 \times 384 = 393,216$ in C. Total $= 394,524$.
- Computing a 32×32-block of A, so computing again 1024 coefficients: 1024 accesses in A, 384×32 in B and 32×384 in C. Total $= 25,600$.
- The iteration space is traversed so as to reduce memory accesses.

Blocking for optimizing temporal locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
            IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,k0,j0,z);
         ended = example_get_time();
         timeTaken = (ended - started)/1.f;
   return timeTaken;
}
```

Transposing and blocking for optimizing data locality

```
float testMM(const int x, const int y, const int z)
{
        double *A; double *B; double *C; double *Cx;
        long started, ended; float timeTaken; int i, j, k, i0, j0, k0;
        A = (double *)malloc(sizeof(double)*x*y);
        B = (double *)malloc(sizeof(double)*x*z);
        C = (double *)malloc(sizeof(double)*y*z);
        srand(getSeed());
        for (i = 0; i < x*z; i++) B[i] = (double) rand() ;
        for (i = 0; i < y*z; i++) C[i] = (double) rand() ;
        for (i = 0; i < x*y; i++) A[i] = 0 ;
        started = example_get_time();
        for (i = 0; i < x; i += BLOCK_X)
          for (j = 0; j < y; j += BLOCK_Y)
            for (k = 0; k < z; k += BLOCK_Z)
              for (i0 = i; i0 < min(i + BLOCK_X, x); i0++)
