Modern Architectures, Profiling and Optimization

Luca Tornatore - I.N.A.F.

Scientific and high performance computing school 2019

Università degli Studi di Trento

Introduction

High Performance Computing requires, by the name itself, to squeeze as much effectiveness from your code on the machine you run it.

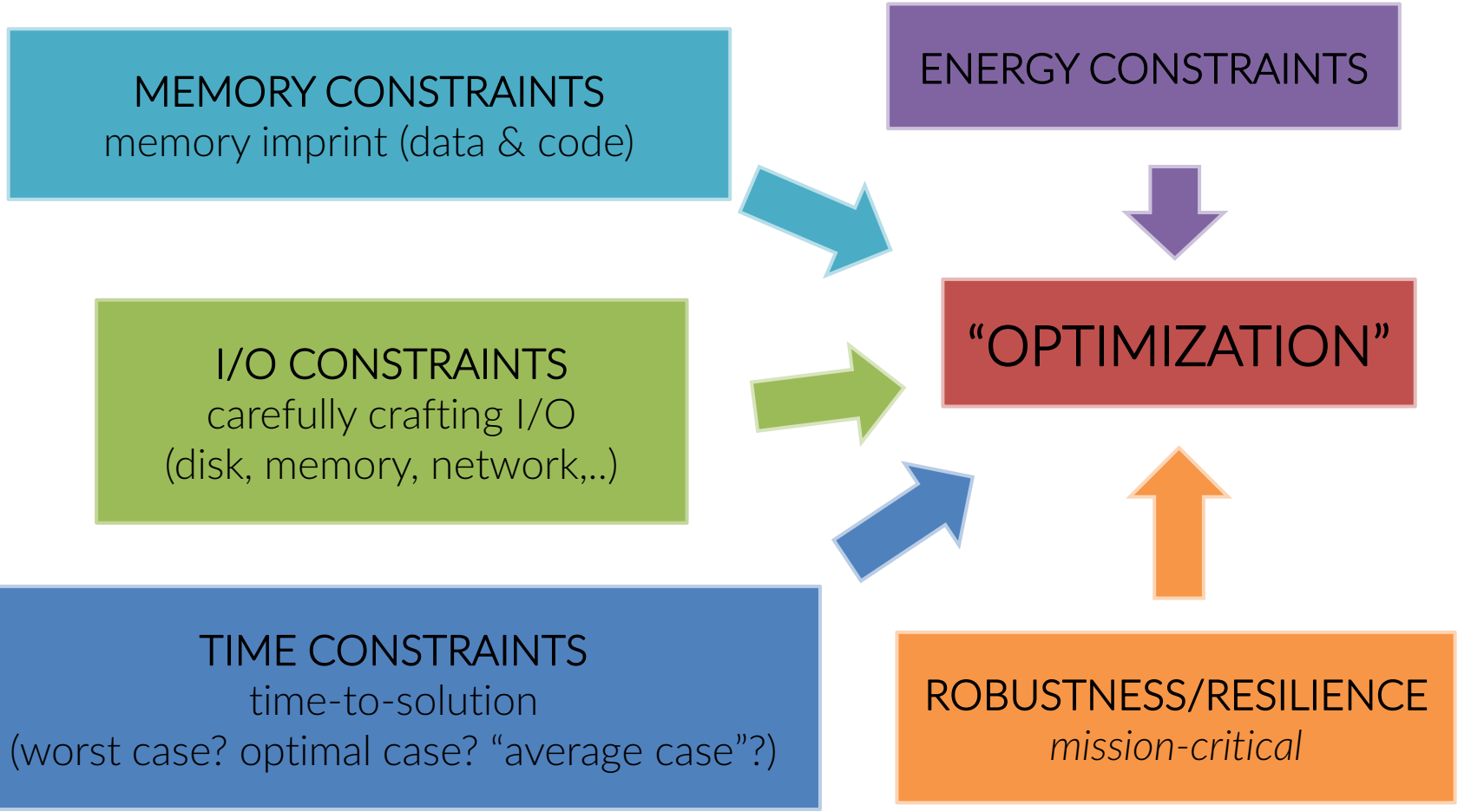
“Optimizing” is, obviously, a key step in this process.

However, both “*optimizing*” and “*effectiveness*” have to be defined. And that is what this lecture is about.

Let’s start with some general comments.

There may be no such thing as an “optimal” code *tout court*

Just try to define “optimal”..


MEMORY CONSTRAINTS
memory imprint (data & code)
ENERGY CONSTRAINTS
I/O CONSTRAINTS
carefully crafting I/O
(disk, memory, network,..)
"OPTIMIZATION"
TIME CONSTRAINTS
time-to-solution
(worst case? optimal case? "average case"?)
ROBUSTNESS/RESILIENCE
mission-critical

Premature optimization is the root of all evil

D. Knuth

You'd better start thinking in terms of "***improved***" code.

- You don't want to hurt your code that honestly does its job: a faster code that gives incorrect results is not optimal in *any* way
- You most probably will have to re-use that code after some time. And you will need it to be *readable*, *maintainable* and *well-designed*.
- *Others* most probably will have to read, understand and re-use that code: a clean, understandable, non-obscure, non-redundant code may not be "improved" in some way, but actually improves the quality of your life.

First steps to consider

To have a **cleaner code**

To **re-design** its fundamental architecture

To **improve** the workflow, the memory imprint, the resource usage, the time-to-solution, …

Dryness

DRY: Don't Repeat Yourself

Do neither add **unnecessary** code nor **duplicate** code

Unnecessary code increases the amount of needed work

- to maintain the code, either debugging or updating it
- to extend its functionalities

Duplicated code increases your *bad technical debt*, that already has a large enough number of sources:

- copy-and-paste programming style
- too much *agile* approach
- hard coding, quick-and-dirty fixes, cargo-cult, sloppiness

Readability

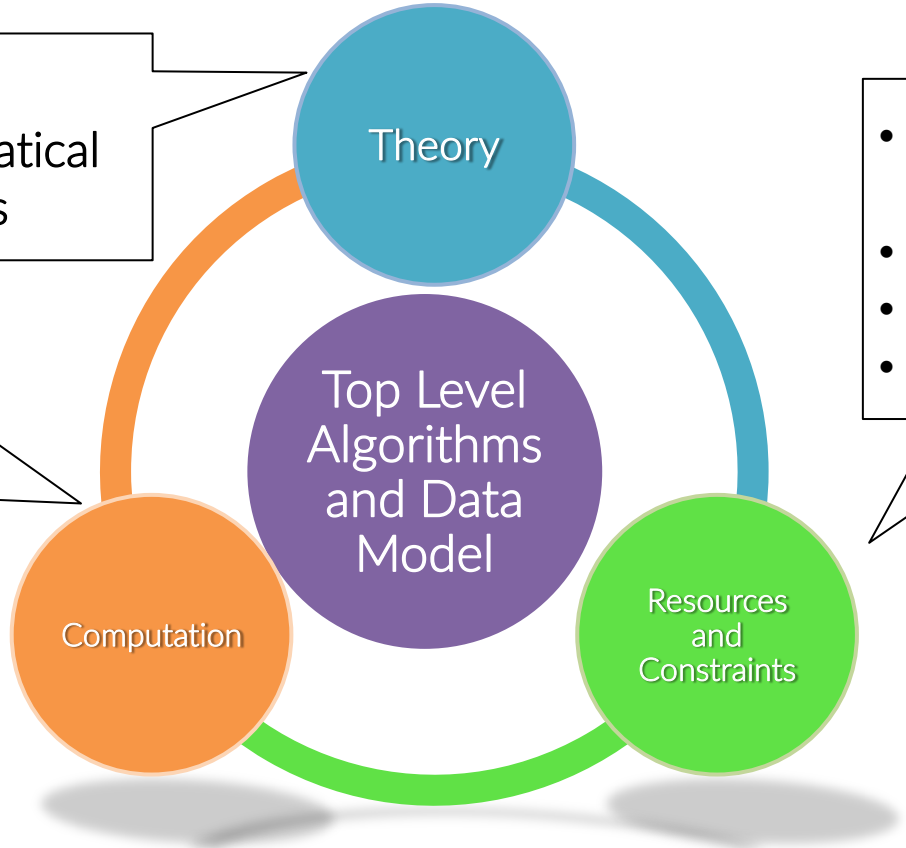
Readability is a pillar of **maintainability**.
Maintainability means that software can be extended, upgraded, debugged, fixed.

- **Baseline**: *write comments*
- **Advanced**: *WRITE COMMENTS !* (possibly, use doxygen-like tools)
- **X-advanced**: avoid stupid, obvious, useless, confusing comments

Keep in mind: "*the code is the ultimate man page*"

Design

- Consult literature
- Understand mathematical and scientific aspects



- Concurrency & parallelism
- Memory constrs.
- I/O constrs.
- Energy constrs.

You're not at the blackboard, but on a digital discrete system. Equations have to *translated*.

Design

ARCHITECTURAL DESIGN

Highest abstract picture:
- Interacting multi-components
- Top-level DATA MODEL

HIGH-LEVEL DESIGN

Sub-systems and modules Communication strategies

Modules as producers of complex tasks

DETAILS DESIGN

Implementation details
- many algorithms
- concurrency

Logical structure of modules

interactions

Very low-level coding and details

ABSTRACTION

CODING DETAIL

Design

TESTING IS part of the design

- Unit test
  separately stress each unit of the code
- Integration test
  stress the integrated behavior of all the units together
- System test

Design

VALIDATION and VERIFICATION ARE part of the design

- Validation ensures that the code does what it was meant to do
  (all modules are designed accordingly to design specifications)
- Verification ensures that the codes does what it does *correctly*
  (black-box testing against test-cases, …)

Improve the code

Improving the workflow, the memory imprint, the resource usage, the time-to-solution, …

Well, that is the focus of this day :)

Outline

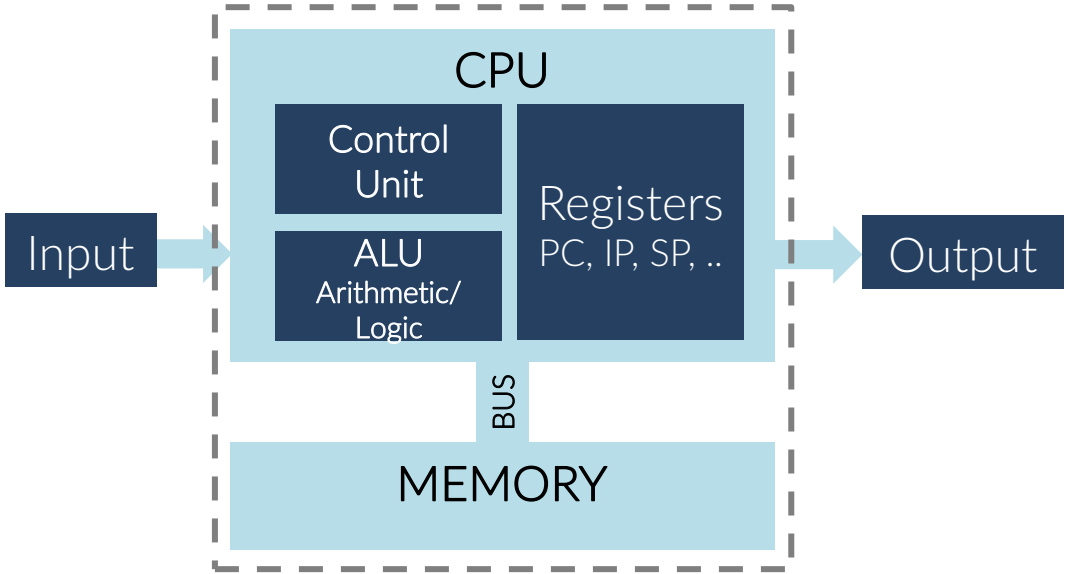
Modern Architecture & Optimization

Profiling

Debugging

Once upon a time..

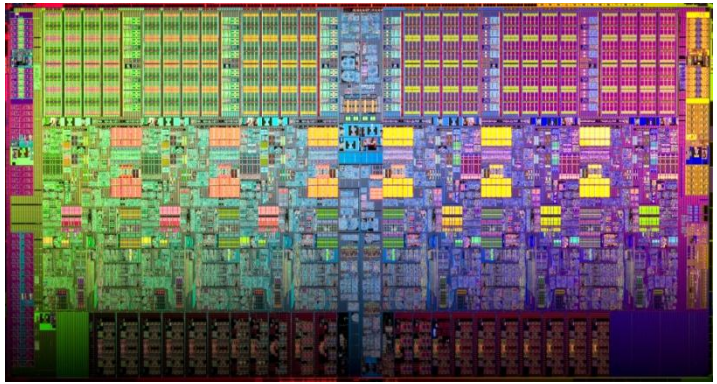
.. computer architecture was much simpler than today.
The exact reasons why it was so – or, better, why nowadays it is much more complex – will be discussed tomorrow.



In the Von Neumann architecture (still taught as the fundamental model)

- there is only 1 processing unit
- 1 instructions is executed at a time
- memory is “flat”:
  - access on any location has always the same cost
  - access to memory has the same cost than op execution

While today…



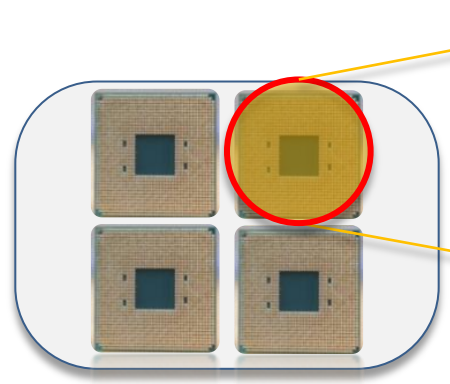
6-cores Intel Xeon e5600

(*) "cost" is in the CPU-cycles currency

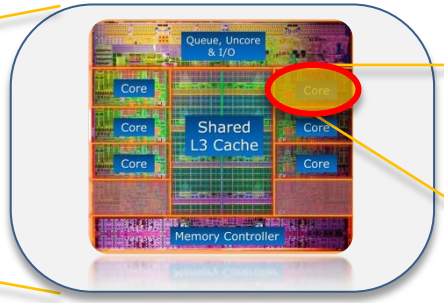
- there *many* processing units
- many instructions can be executed at a time
- many data can be processed at a time
- "instructions" are internally broken down in many simpler µops that are pipelined
  - different instructions could have quite different cost (*)
- memory is strongly *not* "flat":
  - there is a strong memory hierarchy (*)
  - access memory can have *very* different costs (*) depending on the location
  - accessing RAM is *way* more costy than performing a µop or reading an internal register

In the next slides we'll go through all this features in chronological order, as they developed in time

Overview



sockets



Many cores per socket

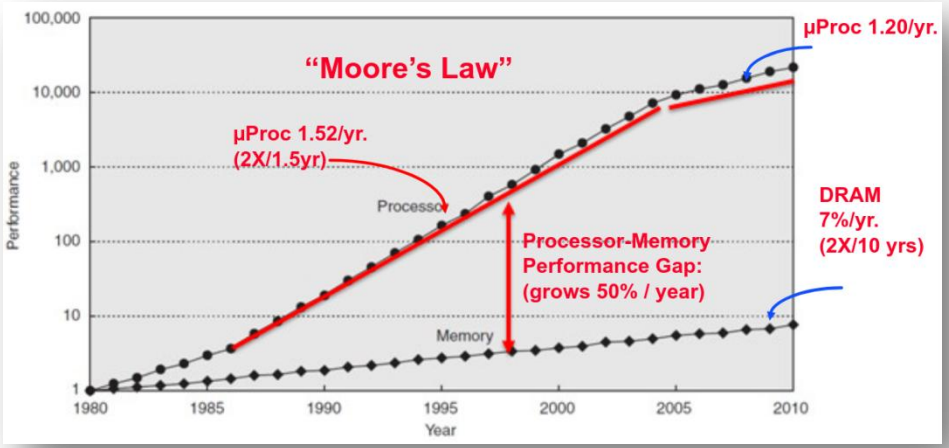
Many threads

Vectorisation

Pipelining

Superscalarity

Early 90s: CPU get faster than memory



Processor	Alpha 21164	
Machine	AlphaServer 8200	
Clock Rate	300 MHz	
Memory Performance	Latency	Bandwidth
I Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
D Cache (8KB on chip)	6.7 ns (2 clocks)	4800 MB/sec
L2 Cache (96KB on chip)	20 ns (6 clocks)	4800 MB/sec
L3 Cache (4MB off chip)	26 ns (8 clocks)	960 MB/sec
Main Memory Subsystem	253 ns (76 clocks)	1200 MB/sec
Single DRAM component	≈60ns (18 clocks)	≈30–100 MB/sec

Taken from a 1997 paper

The CPU may spend more time waiting for data coming from RAM than executing ops.
That is part of the so called "memory wall".
What is the solution ?

