Performance and Bottleneck Analysis

Sverre Jarp

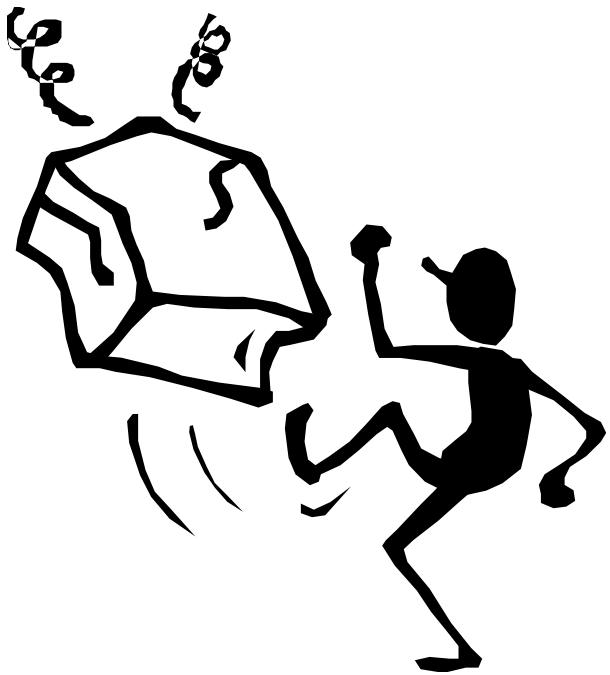
CTO of CERN openlab

Prepared for the HEPix Meeting, Rome

5 April 2006

Initial Question

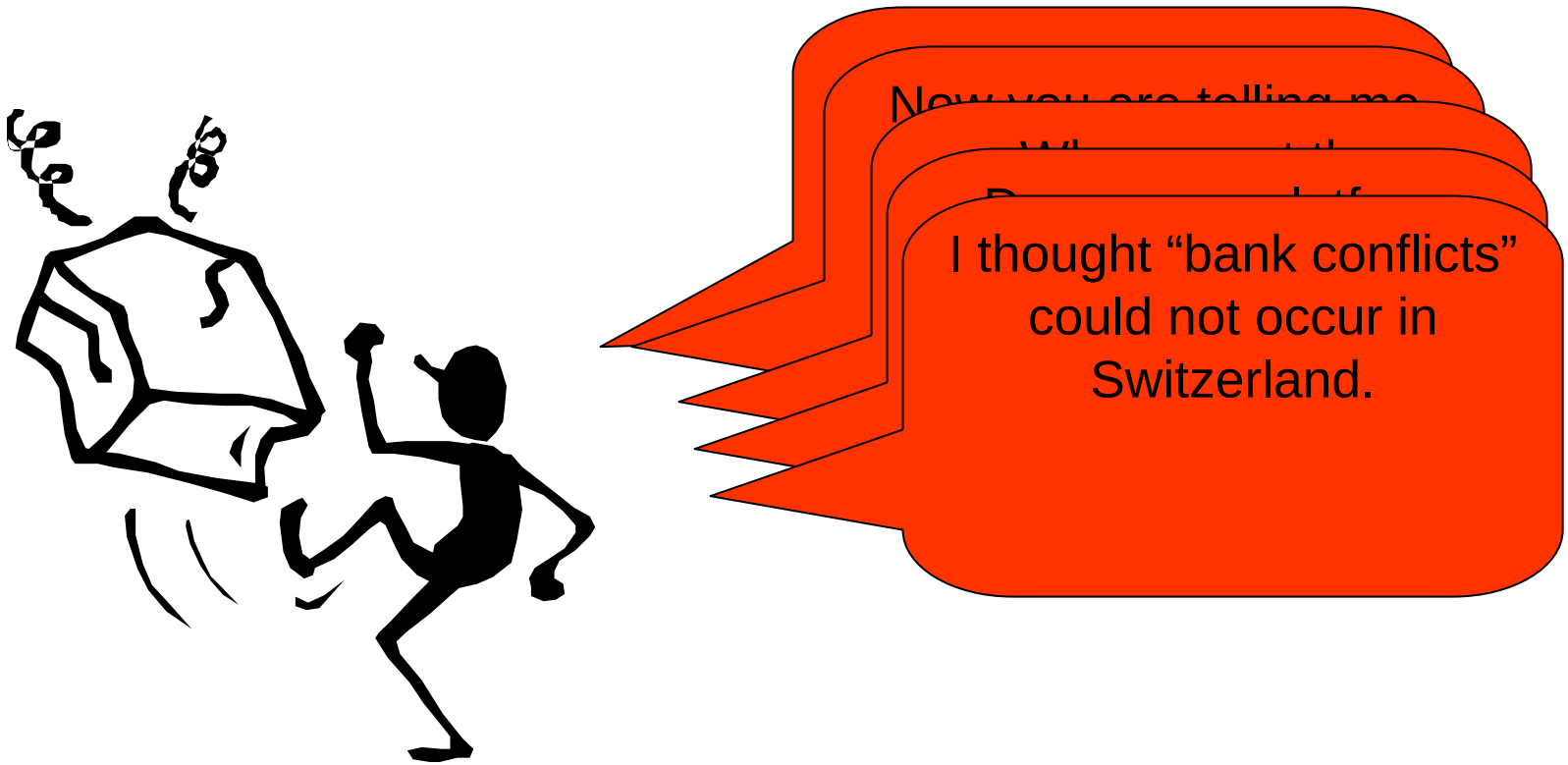
- **How come we (too) often end up in the following situation?**