                for (j0 = j; j0 < min(j + BLOCK_Y, y); j0++)
                   for (k0 = k; k0 < min(k + BLOCK_Z, z); k0++)
              IND(A,i0,j0,y) += IND(B,i0,k0,z) * IND(C,j0,k0,z);
        ended = example_get_time();
        timeTaken = (ended - started)/1.f;
        return timeTaken;
}
```

Experimental results

Computing the product of two $n \times n$ matrices on my laptop (Core2 Duo CPU P8600 @ 2.40GHz, L1 cache of 3072 KB, 4 GBytes of RAM)

n	naive	transposed	speedup	64×64-tiled	speedup	t. & t.	speedup
128	7	3		7		2	
256	26	43		155		23	
512	1805	265	6.81	1928	0.936	187	9.65
1024	24723	3730	6.62	14020	1.76	1490	16.59
2048	271446	29767	9.11	112298	2.41	11960	22.69
4096	2344594	238453	**9.83**	1009445	**2.32**	101264	**23.15**

Timings are in milliseconds.

The cache-oblivious multiplication (more on this later) runs within 12978 and 106758 for $n = 2048$ and $n = 4096$ respectively.

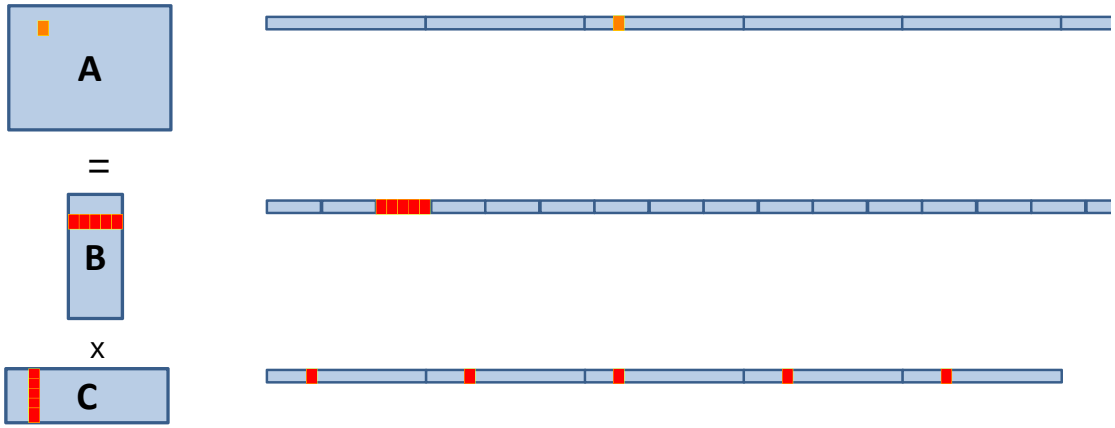
Other performance counters

Hardware count events

- CPI Clock cycles Per Instruction: the number of clock cycles that happen when an instruction is being executed. With pipelining we can improve the CPI by exploiting instruction level parallelism
- L1 and L2 Cache Miss Rate.
- Instructions Retired: In the event of a misprediction, instructions that were scheduled to execute along the mispredicted path must be canceled.