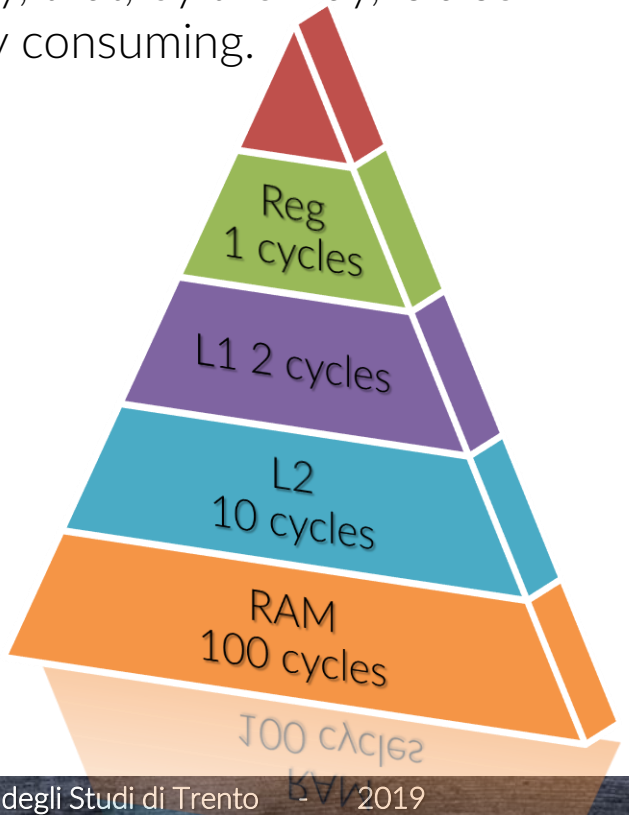
CPU-RAM gap

The solution is to equip your CPU with a *faster* memory, that, by the way, is also *way* more costy (in terms of $/B) and *way* more energy consuming.

Furthermore, to be faster it ought to be *extremely closer*.
All in all, the new memory that will be called ***cache***, will be much *smaller* than RAM.

The cache itself has a hierarchy:

- *Level-I* is inside each core.
- *Level-II* is also inside the core, or may be shared by more cores.
- *Level-III* is inside the CPU, shared by many cores.



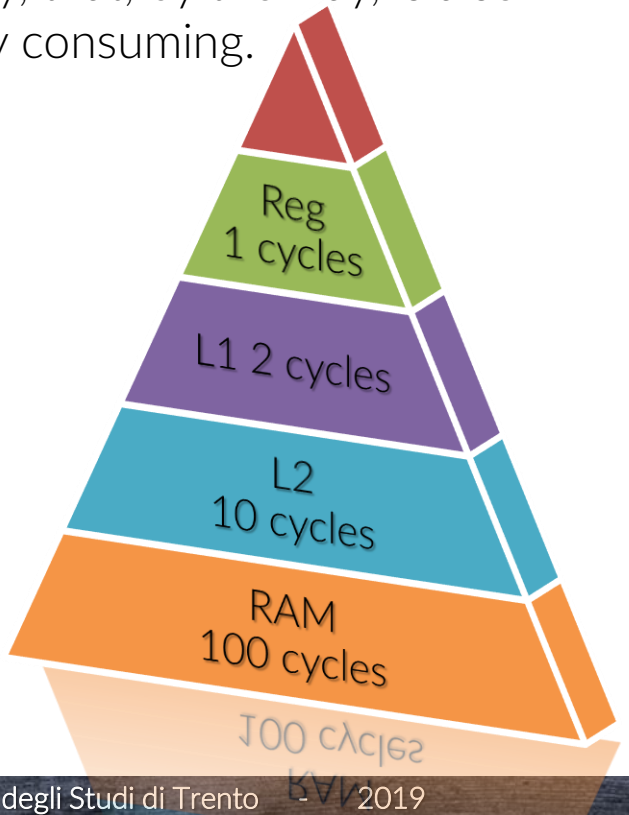
CPU-RAM gap

The solution is to equip your CPU with a *faster* memory, that, by the way, is also *way* more costy (in terms of $/B) and *way* more energy consuming.

Furthermore, to be faster it ought to be *extremely closer*.
All in all, the new memory that will be called ***cache***, will be much *smaller* than RAM.

The cache itself has a hierarchy:

- *Level-I* is inside each core.
- *Level-II* is also inside the core, or may be shared by more cores.
- *Level-III* is inside the CPU, shared by many cores.



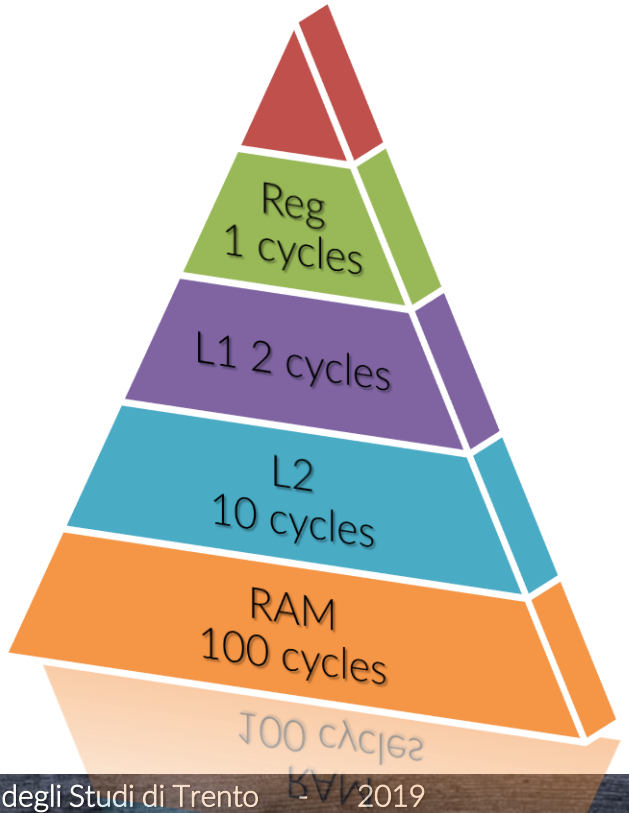
CPU-RAM gap

L1 cache + RAM

$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

} 50% to 150% slower



CPU-RAM gap

L1 cache + RAM

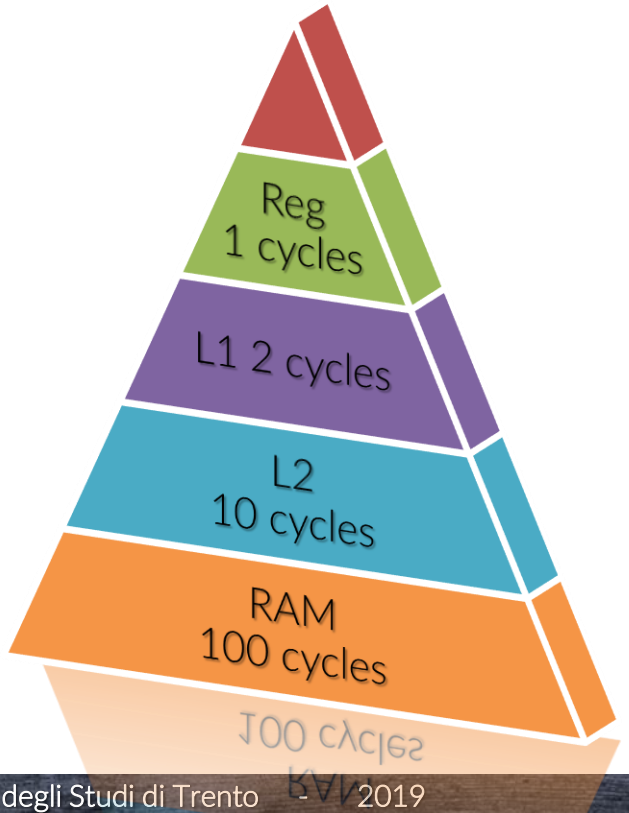
$L_{1cost} + Miss_1 \times RAM_{cost}$

- 100% L1 hit → 2 cycles
- 99% L1 hit → 3 cycles
- 97% L1 hit → 5 cycles

L1 cache + L2 cache + RAM

$L_{1cost} + Miss_1 \times (L2_{cost} + Miss_2 \times RAM_{cost})$

- 100% L1 hit → 2 cycles
- 99% L1 hit, 100% L2 hit → 2.1cycles
- 97% L1 hit, 100% L2 hit → 2.3 cycles
- 90% L1 hit, 97% L2 hit → 3.3 cycles



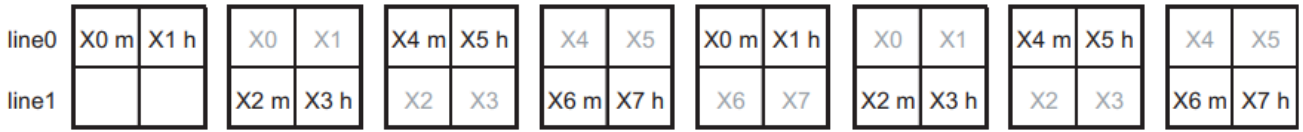
CPU-RAM gap | the memory access pattern

When the cache is hit and when it is not: a simple model

Consider a simple direct mapped **16 byte data cache** with **two cache lines**, each of size 8 bytes (two floats per line)

Consider the following code sequence, in which the array **X** is cache-aligned (that is, **X[0]** is always loaded into the beginning of the first cache line) and accessed twice in consecutive order:

```
float X[8];

for(int j=0; j<2; j++)
   for(int i=0; i<8; i++)
     access(X[i]);
```



The hit-miss pattern is : MH MH MH MH MH MH MH MH,
the miss-rate is 50% (the first miss is compulsory miss).

CPU-RAM gap | the memory access pattern

Let's consider another code sequence that access the array twice as before, but with a strided access

```
float X[8];
for(int j=0; j<2; j++)
   {
      for(int i=0; i<7; i+=2)
         access(X[i]);
      for(int i=1; i<8; i+=2)
         access(X[i]);
   }
```

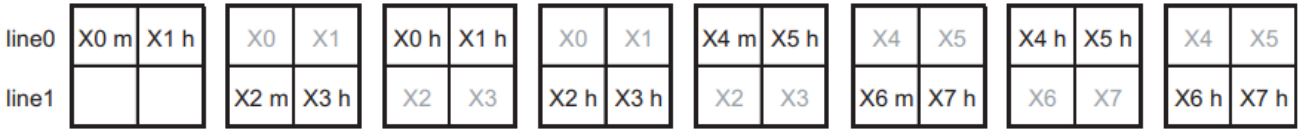
line0	X0 m	X1	X4 m	X5	X0	X1 m	X4	X5 m	X0 m	X1	X4 m	X5	X0	X1 m	X4	X5 m
line1	X2 m	X3	X6 m	X7	X2	X3 m	X6	X7 m	X2 m	X3	X6 m	X7	X2	X3 m	X6	X7 m

The hit-miss pattern now is : MM MM MM MM MM MM MM MM,
the miss-rate is 100%

CPU-RAM gap | the memory access pattern

Finally, consider a third code sequence that again access the array twice:

```
for(int i = 0; i < 2; i++)
   for(int k = 0; k < 2; k++)
      for(int j = 2*i; j < (i+1)*2; j ++)
         access(X[ j ]);
```