Why the heck doesn't it perform as it should!!!

Initial Answer

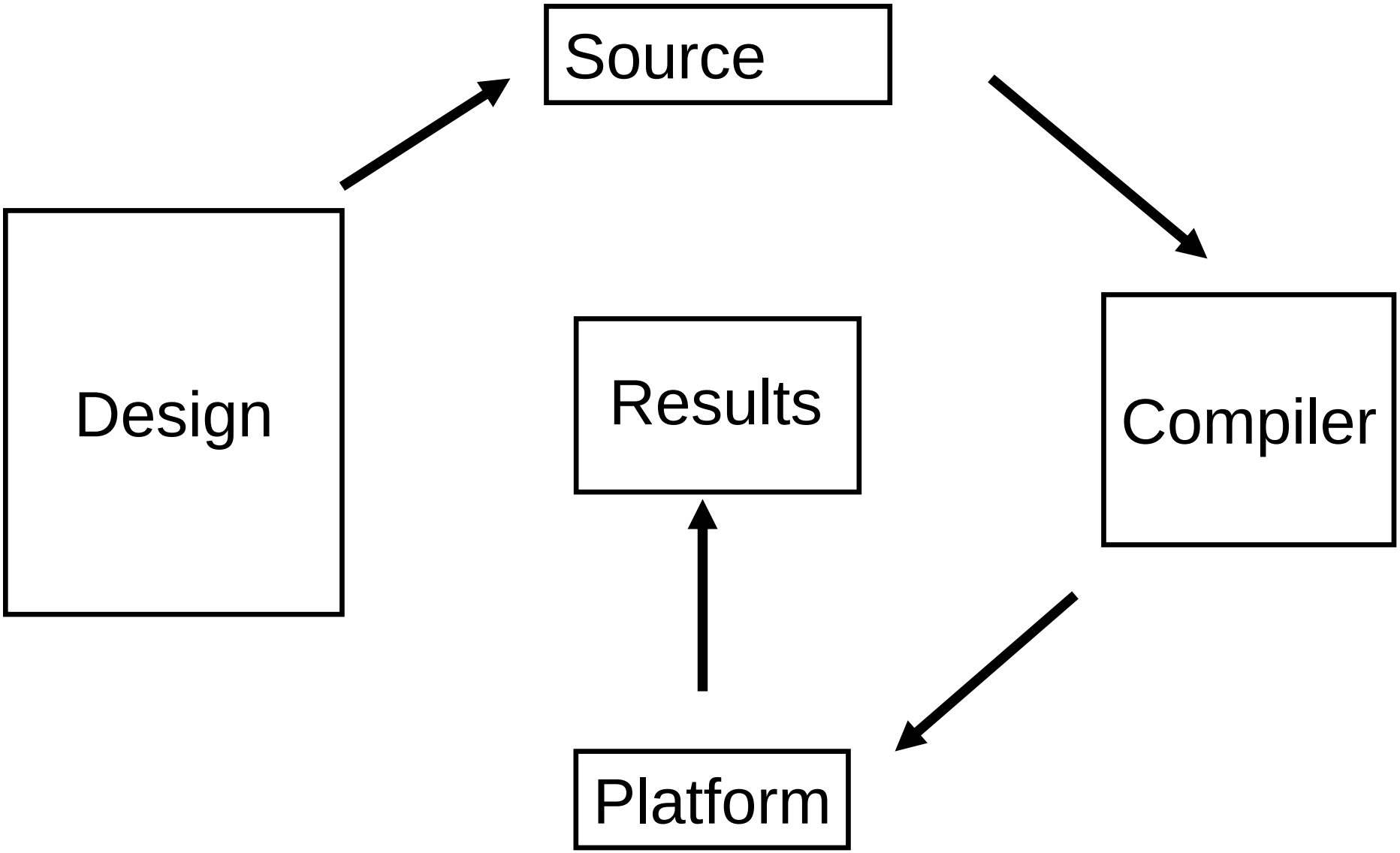
- **In my opinion, there are many (complicated) details to worry about!**