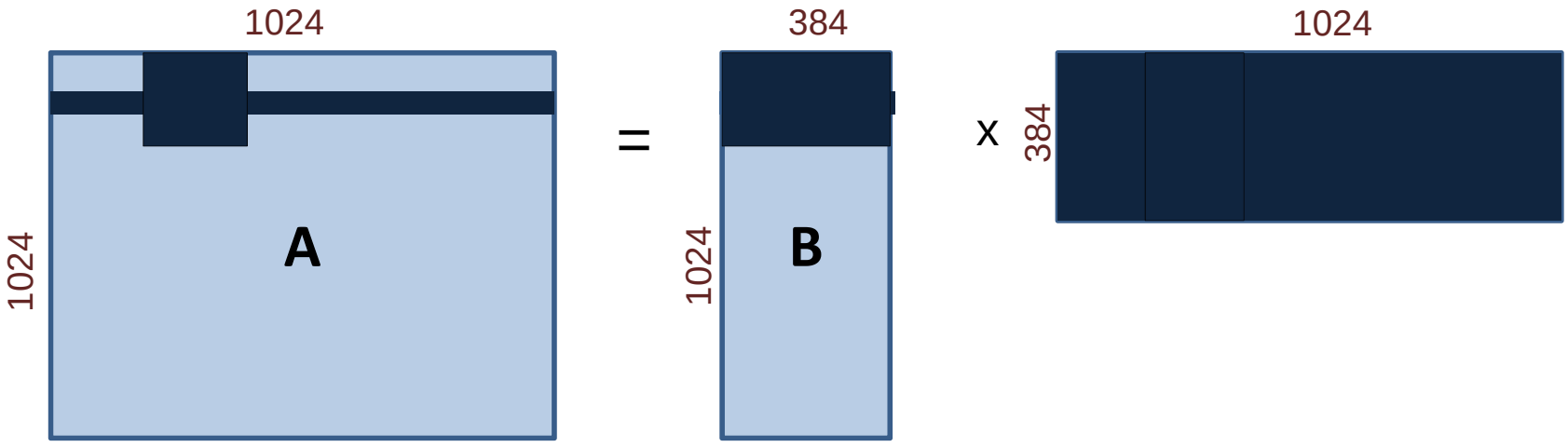
	CPI		L1 Miss Rate		L2 Miss Rate	Percent SSE Instructions	Instructions Retired	
In C	4.78		0.24		0.02	43%	13,137,280,000	
		5x		2x				1x
Transposed	1.13		0.15		0.02	50%	13,001,486,336	
		3x		8x				0.8x
Tiled	0.49		0.02		0	39%	18,044,811,264	

Analyzing cache misses in the naive and transposed multiplication



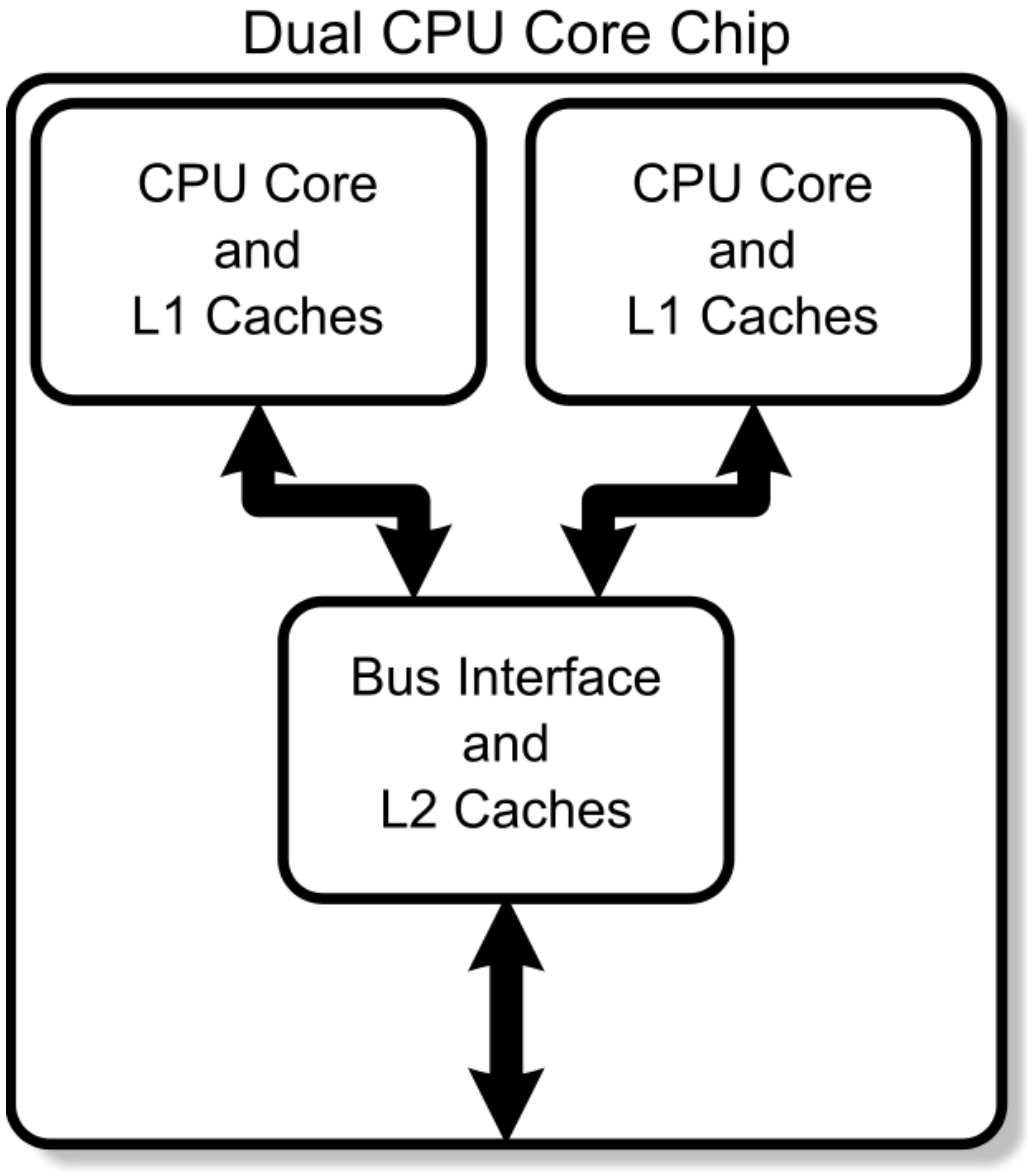
- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- A is scanned one, so mn/L cache misses if L is the number of coefficients per cache line.
- B is scanned n times, so mnp/L cache misses if the cache cannot hold a row.
- C is accessed "nearly randomly" (for m large enough) leading to mnp cache misses.
- Since $2m\,n\,p$ arithmetic operations are performed, this means roughly **one cache miss per flop!**
- If C is transposed, then the ratio improves to 1 for L.

Analyzing cache misses in the tiled multiplication

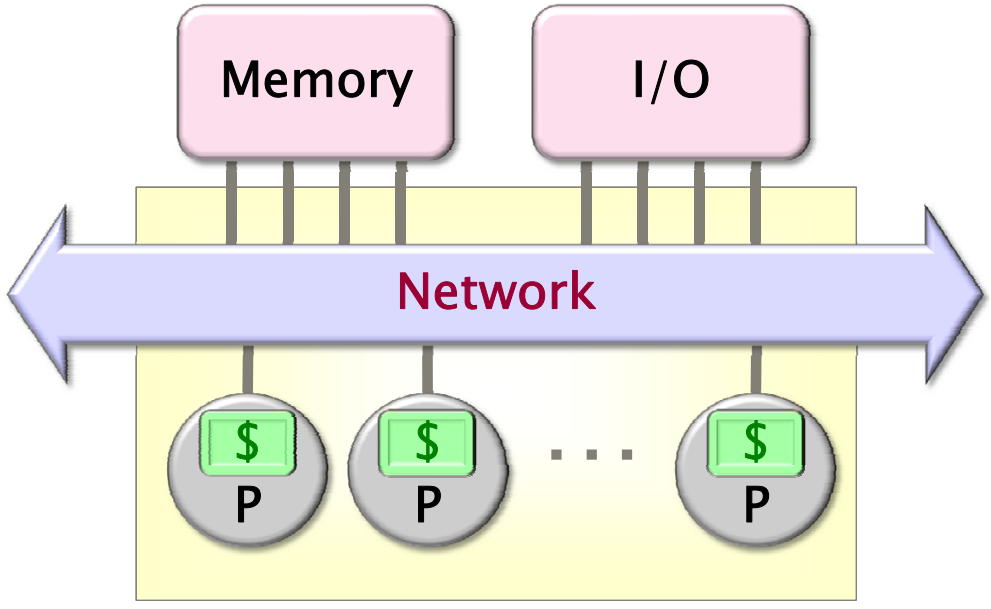


- Let A, B and C have format (m, n), (m, p) and (p, n) respectively.
- Assume all tiles are square of order B and three fit in cache.
- If C is transposed, then loading three blocks in cache cost $3B^2/L$.
- This process happens n^3/B^3 times, leading to $3n^3/(BL)$ cache misses.
- Three blocks fit in cache for $3B^2 < Z$, if Z is the cache size.
- So $O(n^3/(\sqrt{Z}L))$ cache misses, if B is well chosen, which is optimal.