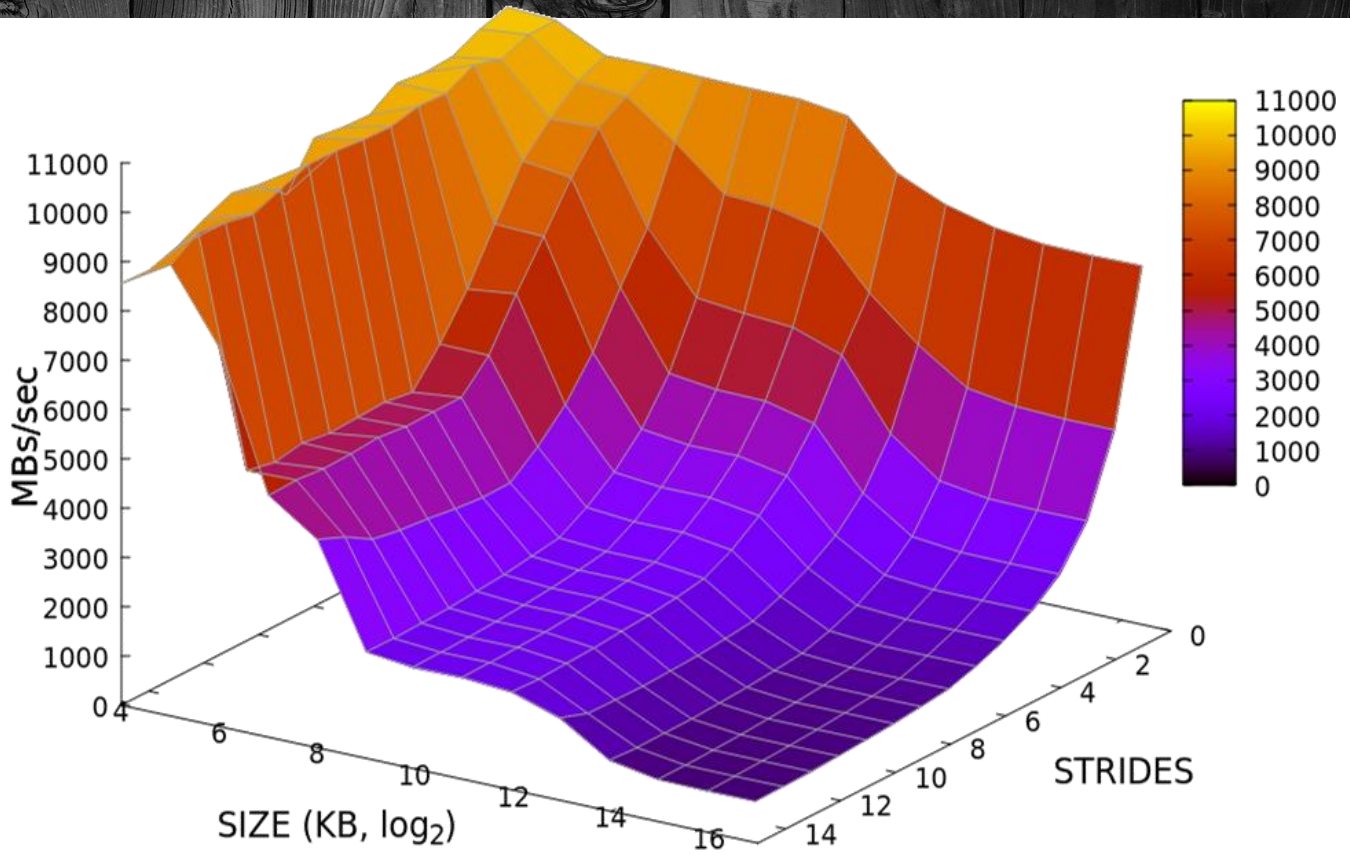


The hit-miss pattern now is : MH MH HH HH MH MH HH HH,
the miss-rate is 25%

The main message is: memory access pattern is of primary importance

CPU-RAM gap | the memory access pattern

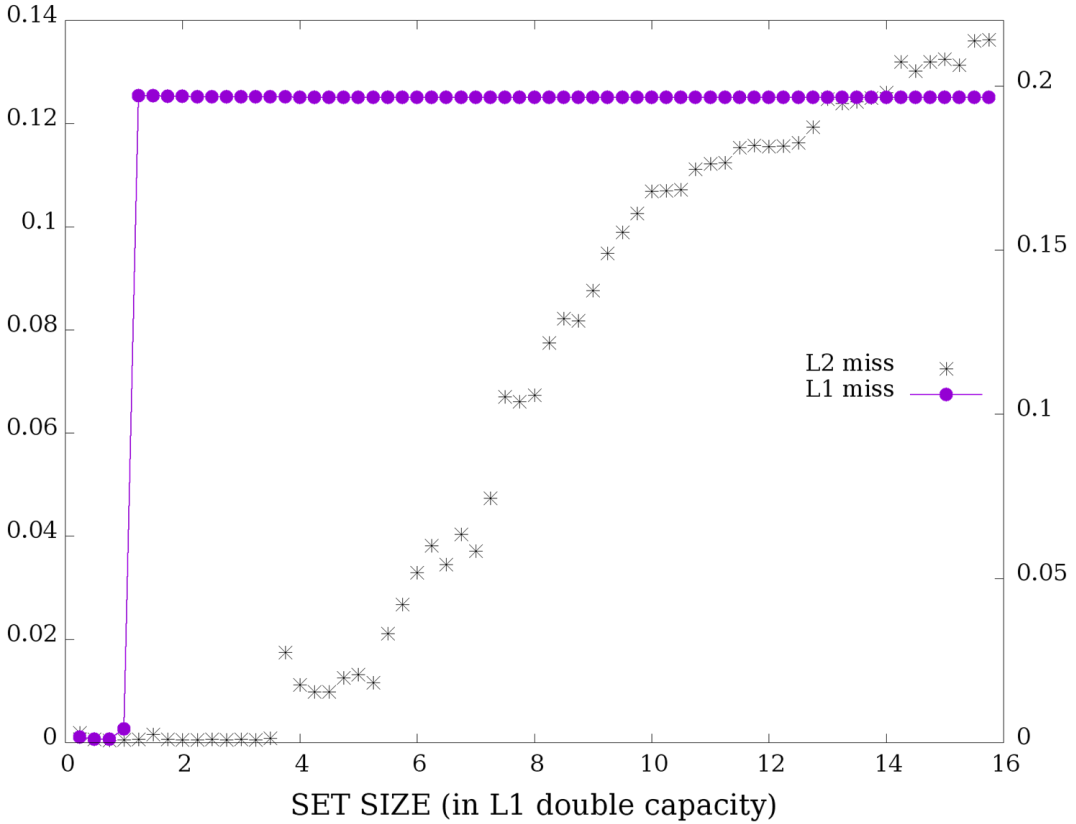
The result is..
the memory mountain



CPU-RAM gap | the memory access pattern

Let's find out our cache size

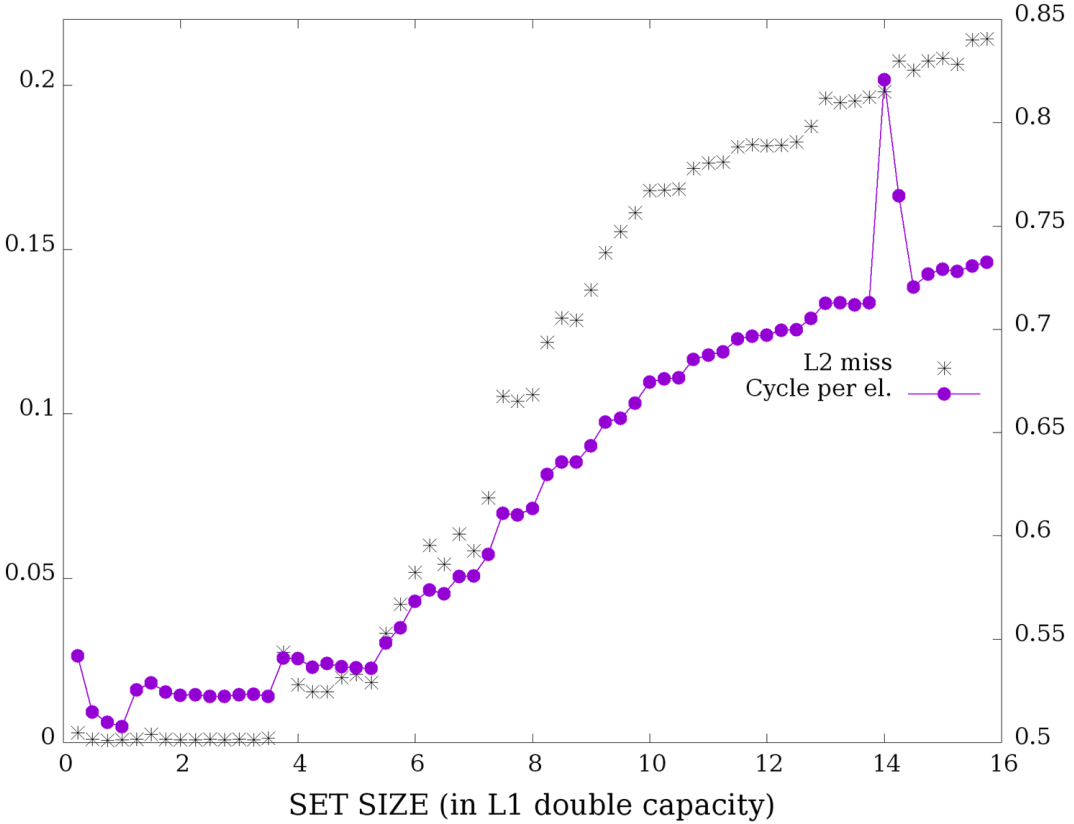
```
for (j=0; j<size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



CPU-RAM gap | the memory access pattern

And the effect on cycles-per-operation metrics

```
for (j=0; j<size; j++)
   array[j] =
     2.3*array[j]+1.2;
```



CPU-RAM gap | the memory access pattern

Matrix transpose

Naïve version:

```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

NOTE: strided access to either **A** or **B** is unavoidable.

However: is it better to have it either on *read* or on *write ?*

CPU-RAM gap | the memory access pattern

Matrix transpose

Naïve version:

```
for(int row = 0; row < N; row++)
    for(col = 0; col < N; col++)
        A [ col*N + row ] = B [ row*N + col ];
```

NOTE: strided access to either **A** or **B** is unavoidable.

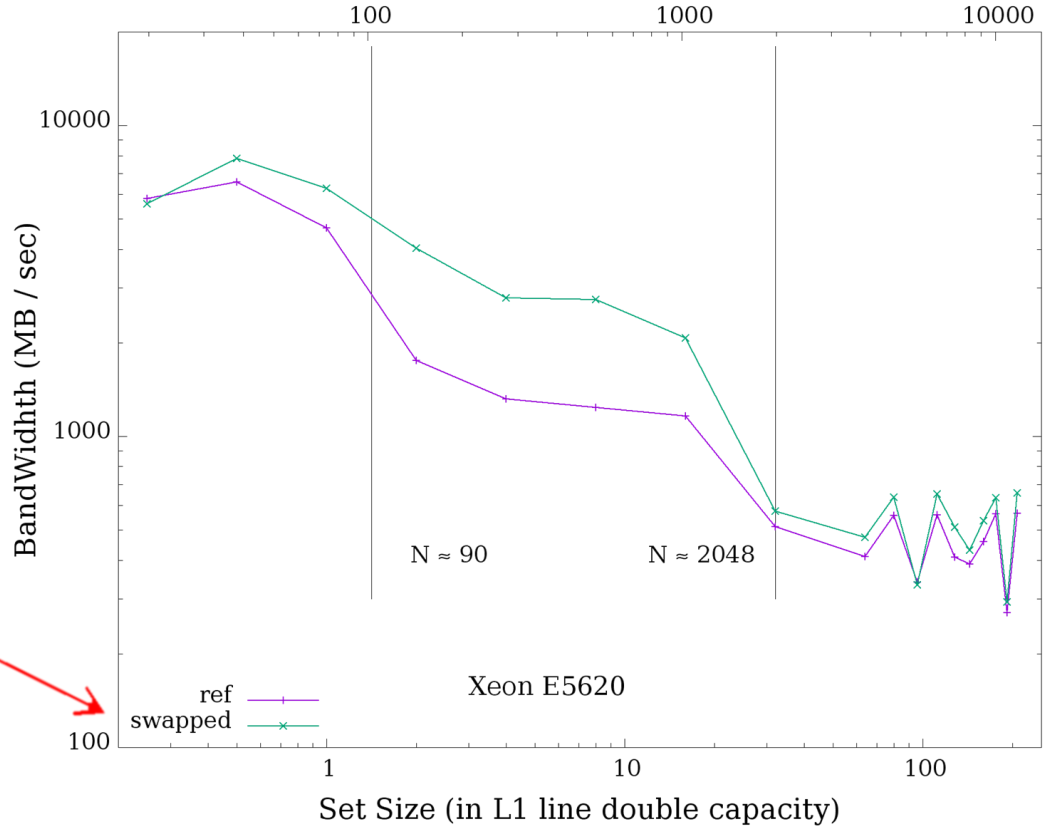
However: is it better to have it either on *read* or on *write* ?

Due to write-allocate transactions in the cache, strided writes are more expensive than strided loads.

CPU-RAM gap | the memory access pattern

Inner cycle flipped:

```
A [ row*N + col ] =
    B [ col*N + row ];
```



Due to time constraints, we can't go in deeper details. Ask if interested, though.

Reorder fields in structures so that what is used together stays together

Linked-list node

```
struct my_node {
   double   key;
   char     my_data[300];
   my_node *next_node; }
```

Linked-list traversal

```
void myfunc(my_node *p, double key, <…>)
{
   while( p != NULL) {
         if( p->key == key ) {
               do_something( <…> );
               break;
            }
         p = p → next_node;
      }
}
```

CPU-RAM gap | fields reordering in structures

Reorder fields in structures so that what is used together stays together

```
struct my_node
{
  double    key;
  char      my_data[300];
  my_node *next_node;
}
```

➔

```
struct my_node
{
  double    key;
  my_node *next_node;
  char      my_data[300];
}
```

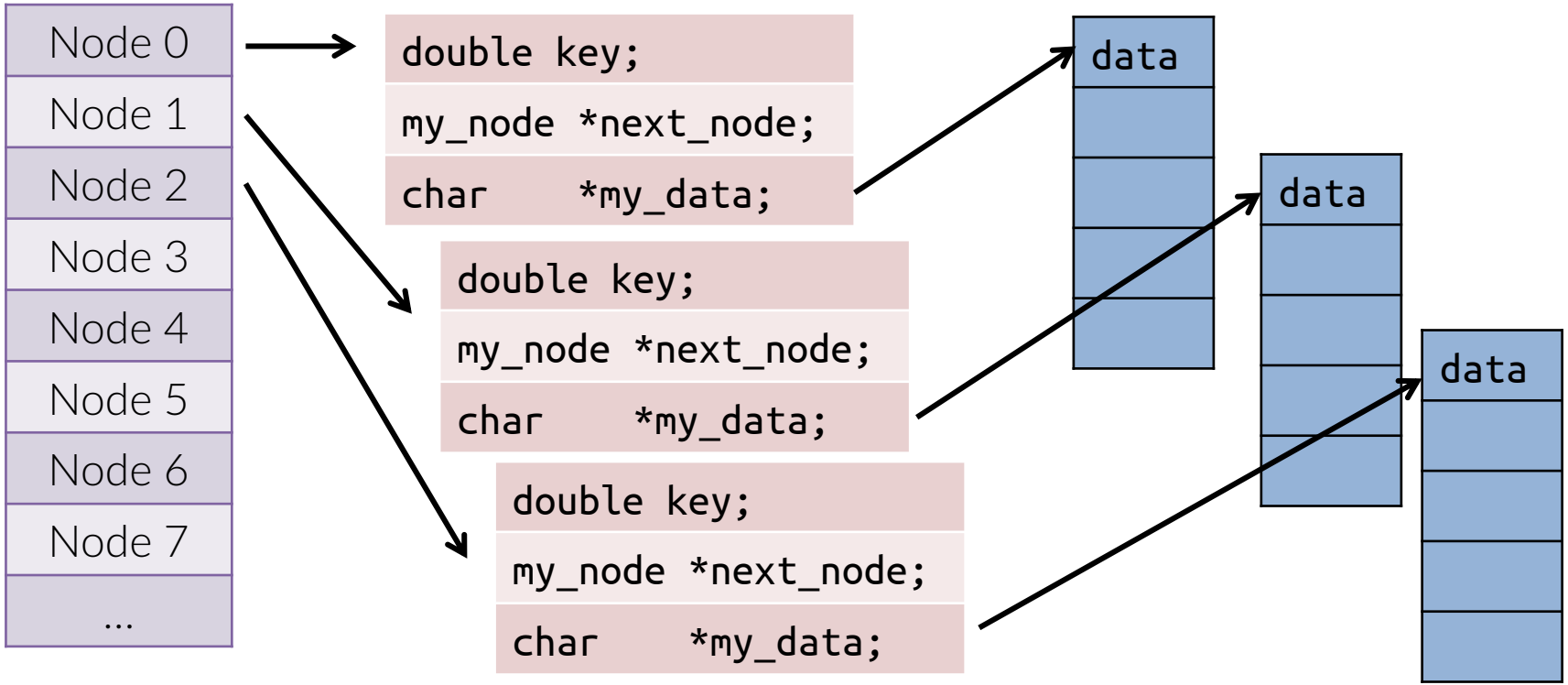
CPU-RAM gap | fields reordering in structures

Split fields so that to keep consecutive the fields that are used sequentially

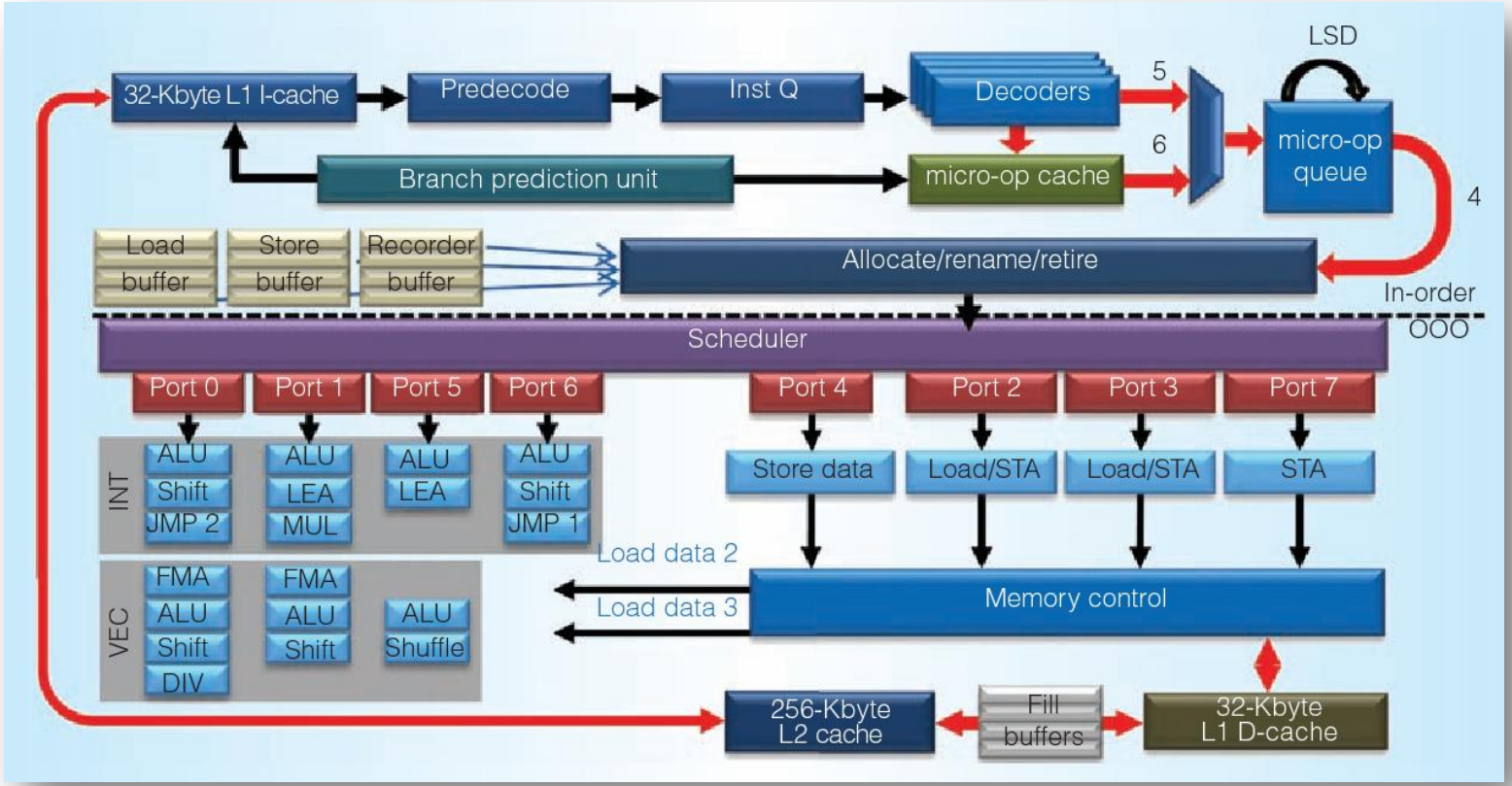
```
struct my_node
{
  double   key;
  char     my_data[300];
  my_node *next_node;
}
```

```
struct my_node
{
  double   key;
  my_node *next_node;
  void    *my_data;
}

struct my_data
{
  char data[300];
}
```

Those are called *hot* and *cold* fields

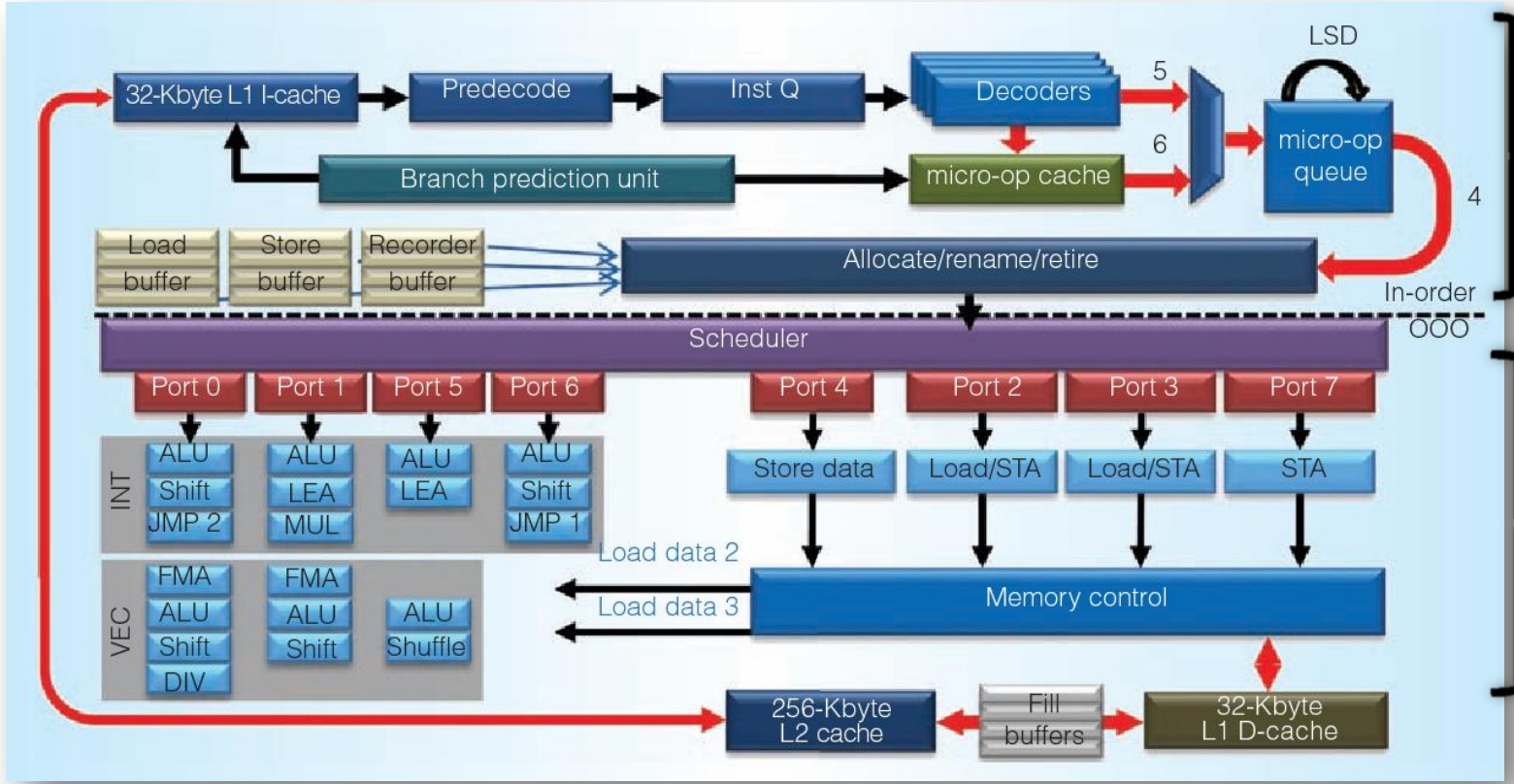


Mid 90s: superscalar and out-of-order CPUs



6th generation SkyLake micro-arch.