Before we start

- **This is an effort to teach a somewhat "systematic approach" to tuning and bottleneck analysis**
  - Main focus is on CPU bound processes
  - It is a methodology - not an "answer book"
- **The introduction of the elements is done "top-down"**
- **But, it is important to understand that in real-life, however, the approach is likely to be "middle-out"**
  - And often "reduced" middle-out

The Planned Path



First step: Analytical

Design of Data Structures and Algorithms

- **Choice of algorithms for solving our problems:**
  - Accuracy, Robustness, Rapidity
- **Choice of data layout**
  - Structure
  - Types
  - Dimensions
- **Design of classes**
  - Interrelationship
  - Hierarchy

Second step: Source

- **Choice of**
  - Implementation language
  - Language features &
  - Style
  - Precision (FLP)
  - Source structure and organization
  - Use of preprocessor
  - External dependencies
  - …
- **For example**
  - Fortran, C, C++, Java, ..
  - In C++: Abstract classes, templates, etc.
  - Single, double, double extended, ..
  - Contents of .cpp and .h
  - Aggregation or decomposition
  - Platform dependencies (such as endianness)
  - Smartheap, Math kernel libraries, …

Third step: Compiler

- **Choice of:**
  - Producer
  - Version
  - Flags
- **As well as:**
  - Build procedure
  - Library layout
  - Run-time environment
- **And (to a large extent):**
  - Machine code is then chosen for you.