Plan



- A multi-core processor is an integrated circuit to which two or more individual processors (called cores in this sense) have been attached.



Chip Multiprocessor (CMP)

- Cores on a multi-core device can be coupled tightly or loosely:
  - may share or may not share a cache,
  - implement inter-core communications methods or message passing.
- Cores on a multi-core implement the same architecture features as single-core systems such as instruction pipeline parallelism (ILP), vector-processing, SIMD or multi-threading.

Cache Coherence (1/6)

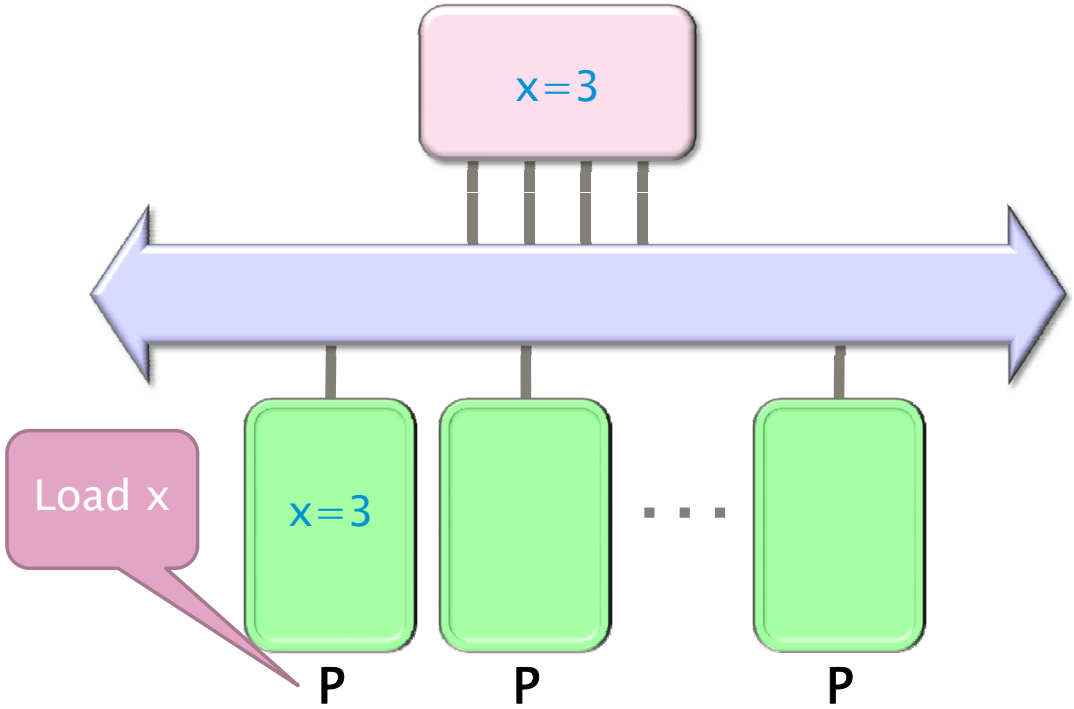


Figure: Processor P_1 reads x=3 first from the backing store (higher-level memory)

Cache Coherence (2/6)

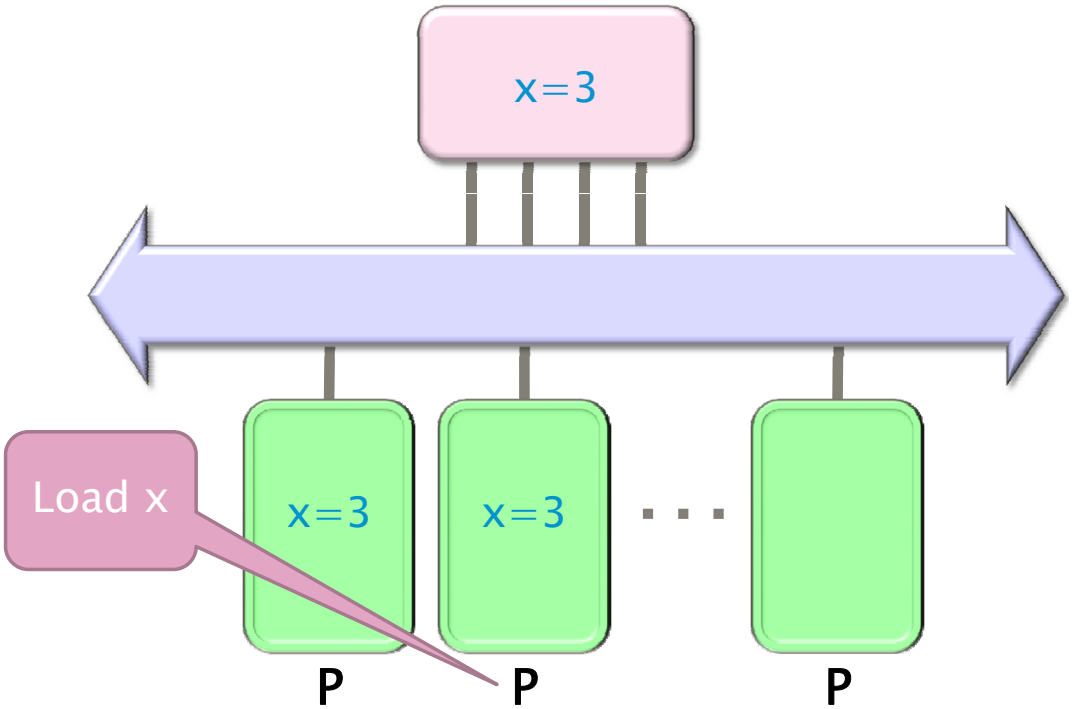


Figure: Next, Processor P_2 loads x=3 from the same memory

Cache Coherence (3/6)

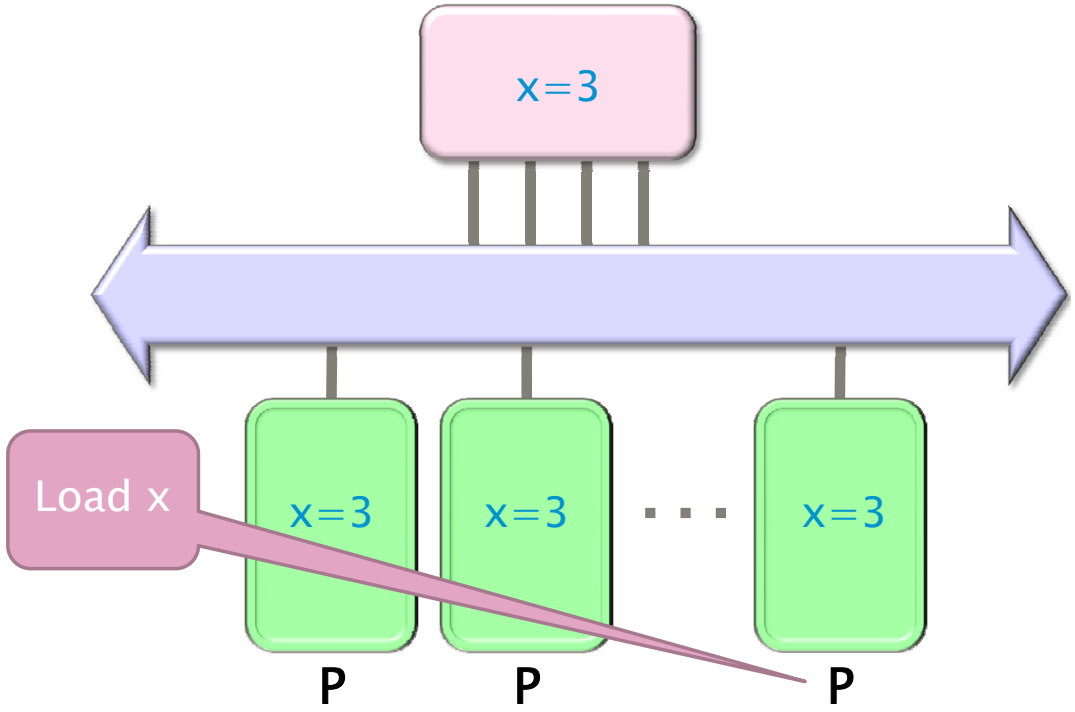


Figure: Processor P_4 loads x=3 from the same memory

Cache Coherence (4/6)

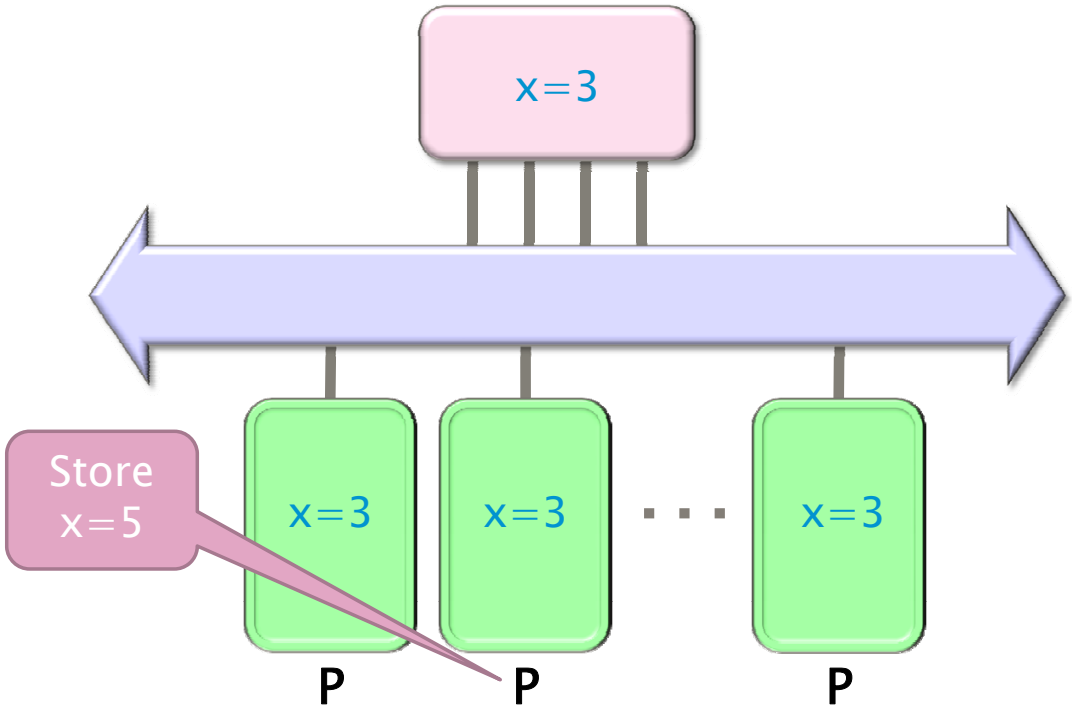


Figure: Processor P_2 issues a write x=5

Cache Coherence (5/6)

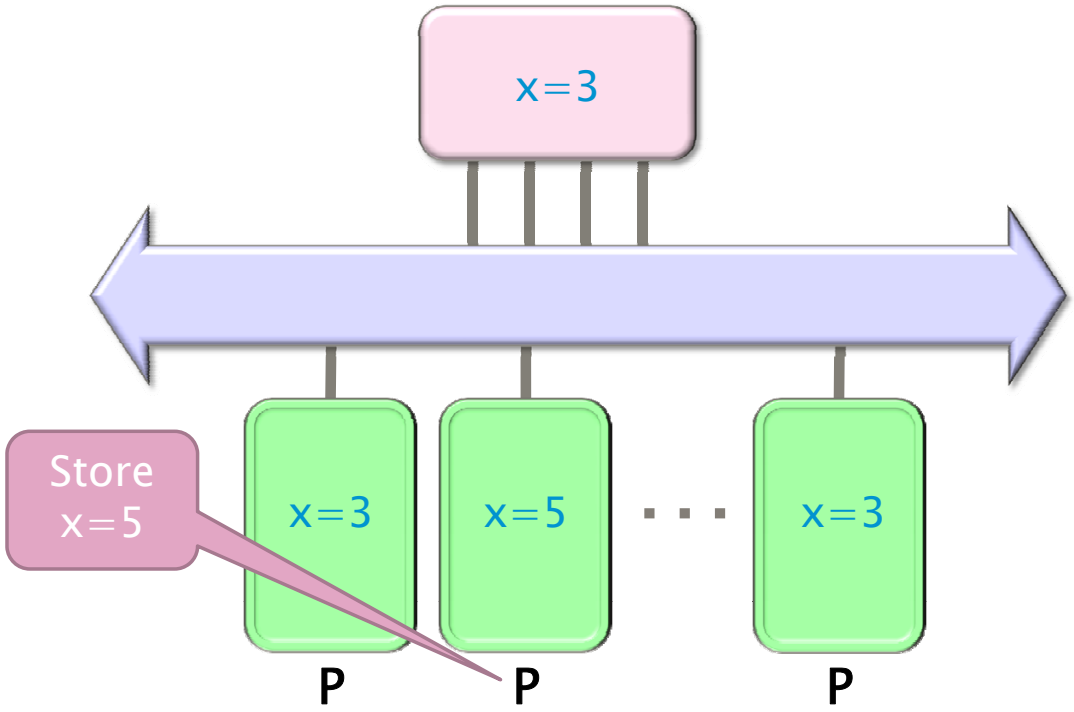


Figure: Processor P_2 writes x=5 in his local cache

Cache Coherence (6/6)

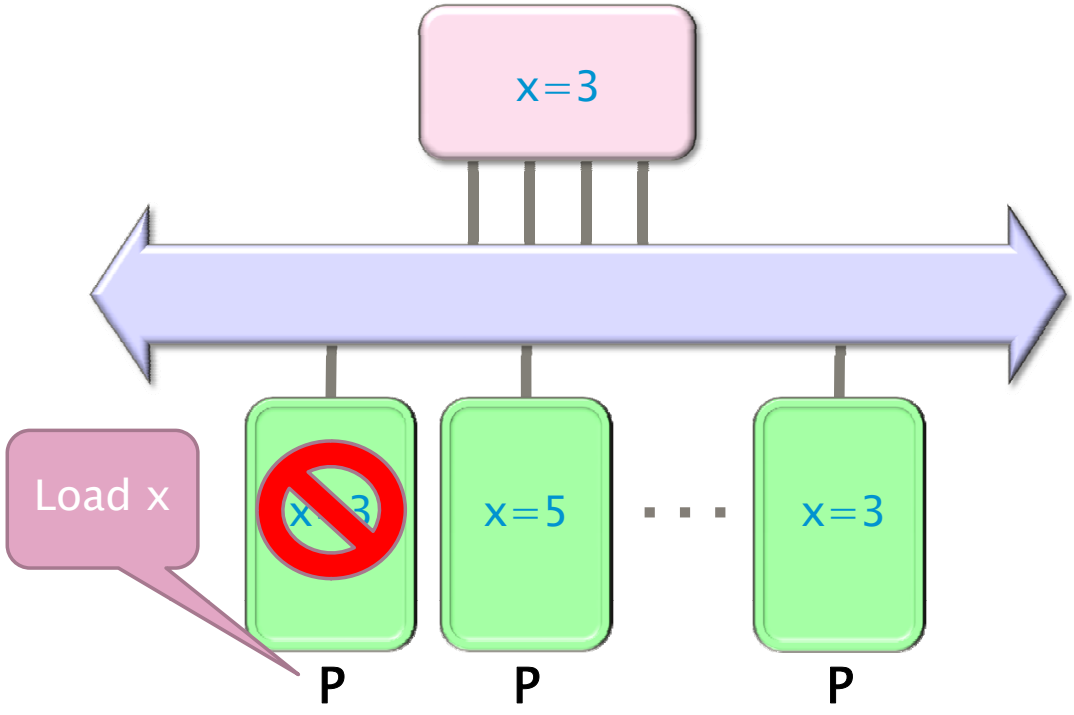


Figure: Processor P_1 issues a read x, which is now invalid in its cache

True Sharing and False Sharing