More than 1 *port* is available to execute CPU instructions, although different units have different specializations (ALU, LEA, SHIFT, FMA, …) : that is superscalar capacity, i.e. the capacity of executing more than 1 instructions per cycle.

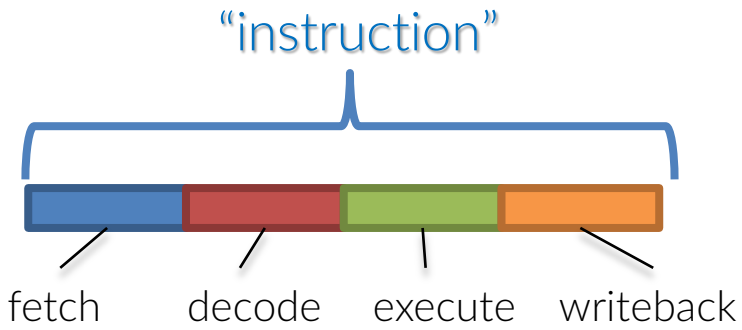


6th generation SkyLake micro-arch.

Pipelines

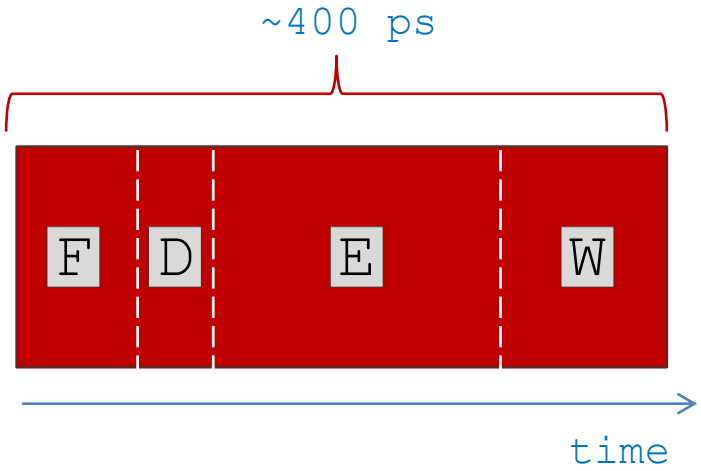
It would be obvious to think that an "instruction" is a kind of *atomic* operation that the CPU perform as a whole. Indeed that was true until the mid of the 80s.

If you think carefully about it, it is easy to understand that actually an "instruction" involves at least the following ***independent*** steps:



1. *Fetching*
   it must be recalled from memory/Icache
2. *Deconding:*
   it must be "understood and interpreted"
3. *Execution*
4. *Writeback :*
   the result must be accounted in memory ()

Pipelines

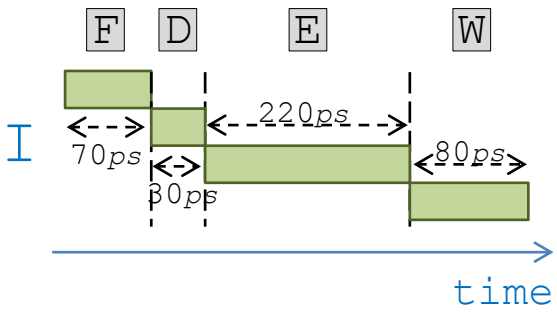


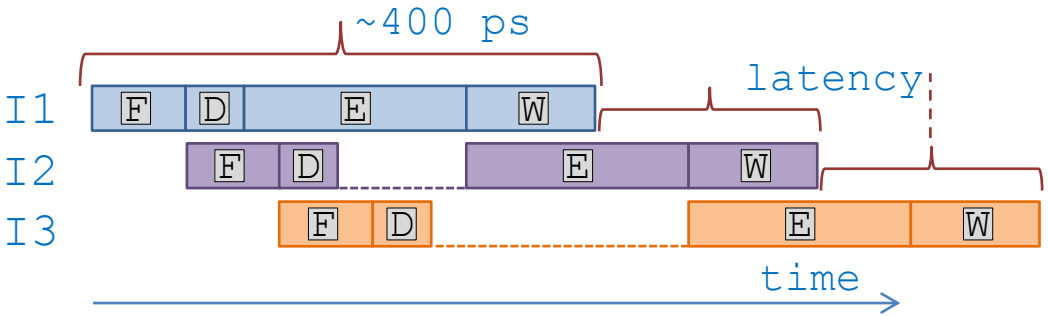
If all the four stages take ~400ps, we then would obtain a **throughput** of 2.5GIPS (giga-instructions per second).
400ps is also the total time required to get a result from an instruction, and so it is the **latency** of the instruction.

However, if we were able to "detach" the four stages, we could organize things differently, like in a car-building chain, or even at the mensa of the university.

Pipelines

Note: all the timing estimates are hypothetical for the purpose of discussing the concepts



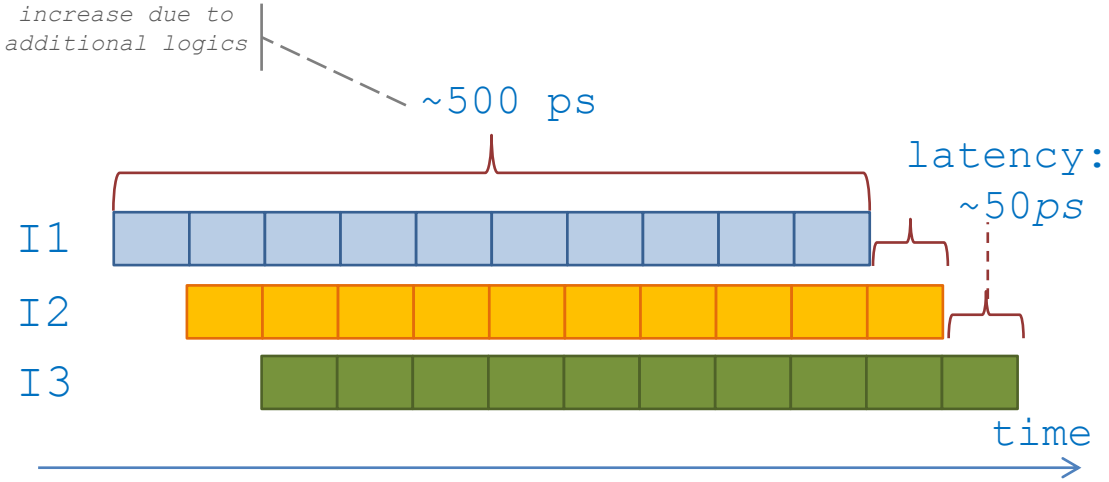


If many independent logical units exist to perform each step, they could operate subsequently on **different instructions**:

If the stage delays are not uniform, the throughput is limited by the latency `F+(D+E)-(F+D)` = `E` ~ 220*ps*, which means we have a throughput of ~**4.5GIPS** just because of logic units separation.

Pipelines

Therefore, introducing the instructions pipelining, we can increase the **throughput** of our system by a large factor.
However, the efficiency of the pipelines is limited by its longest stage: the better option would be to have all equal stages, for instance further subdividing each stage – especially the most demanding ones (*).



Now the throughput of our system has increased to 1 instruction retired every 50*ps*, i.e. 20GIPS

(*) this is called *superpipelining*: modern CPUs may have 10-20 stages per pipeline.

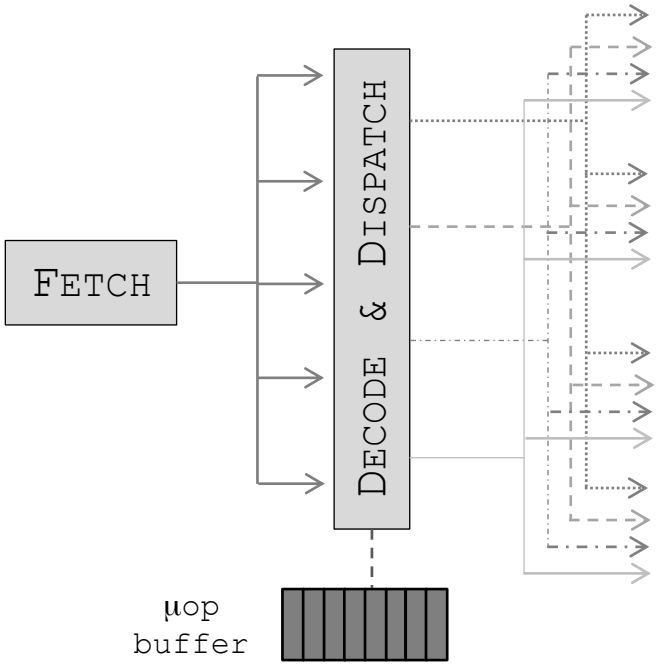
Pipelines

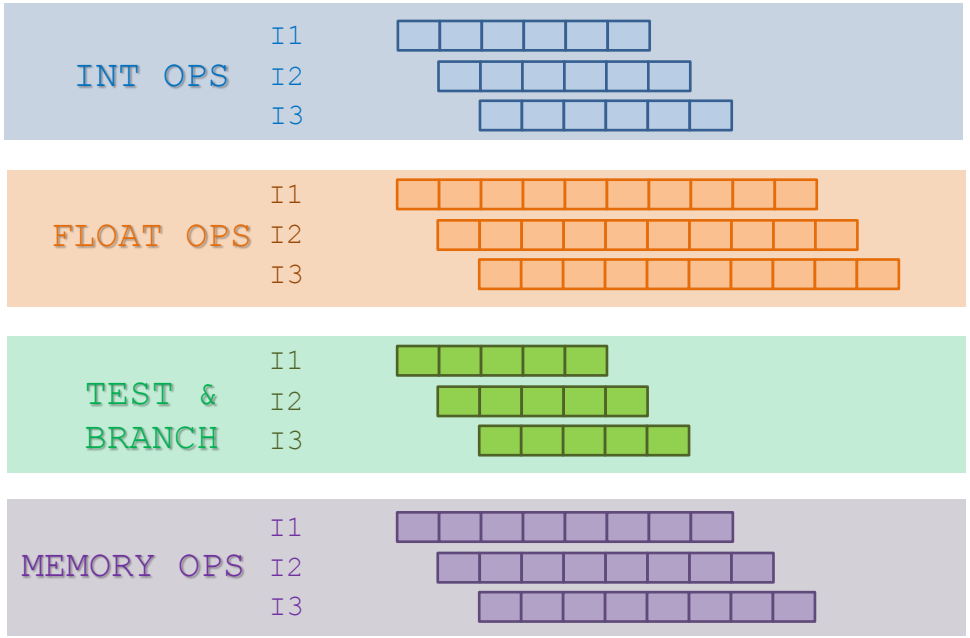
Pipelining is then about increasing the throughput of a system, and as we have seen it could be really effective.

However, there is more that can be done working on the **execution stage** that, of course, encompasses a large number of different ***functional units***, performing a **different and independent tasks**.

With an enhancement of the decode/dispatch stage, we can address multiple pipelines that can be active at the same time:

Pipelines


FETCH
DECODE & DISPATCH
µop buffer


INT OPS
I1
I2
I3
FLOAT OPS
I1
I2
I3
TEST & BRANCH
I1
I2
I3
MEMORY OPS
I1
I2
I3

Pipelines

The number of cycles between when an instruction's execution stage begins and when its result is available for other instructions usage, is the *latency* of the instruction and it grows with the pipeline deepness (the number of stages).

Typical latencies in modern processors (read the manual of yours) range from 1cyc for integer ops, to 3-6cyc for add and mul flop to ≥10-20 for div flop, ~50 for trig flop.

Latency for memory loads can be severely troublesome, because they are highly unpredictable and they block all other operations making it difficult to fill the delay with some other ops.

Pipelines hazards

Pipelining a system with *feedbacks* (i.e.: in which a result from an instruction can be the input of a following instruction, or can change the workflow) can expose the system to *hazards* when there are dependencies between successive instructions.

There are two forms of dependency:

1. *data dependency*: the result computed by the `i`*th* instruction are used by one or more following instructions;
2. *control dependency*: the result of an instruction determines the value of the `pc`, i.e. the location of the next instruction to be executed.

When a dependency can lead to a wrong result, it is called an *hazard*.

Pipelines hazards

If a control hazard arise, the entire pipeline content has to be flushed away, and a new instruction has to start from the beginning, then wasting a lot of CPU cycles.

This is one of the main cause of performance loss in modern superscalar-superpipeline-out-of-order CPUs. The logics for the runtime branch predictor occupies a large space on processor chips but it definitely is worth it.

Pipelines hazards

Best branch predictors are as good as 95% of accuracy: nonetheless, the branch mis-prediction, or branch miss, determines a huge performance loss: typical values for penalty are 10-20 cycles!

That is because the longer the pipeline, the further in the future you have to scrutinize the flow, the more difficult it is and the larger will be the misprediction penalty.

Example

Let's say we have 140 instructions on the flight, and 1 every 7 is a branching instructions. What is the probability that the pipelines shall not be flushed with 95% correct branch predictor? And with a 90% one?

~36% and ~12%

Pipelines hazards

A very long pipeline, then, is not much more effective than a shorter one due to the real intrinsic nature of the codes that run on the CPUs.

The hazards we have just described are actually very common (very rarely a program is a stream of totally independent instructions with no jumps) and so the full exploitation of superscalarity + superpipelining is never reached.

Basically, that's the reason why we do not have 100-stage deep pipelines.

And the reason why branching is a performance-killer, too.

Pipeline optimization

As we have seen, modern processor may be able to “perform more than one operations at a time”, through super-pipelining operations.