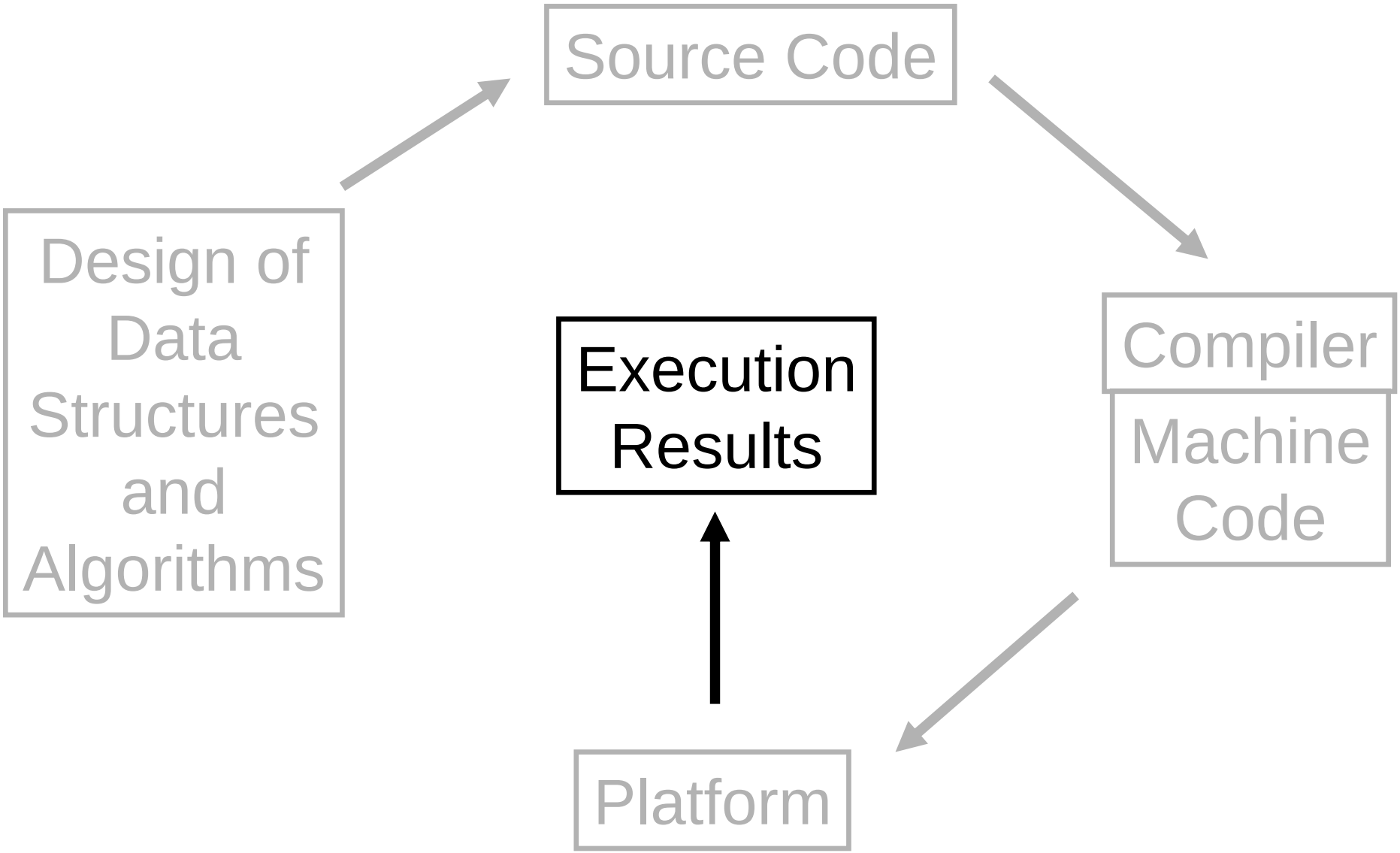
- **For example:**
  - GNU, Intel, Pathscale, Microsoft, ….
  - gcc 3 or gcc 4 ?
  - How to choose from hundreds?
  - Compiling one class at a time?
  - Archive or static?
  - Monolithic executable or dynamic loading?
  - Could influence via -mtune=xxx

By the way, who knows: -xK/W/N/P/B ?

Fourth step: Platform

- **Choice of:**
  - Manufacturer
  - ISA
  - Processor characteristics
    - Frequency
    - Core layout
    - Micro-architecture
    - Cache organization
  - Further configuration characteristics
- **Multiple options:**
  - AMD, Intel, Via, IBM, SUN, ..
  - IA32, AMD64/EM64T, IA64, Power, Cell, SPARC, ..
  - 3.8 GHz Netburst or 2 GHz “Core” ?
  - Single core, dual core, ..
  - Pentium 4, Pentium M, AMD K7, ..
  - Different sizes, two/three levels, ..
  - Bus bandwidth, Type/size of memory, ….