- **True sharing:**
  - True sharing cache misses occur whenever two processors access the same data word
  - True sharing requires the processors involved to explicitly synchronize with each other to ensure program correctness.
  - A computation is said to have **temporal locality** if it re-uses much of the data it has been accessing.
  - Programs with high temporal locality tend to have less true sharing.
- **False sharing:**
  - False sharing results when different processors use different data that happen to be co-located on the same cache line
  - A computation is said to have **spatial locality** if it uses multiple words in a cache line before the line is displaced from the cache
  - Enhancing spatial locality often minimizes false sharing
- See *Data and Computation Transformations for Multiprocessors* by J.M. Anderson, S.P. Amarasinghe and M.S. Lam
  `http://suif.stanford.edu/papers/anderson95/paper.html`

Multi-core processor (cntd)

- **Advantages:**
  - Cache coherency circuitry operate at higher rate than off-chip.
  - Reduced power consumption for a dual core vs two coupled single-core processors (better quality communication signals, cache can be shared)
- **Challenges:**
  - Adjustments to existing software (including OS) are required to maximize performance
  - Production yields down (an Intel quad-core is in fact a double dual-core)
  - Two processing cores sharing the same bus and memory bandwidth may limit performances
  - High levels of false or true sharing and synchronization can easily overwhelm the advantage of parallelism

Plan

From Cilk to Cilk++ and Cilk Plus

- `Cilk` has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo.
- Besides being used for research and teaching, `Cilk` was the system used to code the three world-class chess programs: Tech, Socrates, and Cilkchess.
- Over the years, the implementations of `Cilk` have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon.
- From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became Cilk Plus, see `http://www.cilk.com/`
- `Cilk++` can be freely downloaded at `http://software.intel.com/en-us/articles/download-intel-ci`
- `Cilk` is still developed at MIT `http://supertech.csail.mit.edu/cilk/`

Cilk++ (and Cilk Plus)

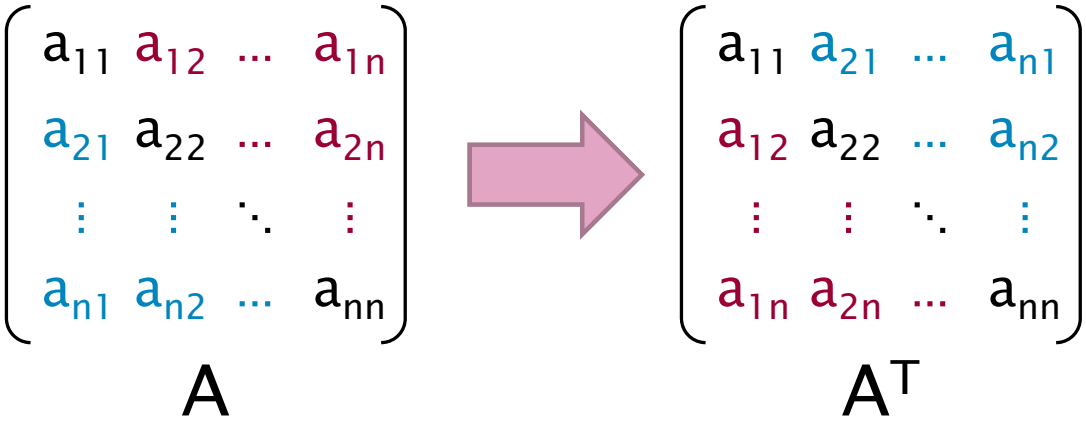
- `Cilk++` (resp. `Cilk`) is a small set of linguistic extensions to `C++` (resp. `C`) supporting fork-join parallelism
- Both `Cilk` and `Cilk++` feature a provably efficient work-stealing scheduler.
- `Cilk++` provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation.
- `Cilk++` includes the `Cilkscreen` race detector and the `Cilkview` performance analyzer.

Nested Parallelism in Cilk ++