As for what concerns the programmer’s perspective, codes must be written so to make the pipelines saturation as effective as possible.

```
for (int i = 0; i < N; i++)
  S += a[i] * b[i];
```

This way S is both read and written (the S as input in iteration *i* depends on S as output of iteration *i−1*) and the FP pipeline is difficult to be exploited (iteration *i* must unavoidably wait for iteration *i−1* to end, which takes ~3-6 cycles at least)

Pipeline optimization

Make things easier for the compiler and the CPU

```
for (i = 0; i < N; i += 2) {
    sum0 += a[i] * b[i];
    sum1 += a[i+1] * b[i+1]; }
```

Unrolling make it easier for the CPU to saturate the pipelines.

```
for (i = 0; i < N; i += 2)   {
    double tmp0 = a[i] * b[i];
    double tmp1 = a[i+1] * b[i+1];
    sum0 += tmp0;
    sum1 += tmp1; }
```

Separate load and multiply from addition

Pipeline optimization

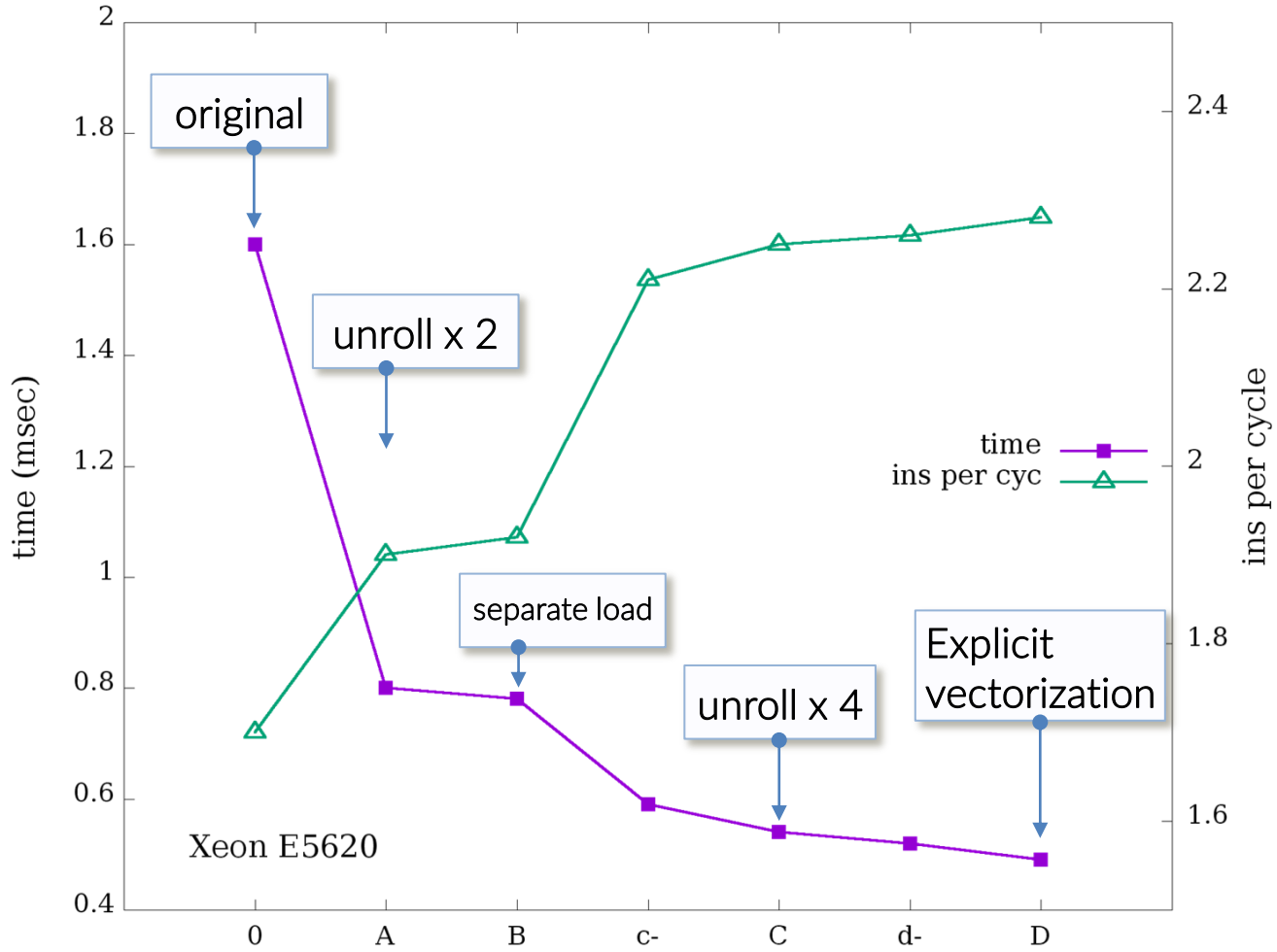
Make things easier for the compiler and the CPU

```
for (i = 0; i < N; i += 4)   {
     sum0 += a[i] * b[i];
     sum1 += a[i+1] * b[i+1];
     sum2 += a[i+2] * b[i+2];
     sum3 += a[i+3] * b[i+3]; }
```

More unrolling may work even better

```
typedef double v4df __attribute__ ((vector_size (4*sizeof(double))));
v4df array1, array2;
v4df sum;
for (i = 0; i < N/4; i ++)
   sum += array1[i] * array2[i];
```

Explicit vectorization


original
unroll x 2
separate load
unroll x 4
Explicit vectorization
Xeon E5620
time
ins per cyc
time (msec)
ins per cycle
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
2.4
2.2
2
1.8
1.6
0
A
B
c-
C
d-
D

Branch prediction

Conditional branches should be avoided as much as possible inside loops:

- moving them outside the loop and writing specialized loops
- performing variables/quantities set-up pree-emptively outside the loop
- using pointers to functions instead of selecting functions inside the loop
- substituting conditional branches with different operations

Branch prediction

ex 1: Taking decisions before and outside the loop

```
for(i = 1; i < top; i++)
   {
     if(case1 == 0) {
       if(case2 == 0) {
          if(case3 == 0)
            result += i;
          else
            result -= i;
          }
          else {
            if(case3 == 0)
              result *= i;
            else
              result /= i;
          }
      }
```

```
else {
  if(case2 == 0) {
     if(case3 == 0)
       result += log10((double)i);
     else
       result -= log10((double)i);
     }
  else {
    if(case3 == 0)
      result *= sin((double)i);
    else
      result /= (sin((double)i) +
                 cos((double)i));
     }
  }
```

Branch prediction

If you do not trust your compiler to perform the *loop hoisting* for you,

- define a specialized function for each case
- before and outside the loop set a function pointer to the right function

```
void (*func)(double *, int);
<here make func pointing to the right place>

double temp   = 0;
double result = 0;
for(i = 1; i < top; i++)
   {
        func( & temp, i);
        result = temp;
    }
```

Branch prediction

However:

- Using function pointers you may incur in additional overhead due to function call.
  If the code snippets in different if-branches (or at least the most executed ones) are large/expensive, it might well be pointless (in modern CPUs).

- "Unrolling" the if-tree outside the for – then having multiple for loops may be highly unpractical if the branches are big piece of code.
  There's no really a Swiss-knife recipe.

```
if (case1 == 0) {
  if (case2 == 0) {
     if (case3 == 0) {
        for(i = 1; i < top; i++)
           result += i;
     else
           for(i = 1; i < top; i++)
              result -= i;
     }
```

Branch prediction

ex 2: code restructuring

Consider the following code snippet

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

Branch prediction

Consider the following code snippet(*)

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

qsort(data, SIZE, sizeof(int), compare);

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

(*) of course, you are adding an overhead due to the sorting routine, so the total running time may be even larger. Moreover, you should have all the values available so that does not work for real-time streamings. However, the point here is to focus on how – in general – it is better to avoid conditionals inside loop, with any possible trick or change in workflow

Branch prediction

You can do even better, without adding operations:

```
// generate random numbers
for (cc = 0; cc < SIZE; cc++)
  data[cc] = rand() % TOP;

// take action depending on their value
for (ii = 0; ii < SIZE; ii++)
  {
     t = (data[ii] - PIVOT -1) >> 31;
     sum += ~t & data[ii];
  }
```

Branch prediction

-O0

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred

sum is 983597794767, elapsed seconds: 5.40445

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow

sum is 983597794767, elapsed seconds: 2.23186
(in total: 2.44473 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart

sum is 983597794767, elapsed seconds: 2.8878

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

-O3

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03

sum is 983597794767, elapsed seconds: 0.660148

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03

sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03

sum is 983597794767, elapsed seconds: 0.679286

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

Branch prediction

-O3

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03

sum is 983597794767, elapsed seconds: 0.660148

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03

sum is 983597794767, elapsed seconds: 0.650005
(in total: 0.795181 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03

sum is 983597794767, elapsed seconds: 0.679286

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2%
```

-O3
-march=native

```
luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.03n

sum is 983597794767, elapsed seconds: 0.217864

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.wow.03n

sum is 983597794767, elapsed seconds: 0.215645
(in total: 0.355377 seconds)

luca@GGG:~/code/HPC_LECTURES/branch_prediction/branch_prediction_2% ./branchpred.smart.03n

sum is 983597794767, elapsed seconds: 0.224288
```

Branch prediction

What changes in the base version with –O3 ? → conditional move

Modern CPUs have the capability of performing *conditional move*, i.e to execute **concurrently** both branches of a conditional – **if they are "simple enough"** – and to select the right result upon the evaluation of the conditional

```
perform op1 → res in AX
perform op2 → res in BX

compare

if flag     → mov BX in AX
```

HOWEVER: loops with conditionals **can not be fully vectorized !!**

Branch prediction

Why the difference in the base v. between **-O3** and **-O3 -march=native** ?

```
.L8:
        movdqu  xmm0, XMMWORD PTR [rax]
        movdqu  xmm6, XMMWORD PTR [rax]
        movdqa  xmm2, xmm4
        add     rax, 16
        pcmpgtd xmm0, xmm5
        pand    xmm0, xmm6
        pcmpgtd xmm2, xmm0
        movdqa  xmm3, xmm0
        punpckldq       xmm3, xmm2
        punpckhdq       xmm0, xmm2
        paddq   xmm1, xmm3
        paddq   xmm1, xmm0
        cmp     rax, rcx
        jne     .L8
```

compare **4 integers** at a time using xmm*X* registers, that are common to x86_64 architectures.

bytes reshuffling to add one `int` at a time. These are SSE 128-bits instructions

increase the counter by 4 **`int`**

12 instructions to process 4 **`int`**

```
.L8:
        vmovdqu         ymm2, YMMWORD PTR [rax]
        add             rax, 32
        vpcmpgtd        ymm0, ymm2, ymm3
        vpand           ymm0, ymm0, ymm2
        vpmovsxdq       ymm2, xmm0
        vextracti128    xmm0, ymm0, 0x1
        vpaddq          ymm1, ymm2, ymm1
        vpmovsxdq       ymm0, xmm0
        vpaddq          ymm1, ymm0, ymm1
        cmp             rax, rcx
        jne             .L8
```

bytes reshuffling inside regs to add one `int` at a time
The v prefix tells you these are AVX2 256-bits instr.

compare **8 integers** at a time using ymm*X* registers. This requires AVX2 that is set on by `-march=native` for this CPU

increase the counter by 8 **`int`**

9 instructions to process 8 **`int`**

Branch prediction

We can do slightly better:

```
for (ii = 0; ii < SIZE; ii++)
  {
    if (data[ii] > PIVOT)
       sum += data[ii];
  }
```

can be changed to

```
for (ii = 0; ii < SIZE; ii++)
  {
     acc = ( data[ii]>PIVOT )? data[ii] : 0;
     sum += acc;
  }
```

Branch prediction

Let's look to another practical example

You have 2 arrays, A and B, and you want to swap their elements so that

$$A[i] \geq B[i]$$

for all *i*.
A straightforward implementation would be:

```
for (i = 0; i < SIZE; i++)
  {
        if ( A[i] < B[i] )
           {
                t = B[i];
                B[i] = A[i];
                A[i] = t; }
           }
```

Branch prediction

However, that implementation suffers exactly of the same problem we have just discussed.
An alternative way to write the same code, but in a more effective style is:

```
for (i = 0; i < SIZE; i++)
  {
     int min = A[i] > B[i] ? B[i] : A[i];
     int max = A[i] >= B[i] ? A[i] : B[i];

      A[i] = max;
      B[i] = min;
   }
```

Branch prediction

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    if ( B[ii] > A[ii] )
      {
        int t = A[ii];
        A[ii] = B[ii];
        B[ii] = t;
      }
```

standard

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int register t = -(A[ii]<B[ii]);
    int register x = A[ii]^B[ii];
    A[ii] = A[ii]^(x & t);
    B[ii] = B[ii]^(x & t);
```

smart2

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int max = (A[ii]>B[ii])? A[ii]:B[ii];
    int min = (A[ii]>B[ii])? B[ii]:A[ii];
    A[ii] = max;
    B[ii] = min;
```

smart

```
for (uint ii = 0; ii < SIZE; ii++)
  {
    int d = A[ii]-B[ii];
    d &= (d >> 31);
    A[ii] = A[ii] - d;
    B[ii] = B[ii] + d;
```

smart3

standard, -O0

predictable data has a regular pattern easily spotted by the CPU's branch predictor

```
1000000000 predictable numbers generation..  done

seconds           :    2.223 +- 0.004446
CPE               :          9.938 +- 0.0138
IPE               :    27.0001758 +- 3.036e-07
BRANCHES_PE       :    2.00000014 +- 6.585e-10
MISP_BRANCHES_PE  :     4.892e-07 +- 2.134e-08  ( tot:          244.6 )


1000000000 random numbers generation..  done

seconds           :    4.501 +- 0.02864
CPE               :         27.4 +-    0.2
IPE               :    26.9996208 +- 2.98e-08
BRANCHES_PE       :    2.00000014 +- 6.585e-10
MISP_BRANCHES_PE  :    0.499933228 +- 2.215e-05  ( tot:     249966614 )
```

loop run-time

cycles per element

Instructions per element

~2.7 IPC

~1 IPC

conditional branches pert element

mis-predicted cond. branches per el.

total # of mis-predicted conditional branches

same number of IPE but almost 3 times as many CPE when data pattern is not predictable

in fact, there is 1 mis-predicted branch every 2 (arrays are 500,000,000 long)

Branch prediction

```
seconds           :    1.824 +- 0.05734
CPE               :         11.12 +- 0.341
IPE               :     34.0000018 +- 4.344e-08
BRANCHES_PE       :     1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :      9.94e-07 +- 3.207e-08   ( tot:          497 )
```

~3 IPC

predictable

number of IPE is larger than for standard case, but the CPE is stable !

mis-predicted branches are very few and comparable in both cases

```
seconds           :    1.857 +- 0.001762
CPE               :         11.32 +- 0.00192
IPE               :     34.0000018 +- 2.581e-08
BRANCHES_PE       :     1.00000014 +- 3.293e-10
MISP_BRANCHES_PE  :      9.512e-07 +- 4.987e-08   ( tot:        475.6 )
```

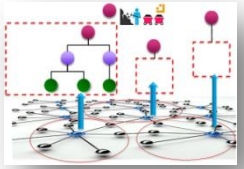
~3 IPC

random

A word of caution

It may be really easy to get lost in “optmization”, in hunting every single line wondering why some incredible trick that – you’re convinced – should work, actually does not.

“Optimization” includes also optimizing your effort and your time, so always **remember that the far most important ingredients** are:



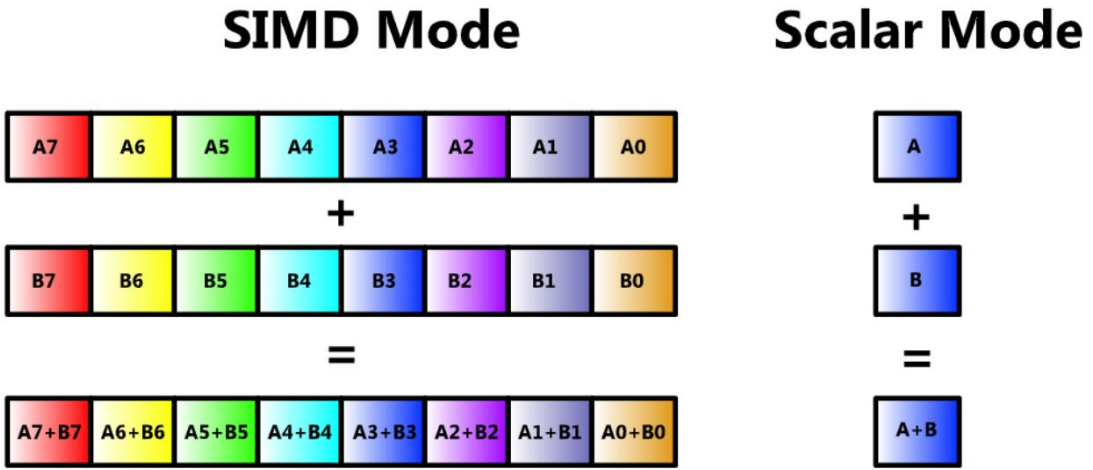
The algorithms that you choose

The data model you design

The overall quality, cleanness and robustness of your code

Late 90s: vector capabilities become mainstream

https://software.intel.com/sites/default/files/m/d/4/1/d/8/Intro_to_Intel_AVX.pdf



Vector registers are large special registers in the CPU that can be considered subdivided in smaller independent chunks over which the same operation can be performed:

SIMD: Single Instruction
Multiple Data

Vector registers size

SSE Data Types (16 XMM Registers)

__m128	Float | Float | Float | Float	4x 32-bit float
__m128d	Double | Double	2x 64-bit double
__m128i	B | B | B | B | B | B | B | B | B | B | B | B | B | B | B | B	16x 8-bit byte
__m128i	short | short | short | short | short | short | short | short	8x 16-bit short
__m128i	int | int | int | int	4x 32bit integer
__m128i	long long | long long	2x 64bit long
__m128i	doublequadword	1x 128-bit quad

AVX Data Types (16 YMM Registers)

__mm256	Float | Float | Float | Float | Float | Float | Float | Float	8x 32-bit float
__mm256d	Double | Double | Double | Double	4x 64-bit double
__mm256i	*256-bit Integer registers. It behaves similarly to __m128i.Out of scope in AVX, useful on AVX2*	

Early 00s: multi-core becomes mainstream

Late 00s : accelerators become mainstream
Early 10s: many-cores + heterogeneous computation become mainstream

That is the focus of the next two days…

At the end.. first things first

"Optimization" may be a tag for several different concepts, as we have seen at the beginning of this course.