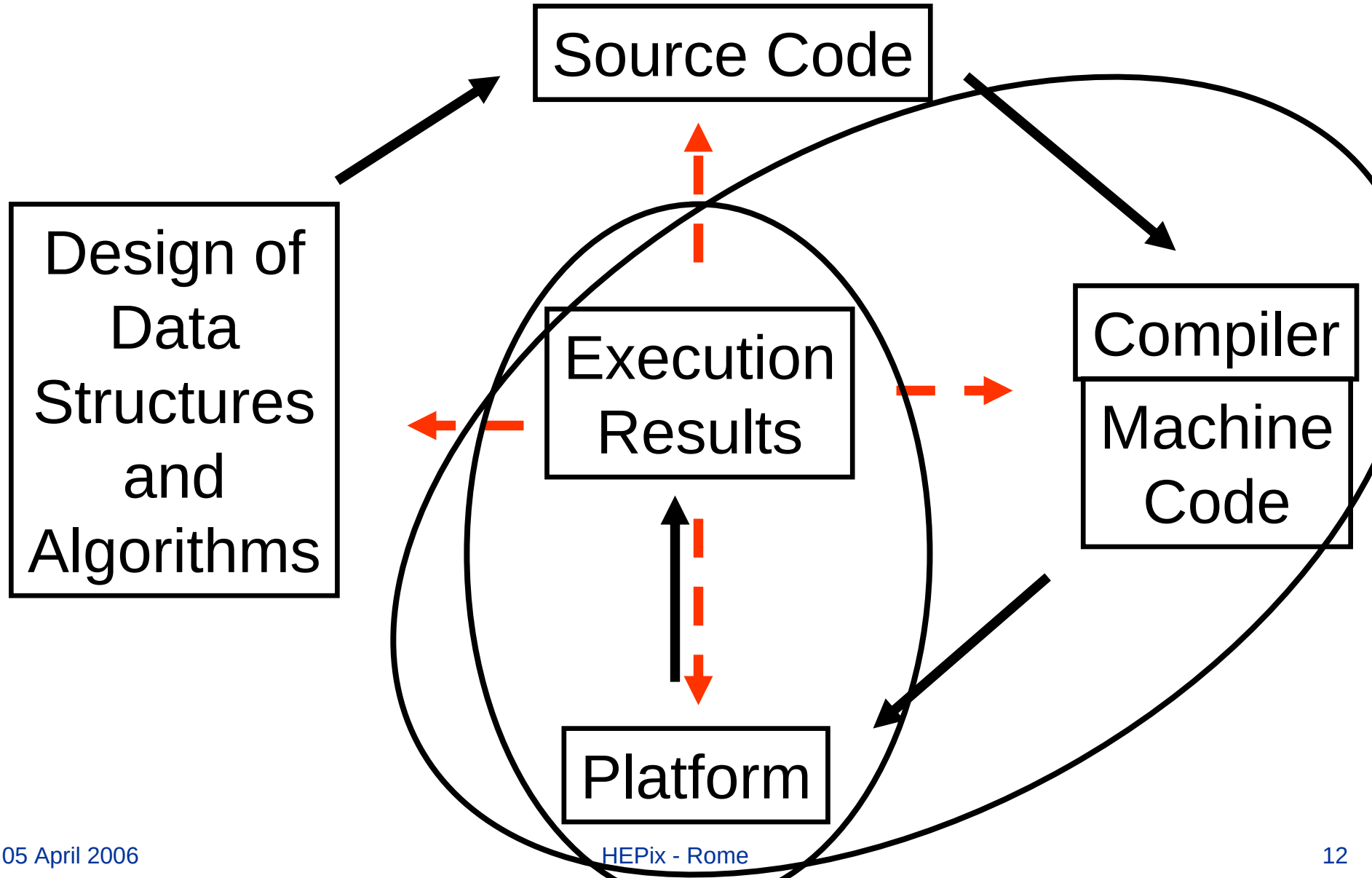
In the end: Execution Results



Back to our cartoon

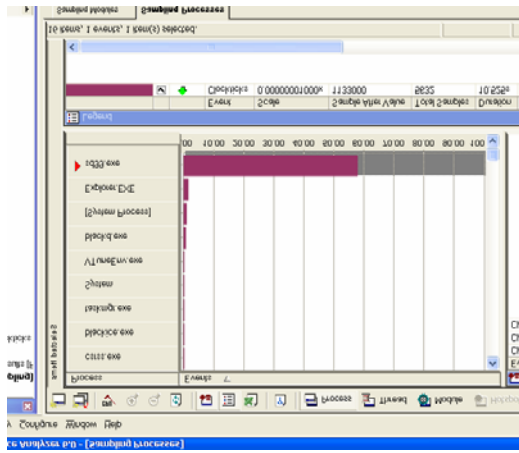
- **First of all, we must guarantee correctness**
- **If we are unhappy with the performance**
  - … and by the way, how do we know when to be happy?
- **We need to look around**
  - Since the culprit can be anywhere

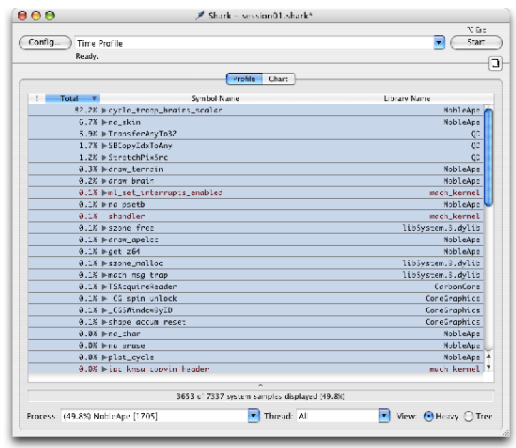
Where to look ?



Need a good tool set

- **My recommendation**
  - Integrated Development Environment (IDE) w/ integrated Performance Analyzer
    - Visual Studio + VTUNE (Windows)
    - Eclipse + VTUNE (Linux)
    - XCODE + Shark (MacOS)
    - .....
- **Also, other packages**
  - Valgrind (Linux x86, x86-64)
  - Qtools (IPF)
  - Pfmon, perfsuit, caliper, oprofile, TAU