```
int fib(int n)
{
    if (n < 2) return n;
    int x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return x+y;
}
```

- The named child function `cilk_spawn fib(n-1)` may execute in parallel with its parent
- Cilk++ keywords `cilk_spawn` and `cilk_sync` grant permissions for parallel execution. They do not command parallel execution.

Loop Parallelism in Cilk ++




```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
    for (int j=0; j<i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

The iterations of a `cilk_for` loop may execute in parallel.

Serial Semantics (1/2)

- `Cilk` (resp. `Cilk++`) is a multithreaded language for parallel programming that generalizes the semantics of `C` (resp. `C++`) by introducing linguistic constructs for parallel control.
- `Cilk` (resp. `Cilk++`) is a faithful extension of `C` (resp. `C++`):
  - The `C` (resp. `C++`) elision of a `Cilk` (resp. `Cilk++`) is a correct implementation of the semantics of the program.
  - Moreover, on one processor, a parallel `Cilk` (resp. `Cilk++`) program scales down to run nearly as fast as its `C` (resp. `C++`) elision.
- To obtain the serialization of a `Cilk++` program

```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Serial Semantics (2/2)

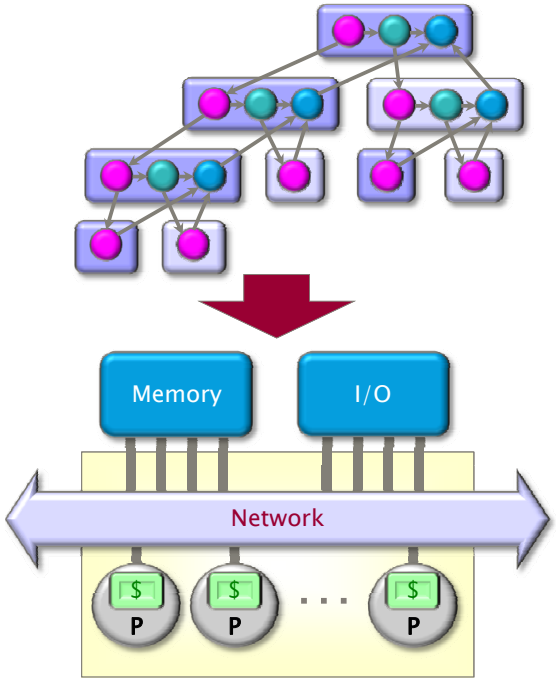
```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Cilk++ source

```
int fib (int n) {
if (n<2) return (n);
  else {
    int x,y;
    x = fib(n-1);
    y = fib(n-2);
    return (x+y);
  }
}
```

Serialization

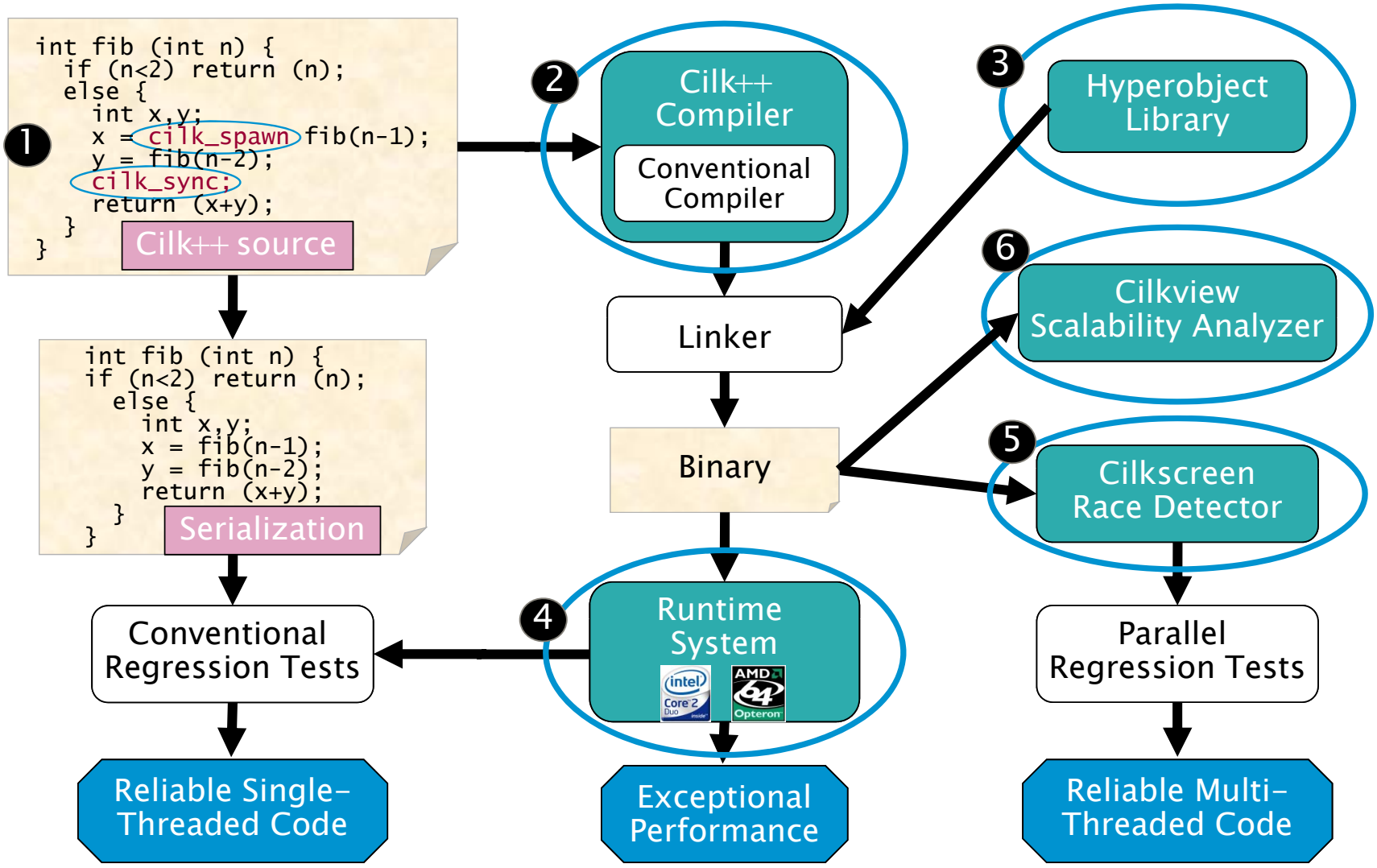
Scheduling



A **scheduler**'s job is to map a computation to particular processors. Such a mapping is called a **schedule**.