Many concurrent facts and factors must be kept together, and it is quite difficult to give general statements about which ones are more fundamental.

Top-level design plays a key role, as well as algorithm choice and implementation details.
Sometime – not at all that often! – even a single line of code, for instance a prefetching, may have brilliant consequences

…

At the end.. first things first

However, unless you are seeking some extreme performance, as a general guideline just keep in mind that “optimization” reads

“let the compiler squeeze the maximum from your code”

Compilers are quite good indeed, and know the hardware they are running on better than you (99% of times).

So, as first, just learn how to :

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

At the end.. first things first

- write non-obfuscated code
  → avoid memory aliasing
  → make it clear what a variable is used for and when

- design a good data structure layout
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

At the end.. first things first

- write non-obfuscated code
- design a good data structure layout
  → be cache-conscious
  → be NUMA-conscious
  → avoid race conditions in multi-threaded codes
- design a “good” workflow
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

At the end.. first things first

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow
  → compiler will be able to optimize branches and memory access patterns
  → prefetching will work better
  → make it easier to use multi-threading
- take advantage of the modern out-of-order, super-scalar, multi-core architectures

At the end.. first things first

- write non-obfuscated code
- design a good data structure layout
- design a “good” workflow

- take advantage of the modern out-of-order, super-scalar, multi-core architectures
  → let the compiler exploit pipelining through operation ordering and unloop
  → let the compiler exploit the vectorization capabilities of CPUs
  → think task-based, data-driven

C-specific hints

Storage class

May load/store latency of heavily used variables

- `extern`
  Global variables, they exist forever
- `auto`
  Local variables, allocated on the stack for a limited scope, and then destroyed. They must be initialized
- `register`
  Suggests that the compiler puts this variable directly in a CPU register

C-specific hints

Variable qualifiers

- `const`
  Global variables, they exist forever
- `volatile`
  Indicates that this variable can be accessed from outside the program.
- `restrict`
  A memory address is accessed only via the specified pointer

C-specific hints

:: note about **restrict** qualifier ::

```
void my_function( double *a, double *b, int n)
{
        for( int i = ; i < n; i++ )
                a[ i ] = s * b[ i - 1 ];
}
```

The compiler can not optimize the access to **a** and **b** because it can not assume that **a** and **b** are pointing to the same memory locations.
That is called *aliasing*, formally forbidden in fortran: which is the reason why in some cases fortran may compile in faster executables.

Help your C compiler in doing the best effort, either writing a clean code or using restrict or using **-fstrict-aliasing -Wstrict-aliasing** options.

Compiler's usage

Language standard

Some constructs can be better implemented if you specify a specific standard to the compiler.

Architecture specification

The compiler is able to detect the architecture it runs on, but – to maximize executable's compatibility – it will not turn on very specific optimization unless you tell him so.

Read *carefully* the manual of your compiler.
In **`gcc`**, for instance, **`-march=native`** turns on a lot of sophisticated optimizations

Compiler's usage

Optimization level: -O*n*

It is not granted that `-O3`, although often generating a faster code, is what you really need.
For instance, sometimes expensive optimizations may generate more code that on some architecture (e.g. with smaller Icaches) run slower, and using `-Os` may bring surprising results.

Take into accounts that modern compilers allows for local specific optimizations or compilation flags. In gcc for instance:

```
__attribute__ ((__option__ ("…")))
__attribute__ ((optimize(n)))
```

Compiler's usage

Profile-guided optimization

Compilers (**`gcc`** , **`icc`** and **`clang`**) are able to instrument the code so to generate run-time information to be used in a subsequent compilations.
Knowing the typical execution patterns enables the compiler to perform more focused optimizations, especially if several branches are present.

For **`gcc`**:

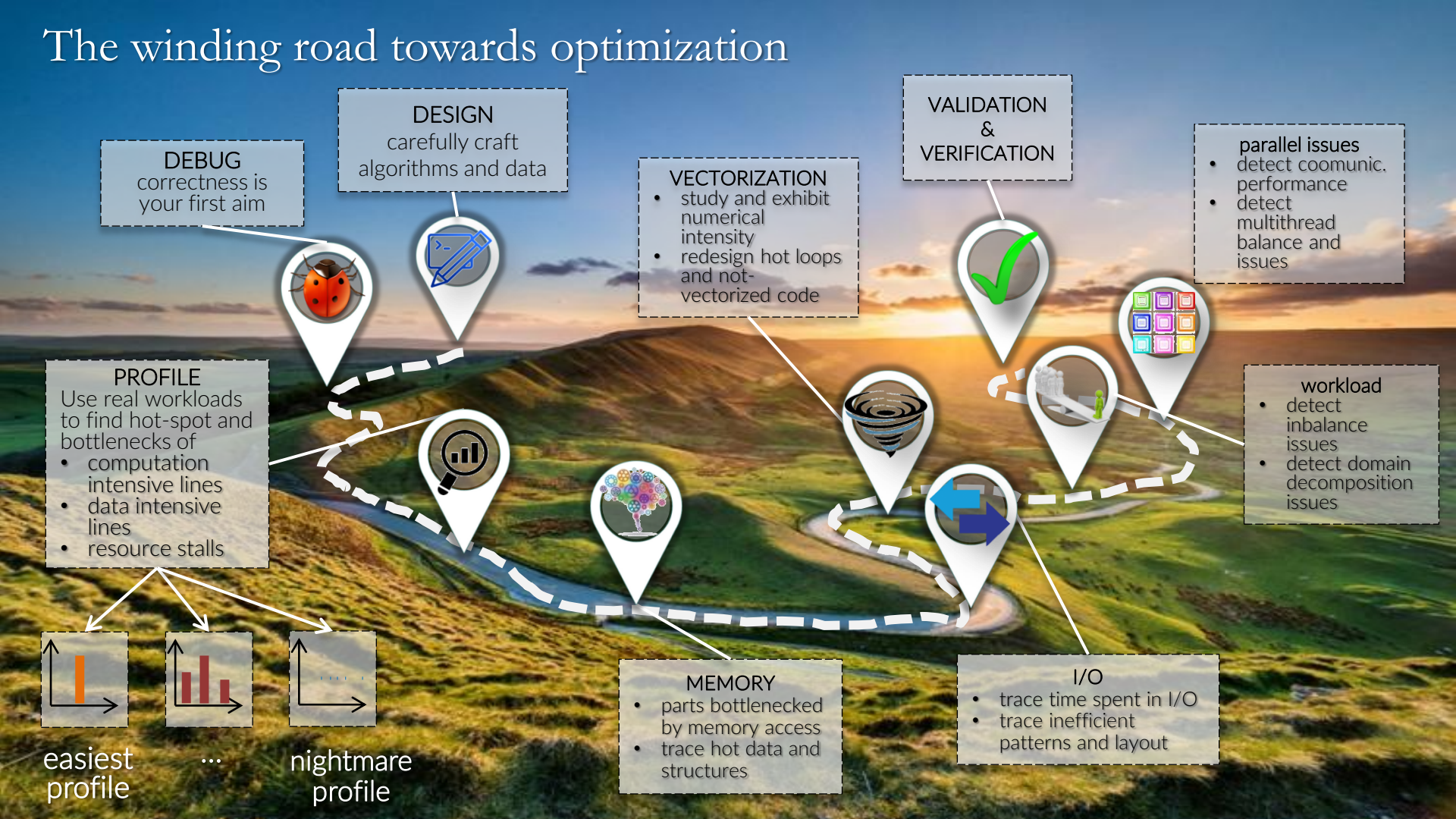
Optimize branch prediction

```
gcc -fprofile-arcs
< … run … >
gcc -fbranch-probabilities
```

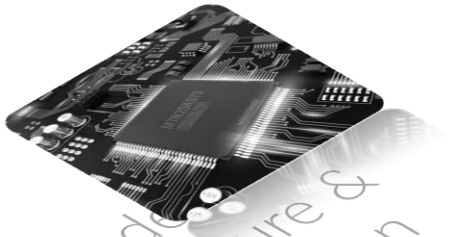
More general profiling

```
gcc -fprofile-generate
< … run … >
gcc -fprofile-use
```

The winding road towards optimization



Outline



Profiling

Debugging

Map the workflow

A call tree (more precisely, a call graph) is a control flow graph that exhibits the calling relationships among routines in a program.
A *node* in the graph represent a routine, while an *edge* represents a calling relationship.

We'll concentrate on ***dynamic*** call graphs – i.e. the records of program executions.

The most complete graph is context-sensitive, which means that every call stack of a procedure is recorded as a separate node (the resulting graph is called *calling context tree* instead of *call tree*).

However, that requires a larger amount of memory for large program, it is useful in case of code reuse (the same code being executed at different points by differentcall paths).

Basic profiling concepts

INSTRUMENTATION

Inserts extra code at compile time wrapping function calls to count how many times it calls / is called and how much time it takes to execute.

SAMPLING

The profiler ask for interrupts $N_{samples}$ per sec + interrupts at function calls + interrupts at selected events, and records on a histogram the number of occurrences in every part of the program. The call graph is inferred from these data.

DEBUGGING

The profiler ask for interrupts at every line code and function call (more correct: enters the by-step execution mode)

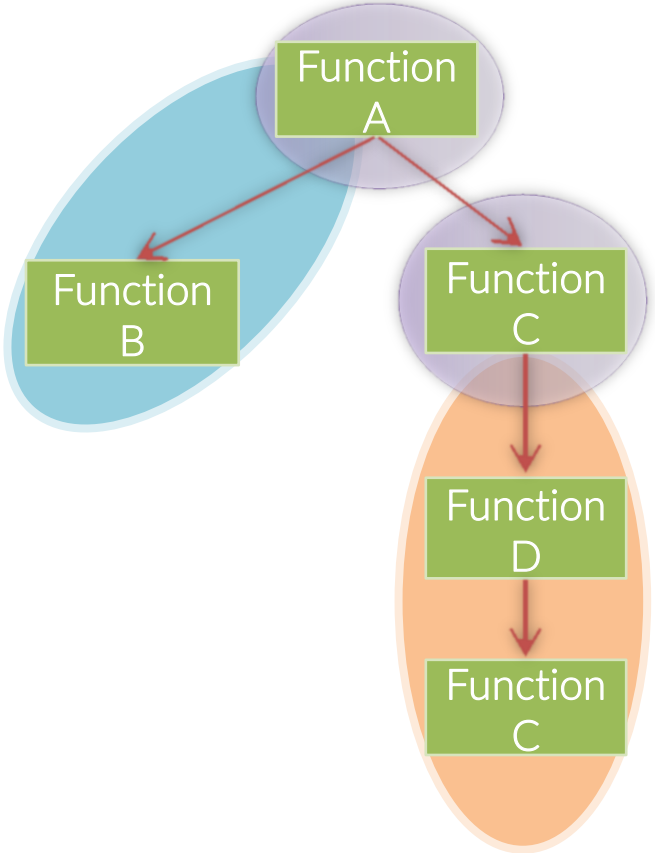
Basic profiling concepts

Collect **program events**

- Hardware interrupts
- Code instrumentation
- Instruction set simulation
- tracing

Periodic sampling

- Top of the stack (exclusive)
- All stack (children inclusive)



How to obtain the call tree

There are several way to obtain a dynamic call tree.
The main open source alternatives are:

1. Using gcc and **gprof**
2. Using linux **google perftools**
3. Using **valgrind**
4. Using **perf**
5. Some others, mostly non-free (among significant free: CodeXL by AMD

Many IDE uses the aforementioned tools or their own plug-ins (eclipse, netbeans, code::block, codelite, …)

The call-tree

Using `gcc + gprof`

Just compile your source using `-pg` option.
You should also profile turning on the optimizations you're interested in.

```
gcc -[my.optimizations] -pg -o myprogram.x myprogram.c
```

Note: don't use the option -p. It provides less information than -pg.

After that, run your program normally. Profiling infos will be written in the file `gmon.out`.

- You can read (options and details in the man page) the informations by

```
gprof myprogram.x
```

- You can visualize the call graph by (read the man pages…)

```
gprof myprogram.x | gprof2dot.py | dot -T png -o callgraph.png
```

The call-tree

Well, now that you just met, say goodbye to the glorious **`gprof`**

He's a dinosaur from the past decades..

- lacks real multithread support
- lacks real line-by-line capability
- does not profile shared libraries
- need recompilation
- may lie easier than other tools (see later..)

You may still consider it for some call counts business

How to induce you profiler to lie

`gprof` (**`gcc -pg`**) does not record the call stack

```
#include <stdlib.h>
void loop(int n)
{
  int volatile i; // does not optimize out
  i = 0;
  while(i++ < n);
}

void light(int n) { loop(n); }void heavy(int n) {
loop(n); }

int main(void)
{
  light(100000);
  heavy(100000000);
return 0;
}
```

How to induce you profiler to lie

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
101.30      0.20     0.20        2   101.30   101.30  loop
  0.00      0.20     0.00        1     0.00   101.30  heavy
  0.00      0.20     0.00        1     0.00   101.30  light
```

How to induce you profiler to lie

How could it happen?
The man page is clear:

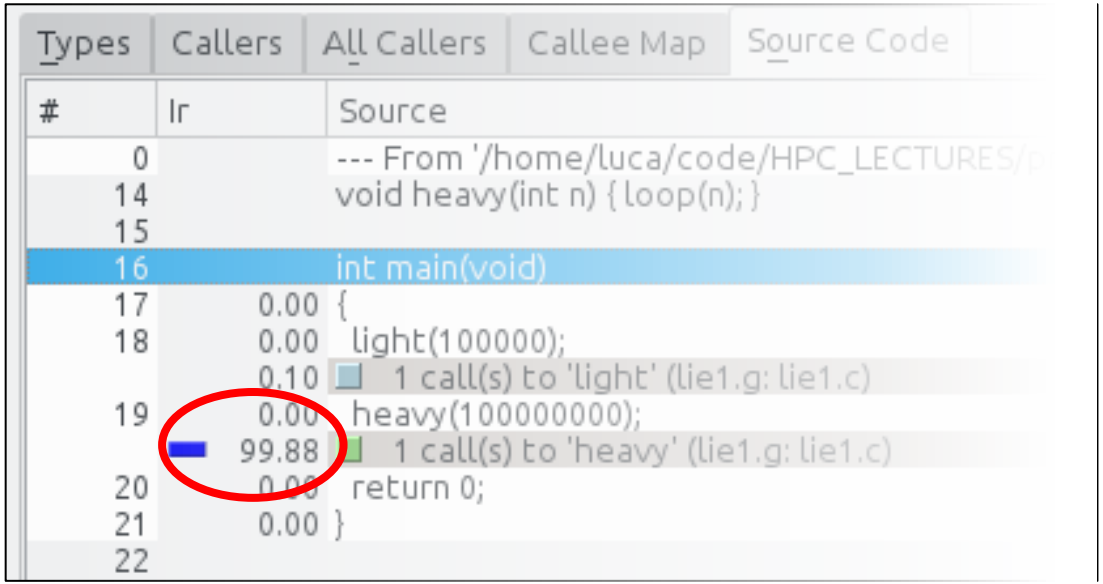
We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called.

Thus the time propagated along the call graph arcs to the function's parents is directly proportional to the number of times that arc is traversed.

How to induce you profiler to lie

VALGRIND seems to better understand the situation, because it doesn't record only the number of calls to a function but also the time spent in a function given a call path (which, at odds, gprof infers backwardly).
That's way the valgrind measure of time spent in a function plus its callees is reliable.

However, it may as well end up to a misleading picture if we stick in and additional layer of complexity, *unless* you explicit tells it to track separately different call stacks with the command-line option `--separate-callers = N`



The call-tree

Using google perftools

That is the CPU profiler used at Google's. It provides a thread-caching malloc, a heap-checker, heap-profiler, CPU profiler.

As for the latter, basically there are 3 phases:

1. linking the library to the executable
2. running the executable
3. analyzing results

[1] LINKING

There are two options:

- Link `-lprofiler` at the executable
- Adding the profiler at run time `LD_PRELOAD="/usr/lib/libprofiler.so" /path/to/exec`

This does not start the CPU profiling, though.

The call-tree

Using google perftools

[2] RUNNING

- Define

```
env CPUPROFILE= exec.prof /path/to/exec
```

  You may define a signal, too

```
env CPUPROFILE= exec.prof /path/to/exec \
CPUPROFILESIGNAL= XX exec.prof &
```

  so that to be able to trigger the start and stop of the profiling by

```
killall –XX exec
```

- In the code, include **<gperftools/profiler.h>** and encompass the code segment to be profiled within

```
ProfilerStart("name_of_profile_file")
…
ProfilerStop()
```

CPUPROFILE_FREQUENCY=x modifies the sampling frequency

The call-tree

Using google perftools

[3] ANALYZING RESULTS

`pprof exec exec.prof`	"interactive mode"
`pprof --text exec exec.prof`	output one line per procedure
`pprof --gv exec exec.prof`	annotated call graph via 'gv'
`pprof --gv --focus = some_func …`	restrict to code paths including "*some_func*"
`pprof --list = some_func …`	per-line annotated list of some_func
`pprof --disasm= some_func …`	annotated disassembly of some_func
`pprof –callgrind exec exec.prof`	output call infos in callgrind format

The call-tree

Using perf

[1] COMPILING

```
gcc -g –fno-omit-frame-pointer my_prog.c –o my_prog
```

[2] RUNNING

```
perf record –F ffff –call-graph <fp|lbr|dwarf> \
my_prog.c <args>
```

[3] ANALYZING

```
perf report --call-graph=graph < --stdio >
```

The call-tree

Using valgrind

[1] COMPILING

```
gcc -g –fno-omit-frame-pointer my_prog.c –o my_prog
```

[2] RUNNING

```
Valgrind –tool=callgrind –callgrind-out-file= $CALLGRIND_OUT –dump-
instr=yes –coolect-jumps=yes –cache-sim=yes –branch-sim=yes < --I1=… >
< --D1=…>  my_prog <args>
```

[3] ANALYZING

```
kcachegrind $CALLGRIND_OUT
```

Tools

- BASIC / SYSTEM tools
  - gprof / gdb / perf / gperftools
  - Valgrind – cachegrind, callgrind, ..
- HARDWARE COUNTER / PMU interface
  - perf
  - PAPI
  - Intel PMI
  - Likwid
  - …
- HPC Tools
  - HPCtoolkit
  - OpenlSpeedShop
  - TAU
  - SCOREP +
- VENDOR tools
  - ARM-Allinea
  - CodeXL (AMD)
  - Intel tools
  - …

Tools : valgrind

An instrumentation framework for building dynamic analysis tools.
Valgrind basically runs your code in a virtual “sandobx” where a synthetic CPU (the same you have) is simulated and executes an instrumented code.

There are various Valgrind based tools for debugging and profiling purposes.

- Memcheck is a memory error detector → correctness
- Cachegrind is a cache and branch-prediction profiler → velocity
- Callgrind is a call-graph generating cache profiler. It has some overlap with Cachegrind
- Helgrind is a thread error detector → correctness
- DRD is also a thread error detector.
  Different analysis technique than Helgrind
- Massif is a heap profiler → memory efficiency using less memory
- DHAT is a different kind of heap profiler → memory layout inefficiencies
- SGcheck (experimental tool) that can detect overruns of stack and global arrays

KCacheGrind is a very useful GUI

Tools : valgrind

Memcheck : highlighting memory errors

- Unvalid memory access: overrunning/underrunning of heap blocks or top of stack, addressing freed blocks, …
- Use of variables with undefined values
- Incorrect freeing of heap memory
- Errors in moving memory (unwanted src/dst overlaps, …)
- Memory leaks

Cachegrind: simulating the cache

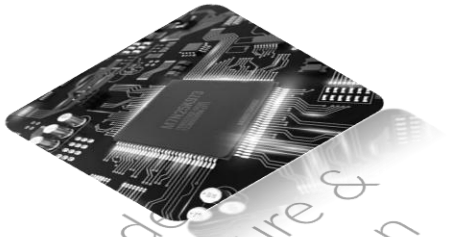
It can report how many hits (L1, L2 and L3, I- and D-) and how many misses.
It can analyze CPU's branch prediction.

```
Ir, I1mr, LLmr, Dr, D1mr, DLmr, Bc, Bcm, Bi, Bim, …
```

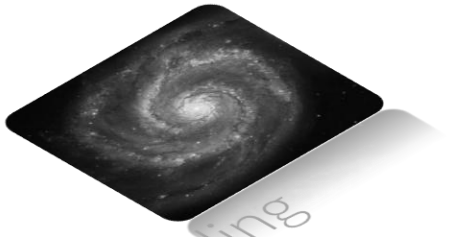
Callgrind: profiling the CPU

It collects the number of instructions executed, links them to source lines, records the caller/callee relationship between functions, and the numbers of such calls. It can collect data on cache simulation and/or branches.

Outline



Model Architecture & Optimization



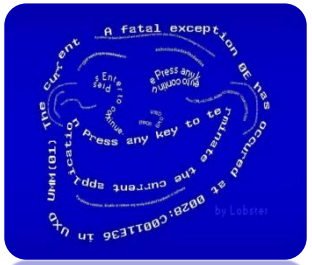
Profiling

Debugging

Debugging outline

Introduction

Debugging a code



Inspecting a code crash

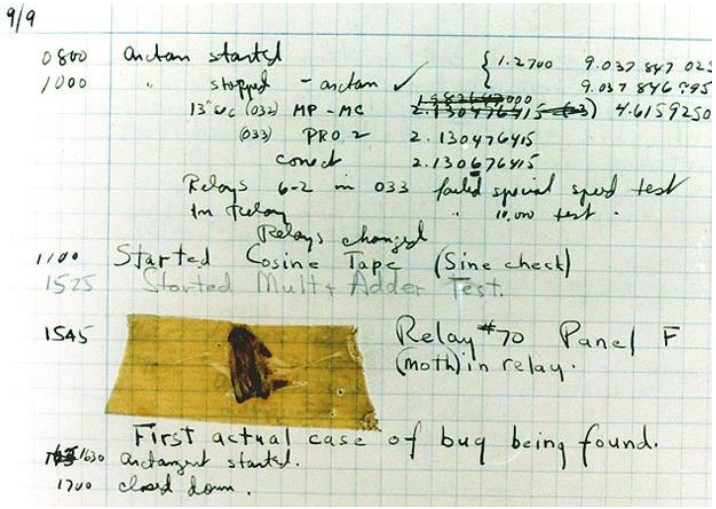
Debugging a running process

Introducing debugging

Smithsonian

It has long been recognized and documented that insects are the most diverse group of organisms, meaning that the numbers of species of insects are more than any other group. In the world, some 900 thousand different kinds of living insects are known. This representation approximates 80 percent of the world's species. The true figure of living species of insects can only be estimated from present and past studies. Most authorities agree that there are more insect species that have not been described (named by science) than there are insect species that have been previously named. Conservative estimates suggest that this figure is 2 million, but estimates extend to 30 million. In the last decade, much attention has been given to the entomofauna that exists in the canopies of tropical forests of the world. From studies conducted by Terry Erwin of the Smithsonian Institution's Department of Entomology in Latin American forest canopies, the number of living species of insects has been estimated to be 30 million. Insects also probably have the largest biomass of the terrestrial animals. At any time, it is estimated that there are some 10 quintillion (10,000,000,000,000,000,000) individual insects alive.

Introducing debugging



The usage of ***bug*** and, hence, of ***de-bugging*** referred to programming is a long-standing tradition, whose origin is difficult to trace back.

An often-told story is about Mark-II calculator located at Harvard: On Sept. 9th, 1945, a technician found a moth in a relay that caused a flaw in a “program” execution.

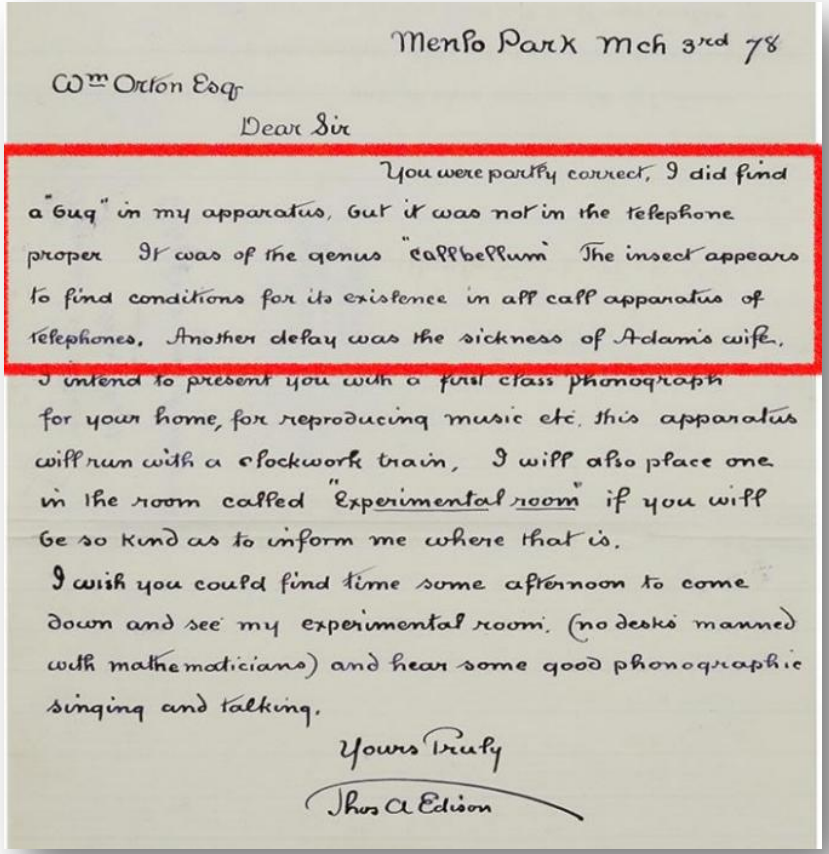
Adm. Grace Hopper is reported to have written in its diary about that as “first *actual* case of bug being found”

Introducing debugging

However, **Thomas Edison** found another *actual* bug in one of his phones, as he reports in a letter to an associate.

He later writes:

"It has been just so in all of my inventions. The first step is an intuition and comes with a burst, then difficulties arise—this thing gives out and [it is] then that "Bugs"—as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached."



Menlo Park Mch 3rd 78

Wm Orton Esq

Dear Sir

You were partly correct, I did find a "bug" in my apparatus, but it was not in the telephone proper It was of the genus "callbellum" The insect appears to find conditions for its existence in all call apparatus of telephones. Another delay was the sickness of Adams wife.

I intend to present you with a first class phonograph for your home, for reproducing music etc. this apparatus will run with a clockwork train, I will also place one in the room called "Experimental room" if you will be so kind as to inform me where that is.

I wish you could find time some afternoon to come down and see my experimental room, (no desks manned with mathematicians) and hear some good phonographic singing and talking.

Yours Truly

Thos A Edison

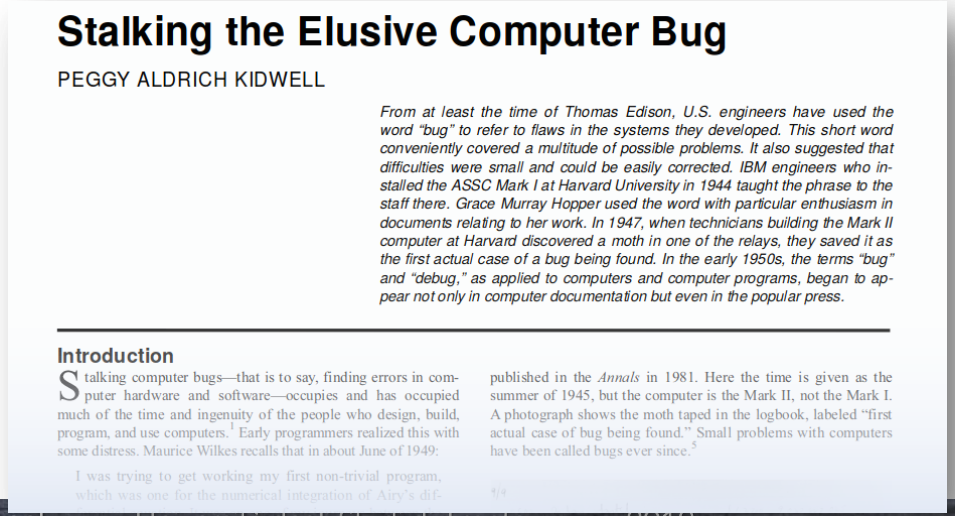
Introducing debugging

It is difficult to trace back how this term has been poured into computer programming jargon.
However, already in the early 60s it was appearing in technical papers without need of explanation.

The immortal I. Asimov used it in a 1944 short robot story “Catch that rabbit”, and his incredible influence contributed much to make the term popular.

More funny infos:
IEEE Annals of the History of Computing
(Volume: 20 , Issue: 4 , Oct-Dec 1998)

ask for the PDF if you’re interested and do not have access



Stalking the Elusive Computer Bug

PEGGY ALDRICH KIDWELL

From at least the time of Thomas Edison, U.S. engineers have used the word “bug” to refer to flaws in the systems they developed. This short word conveniently covered a multitude of possible problems. It also suggested that difficulties were small and could be easily corrected. IBM engineers who installed the ASSC Mark I at Harvard University in 1944 taught the phrase to the staff there. Grace Murray Hopper used the word with particular enthusiasm in documents relating to her work. In 1947, when technicians building the Mark II computer at Harvard discovered a moth in one of the relays, they saved it as the first actual case of a bug being found. In the early 1950s, the terms “bug” and “debug,” as applied to computers and computer programs, began to appear not only in computer documentation but even in the popular press.

Introduction

Stalking computer bugs—that is to say, finding errors in computer hardware and software—occupies and has occupied much of the time and ingenuity of the people who design, build, program, and use computers.[1] Early programmers realized this with some distress. Maurice Wilkes recalls that in about June of 1949:

I was trying to get working my first non-trivial program, which was one for the numerical integration of Airy’s dif-

published in the *Annals* in 1981. Here the time is given as the summer of 1945, but the computer is the Mark II, not the Mark I. A photograph shows the moth taped in the logbook, labeled “first actual case of bug being found.” Small problems with computers have been called bugs ever since.[5]

Introducing debugging

That maybe is too dramatic and emphatic, but the point to get from Edison is that the *de-bugging activity* is an inherent and intrinsic one in software development

*"It has been just so in all of my inventions. The **first step** is an intuition and comes with a burst, **then difficulties arise**—this thing gives out and [it is] then that "**Bugs**"—as such little faults and difficulties are called—show themselves and **months of intense watching, study and labor are requisite** before commercial success or failure is certainly reached."*

T. Edison

Introducing debugging

Provided that

- you'll have close encounters with bugs in your life;
- de-bugging is a fundamental and unavoidable part of your work;

what is the best way to proceed ?

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works

2. Add **`printf`** statements everywhere
   Highly discouraged, but sometimes works.
   In case of memory problems – tipically due to pointers chaos – you may see that the problem "disappear", or changes its appearance, when you insert a new `printf`

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works
2. Add `printf` statements everywhere
   Highly discouraged, but sometimes works.
   In case of memory problems – tipically due toaos
3. Use a DEBUGGER
   That is definitely the best choice and, fortunately, the subject of this lecture.

Introducing debugging

1. Do not insert bugs
   Highly encouraged, but rarely works

gdb is almost certainly the best free, extremely feature-rich command-line debugger ubiquitously available on *niX systems.

3. Use a DEBUGGER
   That is definitely the best choice and, fortunately, the subject of this lecture.

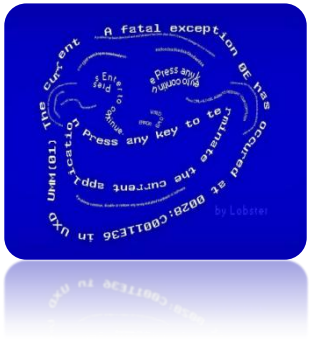
There are 3 basic usages(*) of GDB:

1. Debugging a code
   best if it has been compiled with –g
2. Inspecting a code crash through a core file
3. Debugging / inspecting a running code

(*) Higly advised: learn keyboard commands. Although many gui exist (we'll see some later), keyboard is still the best productivity tool.

Debugging outline

Debugging a code



Inspecting a code crash

Debugging a running process

Compiling with dbg infos

In order to include debugging information in your code, you need to compile it with `-g` family options (read the gcc manual for complete info):

`-g`	produce dbg info in O.S. native format
`-ggdb`	produce gdb specific extended info, as much as possible
**`-g`*`level`*	default level is 2. 0 amounts to no info, 1 is minimal, 3 includes extra information (for instance, macros expansion) – this allows macro expansion; add **`-gdwarf-`*`n`*** in case, where possibile, where *`n`* is the maximum allowed (4)
**`-ggdb`*`level`*	you can combine the two to maximize the amount of useful info generated remember: **`-fno-omit-frame-pointer`**, especially if you are using –O*x*

Debugging: start

You just start your code under `gdb` control:

```
%> gdb program
```

You can define the arguments needed by your program already at invocation:

▶ live demo with **`gdb_try_args.c`**

```
%> gdb --args program arg1 arg2 … argN
```

Or you can define the arguments from within the gdb session:

```
%> gdb program
Reading symbols from program...done.
(gdb) set args arg1 arg2 … argN
(gdb) run
```

Debugging: run & stop

You may just want the code to run, for instance to reach the point of a seg fault:

```
%> (gdb) run
```

Or, you may want to stop it from the beginning to have full control of each step:

```
%> (gdb) break main
%> (gdb) run
```

Or you may already know what is the problematic point to stop at:

```
%> (gdb) break location
%> (gdb) run
```

Debugging: run & stop

Breakpoints are a key concept in debugging. They are stopping point at which the execution interrupts and the control is given back to you, so that you can inspect the memory contents (variables values, registers values, …) or follow the subsequent execution step by step.

You can define a breakpoint in several way

`(gdb) break`	insert a break at the current pos
`(gdb) break ±offset`	insert a break *offset* lines after/before the current line
`(gdb) break filename:linenum`	insert a break at *linenum* of file *filename*
`(gdb) break functionname`	insert a break at the entry point of function *functionname*

`There are more options, just check the manual`

Debugging: run & stop

A breakpoint may be defined as dependent on a given condition:

```
(gdb) break my_function if (arg1 > 3 )
```

This sets a breakpoint at function *my_function* : the condition will be evaluated each time the point is reached, and the execution is stopped only if it is true.

Condition can be any valid expression.