- If decisions are made at runtime, the scheduler is *online*, otherwise, it is *offline*
- `Cilk++`'s scheduler maps strands onto processors dynamically at runtime.

The Cilk++ Platform



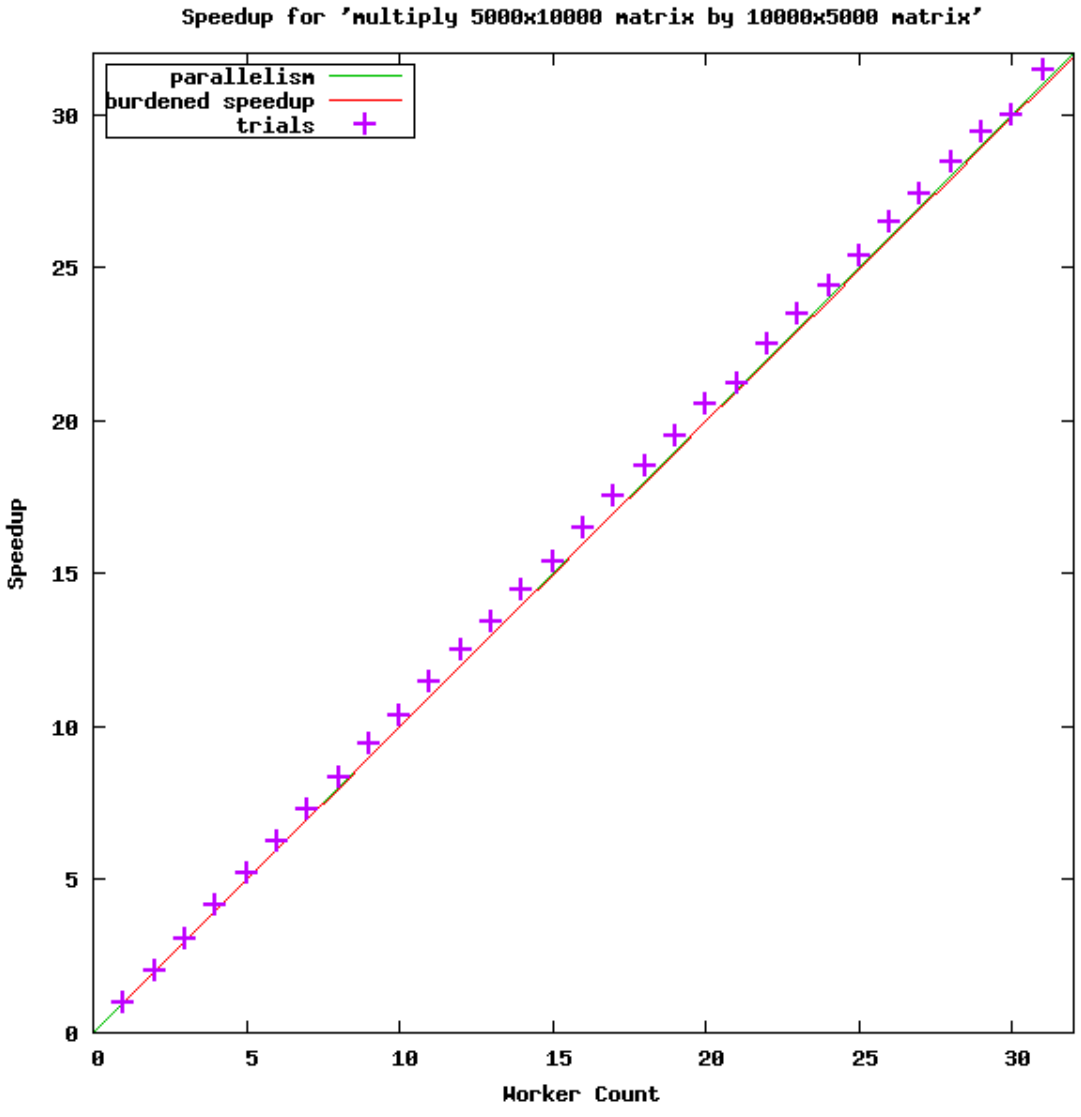
Benchmarks for the parallel version of the cache-oblivious mm

Multiplying a 4000x8000 matrix by a 8000x4000 matrix

- on 32 cores = 8 sockets x 4 cores (Quad Core AMD Opteron 8354) per socket.
- The 32 cores share a L3 32-way set-associative cache of 2 Mbytes.

#core	Elision (s)	Parallel (s)	speedup
8	420.906	51.365	8.19
16	432.419	25.845	16.73
24	413.681	17.361	23.83
32	389.300	13.051	29.83

So does the (tuned) cache-oblivious matrix multiplication



Plan

Course Topics

Week 1: Introduction to UNIX (command lines, editors)

Week 2: Introduction to C (basic types, flow of control, expressions, functions)

Week 3: UIX Fundamentals (permissions, job control, redirections)

Week 4: Arrays and pointers in C

Week 5: UNIX Regular expressions, shell programming

Week 6: Advanced topics on pointers and files in C

Weeks 7-8: Introduction to Multicore Programming

Week 9-10: Multithreaded Parallelism and Performance Measures

Week 11: Analysis of Multithreaded Algorithms

Weeks 12: Issues with code parallelization anddata locality

Week 13: Synchronizing without Locks and Concurrent Data Structures

About this course

- Prerequisites: no CS courses, but familiarity with a programming language. Knowledge of linear algebra, linear recurrences is assumed.
- Objectives: introduce students to *performance software enginnering*.
- Methods: build a strong knowledge in C/UNIX, then study multicore programming in Cilk.
- We will cover a large of materials and we will have tutorial every week.