```
(gdb) info break
```

gives you informations on active breakpoints.

```
(gdb) delete [n]
      clear [location]
      <disable | enable>  see the manual
```

Debugging: run & stop

You can define a list of commands to be executed when a given breakpoint is reached:

```
(gdb) break my_function if (arg1 > 3 )
Breakpoint 1 at 0x……: file blabla.c, line 42
(gdb) command 1
Type commands for when breakpoint 1 is hit, one per
line.
End with a line saying just "end".
> print arg1
> print another_useful_variable
> x/10wd a_global_integer_array
```

Debugging: run & stop

When you have the control of the program execution, you can decide how to proceed:

`(gdb) cont [c]`	continue until the end / next stop
`(gdb) cont` *`count-ignore`*	continue ignoring the next *count-ignore* stops (for instance, occurrences of a bp)
`(gdb) next [n] | count` `nexti`	continue to the next src line *in the current stack frame*
`(gdb) step [s] | count` `stepi`	continue to the next src line
`(gdb) until [u] | count`	continue until a src line past the current one is reached in the current stack fr.
`(gdb) advance` *`location`*	continue until the specified location is reached

▶ live demo with **`gdb_try_breaks.c`**

Debugging: run & stop

You can also rewind your execution step-by-step

```
(gdb) reverse-continue [rc]

(gdb) reverse-step [count]
      reverse-stepi

(gdb) reverse-next [count]
      reverse-nexti

(gdb) set exec-direction <verse | forward >
```

Debugging: source list

Often, when you are debugging, you may have the need of looking at either the source lines or at the generated assembler:

`(gdb) list` *`linenum`*	print src lines around line *linenum* of the current source file
`(gdb) list` *`function`*	print the source lines of *function*
`(gdb) list` *`location`*	print src lines around *location*
`(gdb) set listsize` *`count`*	control the number of src lines printed
`(gdb) disass [/m][`*`function`*`][`*`location`*`]`	show the assembler

▶ live demo with **`gdb_try_breaks.c`**

Debugging: stack examination

Examining the stack is often of vital importance. With GDB you can have a quick and detailed inspection of all the stack frames.

`(gdb) backtrace[`*`args`*`]`	print the backtrace of the whole stack
n	print only the *n* innermost frames
`-`*n*	print only the *n* outermost frames
full	print local variables value, also
`where, info stack`	additional aliases

▶ live demo with **gdb_try_breaks.c**

Debugging: memory examination

Accessing to the content of memory is a fundamental ability of a debugger. You have several different ways to do that:

`(gdb)print` *variable* `p/F` *variable*	print the value of *variable* print *variable* in a different format (`x,d,u,o,t,a,c,f,s`) *see then manual for advanced location*
`(gdb)x/FMT` *address*	Explore memory starting at address *address* -> see at live demo how to use this
`(gdb)display` *expr* `display/`*fmt expr* `display/`*fmt addr*	Add *expr* to the list of expressions to display each time your program stops

▶ live demo with **`gdb_try_breaks.c`**

Debugging: memory examination

Memory can be searched to find a particular value, of a given size

▶ live demo with **gdb_try_breaks.c**

```
(gdb) find [/sn] start, end, val1 [,val2, …]

      find [/sn] start, +len, val1 [,val2, …]
```

Search memory for a particular sequence of bytes.
s is the size of type to be searched
n is the max number of occurrences

Debugging: registers examination

Examining registers may be also useful (although it's something that only quite advanced users can conceive)

`(gdb) info registers`	print the value of all registers
`(gdb) info vector`	print the content of vector registers
`(gdb) print $rsp`	print value of the stack pointer
`(gdb) x/10wd $rsp`	print values of the first 10 4-bytes integers on the stack
`(gdb) x/10i $rip`	print the next 10 asm instructions

▶ live demo with **`gdb_try_breaks.c`**

Debugging: macro expansion

If there are macros in your code, they can be expanded, provided that you compiled the code with the appropriate option:

-g3 [gdb3][-dwarf-4]

`(gdb) macro expand` *macro*	shows the expansion of macro *macro*; *expression* can be any string of tokens
`(gdb) info macro [-a|-all]` *macro*	shows the current (or all) definition(s) of *macro*
`(gdb) info macros` *location*	shows all macro definitions effective at *location*

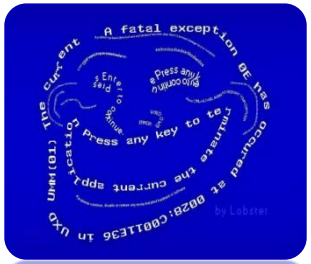
Debugging: watch points

You can set *watchpoints* (aka “keep an eye on this and that”) instead of breakpoints, to stop the execution whenever a value of an expression / variable / memory region changes

`(gdb) watch variable`	keep an eye onto *variable*
`(gdb) watch expression`	stops when the value of expression changes (the scope of variables is respected)
`(gdb) watch -l expression`	Interpret *expression* as a memory location to be watched
`(gdb) watch -l expression [mask maskvalue]`	a mask for memory watching: specifies what bits of an address should be ignored (to match more addressese)
`(gdb) rwatch [-l] expr [mask mvalue]`	stops when the value of *expr* is read

▶ live demo with **`gdb_try_watch.c`**

Debugging outline



Inspecting a code crash

Debugging a running process

After a crash : having the core file

It happens that you have code crashes in conditions not easily reproducible when you debug the code itself, for a number of reasons.

However, the O.S. can dump the entire “program status” on a file, called the *core file*:

```
luca@GGG:~/code/tricks% ./gdb_try_watch
no arguments were given, using default: 100

something wrong at point 2
[1]    8435 segmentation fault (core dumped)  ./gdb_try_watch
luca@GGG:~/code/tricks% ls -l core
-rw------- 1 luca luca 413696 nov  8 15:45 core
luca@GGG:~/code/tricks%
```

In order to allow it to dump the core, you have to check / set the core file size limit:

```
%> ulimit -c [size limit in KB]
```

After a crash : inspect the core file

Once you have a core, you can inspect it with GDB

```
%> gdb executable_name core_file_name
```

Or

```
%> gdb executable_name
  (gdb)core core_file_name
```

The first thing to do, normally, is to unwind the stack frame to understand where the program crashed:

```
  (gdb) bt full
```

Debugging outline

Introduction

Debugging a code

Inspecting a code crash

Debugging a running process

▶ live demo with **gdb_try_attach.c**

In order to debug a running process, you can simply attach gdb to it:

```
%> gdb
(gdb) attach process-id
```

and start searching it to understand what is going on

Debugging outline

GUI

for GDB

1. GDB text-user-interface
2. GDB DASHBOARD
3. GDBGUI
4. EMACS
5. DDD
6. NEMIVER, ECLIPSE, NETBEANS, CODEBLOCKS, many others..

GDB built-in tui

You can start gdb with a text-user-interface:

```
%> gdb -tui
```

Or you can activate/deactivate it from gdb itself:

`(gdb)ctrl-x a`	
`(gdb)ctrl-x o`	change focus
`(gdb)ctrl-x 2`	shows assembly windows
`(gdb)layout src`	display src and commands
`asm`	display assembly and commands
`split`	display src, asm and commands
`regs`	display registers window

GDB built-in tui

```
gdb_try_breaks.c
372     {
373
B+>374        if ( argc > 1)
375          // arg 0 is the name of the program itself
376          {
377            printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
378              for ( int i = 1; i < argc; i++ )
379                {
380                  printf( "\targument %d is : %s\n", i, *(argv+i) );
381                }
382              printf( "\n" );
383          }
384
385        else
386
387          printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );
388
389
390        int arg1;
391
392        if ( argc > 1 )
393          arg1 = atoi( *(argv+1) );
394
395        else
396          arg1 = DEFAULT_ARG1;
397
398        int ret;
399
400        ret = function_1( arg1 );
native process 8943 In: main                                    L374  PC: 0x555555554d38
(gdb) l
360     in /home/luca/code/tricks/gdb_try_breaks.c
(gdb) break main
Breakpoint 1 at 0xd38: file gdb_try_breaks.c, line 374.
(gdb) r
Starting program: /home/luca/code/tricks/gdb_try_breaks

Breakpoint 1, main (argc=1, argv=0x7fffffffdaa8) at gdb_try_breaks.c:374
(gdb)
```

GDB built-in tui

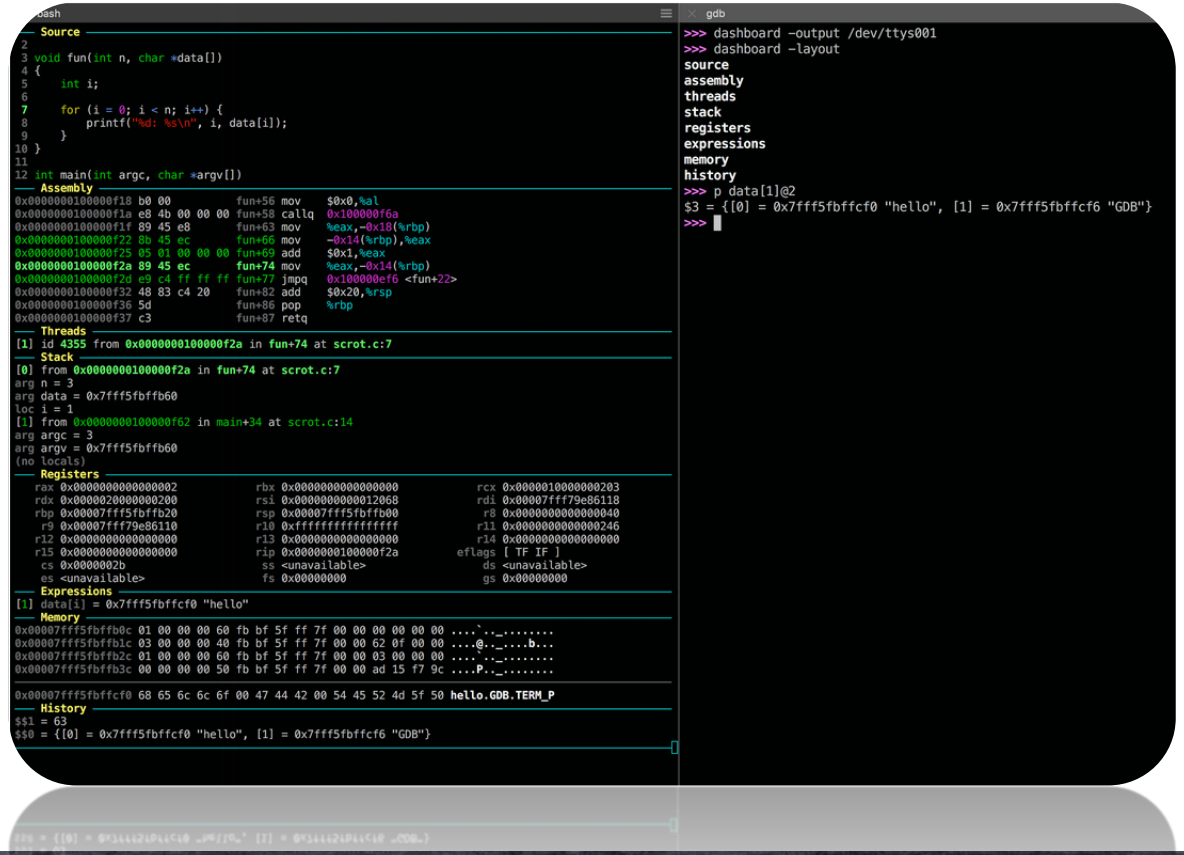
```
─gdb_try_breaks.c──
B+> 374       if ( argc > 1)
    375         // arg 0 is the name of the program itself
    376         {
    377           printf( "\nexploring my %d argument%c:\n", argc-1, (argc>2)?'s':' ' );
    378             for ( int i = 1; i < argc; i++ )
    379               {
    380                 printf( "\targument %d is : %s\n", i, *(argv+i) );
    381               }
    382             printf( "\n" );
    383         }
    384
    385       else
    386
    387         printf ( "no arguments were given, using default: %d\n\n", DEFAULT_ARG1 );
```

```
B+> 0x555555554d38 <main+15>        cmpl   $0x1,-0x14(%rbp)
    0x555555554d3c <main+19>        jle    0x555555554db7 <main+142>
    0x555555554d3e <main+21>        cmpl   $0x2,-0x14(%rbp)
    0x555555554d42 <main+25>        jle    0x555555554d4b <main+34>
    0x555555554d44 <main+27>        mov    $0x73,%edx
    0x555555554d49 <main+32>        jmp    0x555555554d50 <main+39>
    0x555555554d4b <main+34>        mov    $0x20,%edx
    0x555555554d50 <main+39>        mov    -0x14(%rbp),%eax
    0x555555554d53 <main+42>        sub    $0x1,%eax
    0x555555554d56 <main+45>        mov    %eax,%esi
    0x555555554d58 <main+47>        lea    0x2bf(%rip),%rdi        # 0x55555555501e
    0x555555554d5f <main+54>        mov    $0x0,%eax
    0x555555554d64 <main+59>        callq  0x555555554680 <printf@plt>
    0x555555554d69 <main+64>        movl   $0x1,-0xc(%rbp)
```

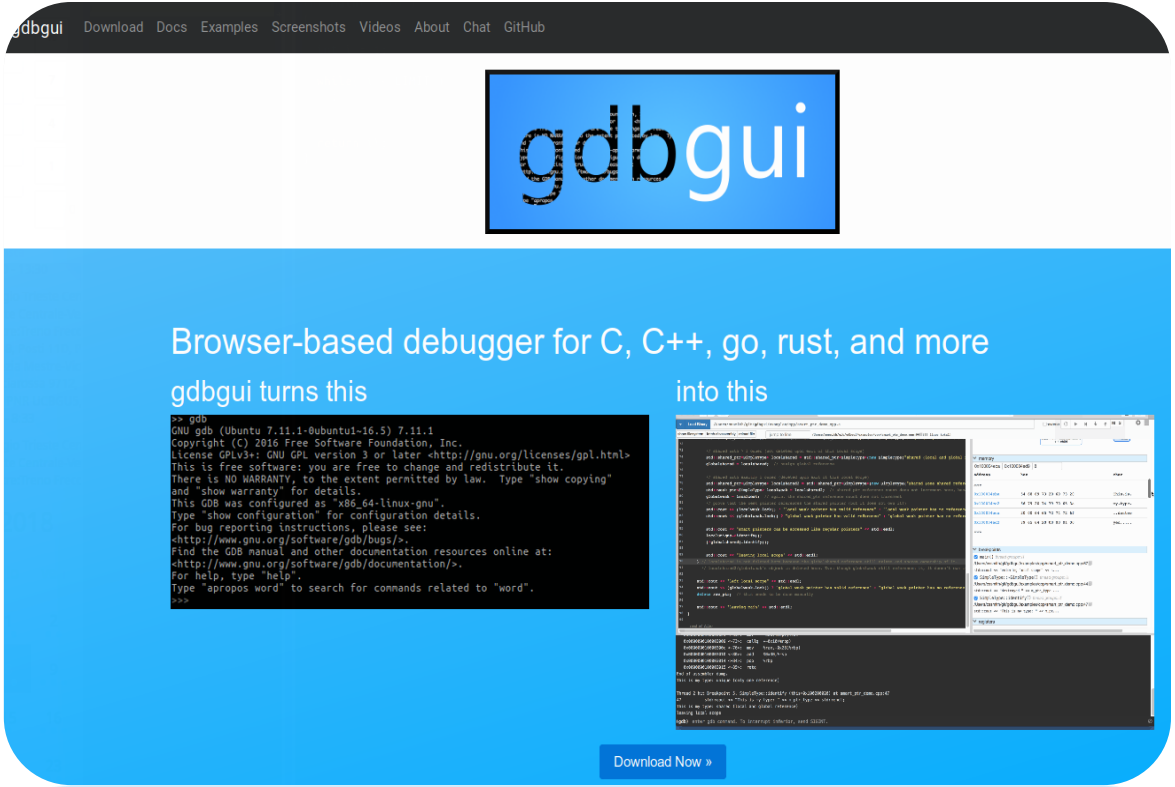
```
native process 9102 In: main                                    L374  PC: 0x555555554d38
(gdb) break main
Breakpoint 1 at 0xd38: file gdb_try_breaks.c, line 374.
(gdb) r
Starting program: /home/luca/code/tricks/gdb_try_breaks

Breakpoint 1, main (argc=1, argv=0x7fffffffdaa8) at gdb_try_breaks.c:374
(gdb) layout split
```

GDB dashboard

https://github.com/cyrus-and/gdb-dashboard



https://gdbgui.com/



that's all, have fun