Obstacles for HEP

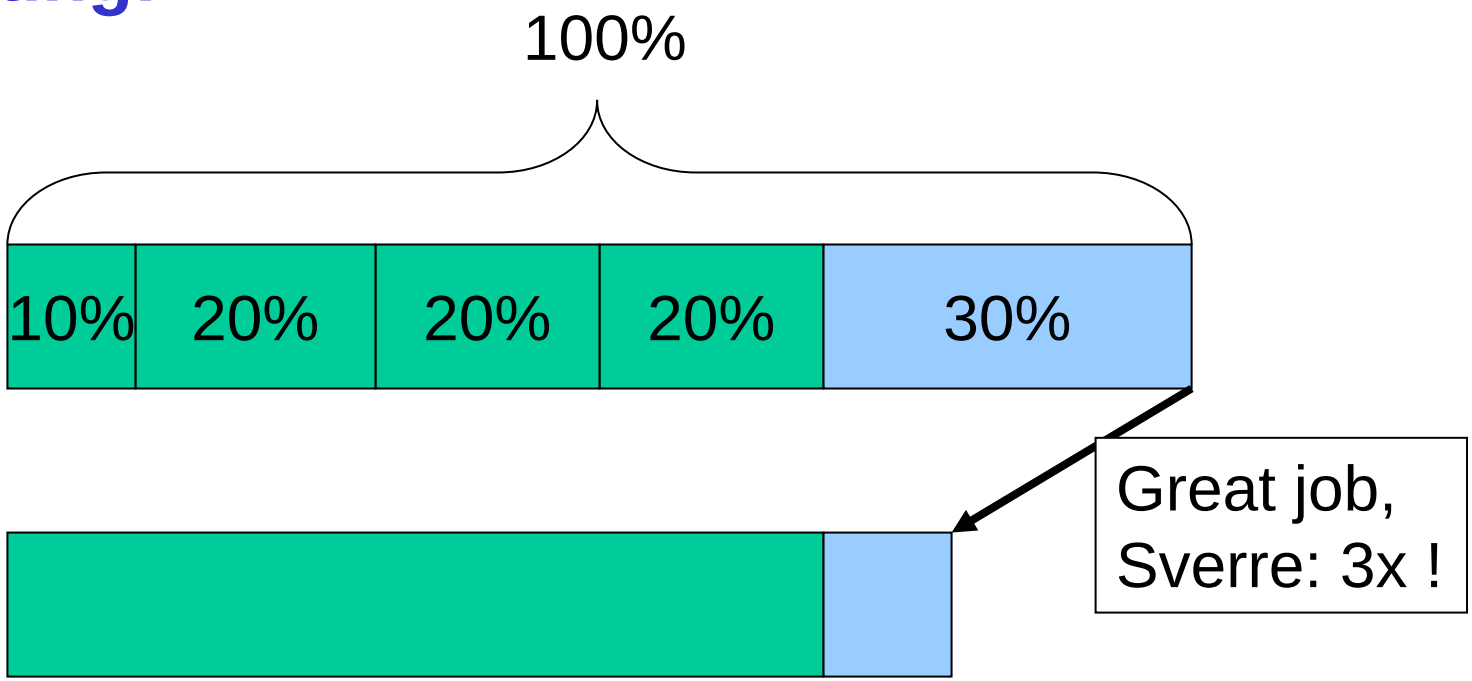
- **In my opinion, High Energy Physics codes present (at least) two obstacles**
  - Somewhat linked
- **One: Cycles are spread across many routines**
- **Two: Often hard to determine when algorithms are optimal**
  - In general
  - On a given h/w platform

Typical profile Geant4 (test40)

Samples	Self %	Total %	Function
280775	3.48%	3.48%	G4AffineTransform::InverseProduct(G4AffineT
262995	3.26%	6.74%	G4VoxelNavigation::LevelLocate(G4Navigation
250309	3.10%	9.84%	G4AffineTransform::G4AffineTransform(double
229238	2.84%	12.68%	??
198824	2.46%	15.15%	G4Tubs::DistanceToOut(Hep3Vector const&, Hep
183453	2.27%	17.42%	G4Transportation::AlongStepGetPhysicalInter
161668	2.00%	19.43%	G4SteppingManager::DefinePhysicalStepLength
158807	1.97%	21.40%	G4Tubs::CalculateExtent(EAxis, G4VoxelLimit
157553	1.95%	23.35%	G4Navigator::LocateGlobalPointAndSetup(Hep3V
147965	1.83%	25.18%	RanecuEngine::showStatus() const
138531	1.72%	26.90%	RanecuEngine::flat()
131271	1.63%	28.53%	G4SteppingManager::Stepping()
116957	1.45%	29.98%	G4ReplicaNavigation::ComputeStep(Hep3Vector
115858	1.44%	31.41%	__ieee754_log
110690	1.37%	32.78%	G4Navigator::ComputeStep(Hep3Vector const&,
108515	1.35%	34.13%	G4SteppingManager::InvokePSDIP(unsigned)
107472	1.33%	35.46%	G4Tubs::Inside(Hep3Vector const&) const
105221	1.30%	36.77%	pthread_mutex_lock
104726	1.30%	38.06%	G4SteppingManager::InvokeAlongStepDoItProcs
101621	1.26%	39.32%	G4Navigator::LocateGlobalPointAndUpdateTouch
99141	1.23%	40.55%	log

Amdahl's Law

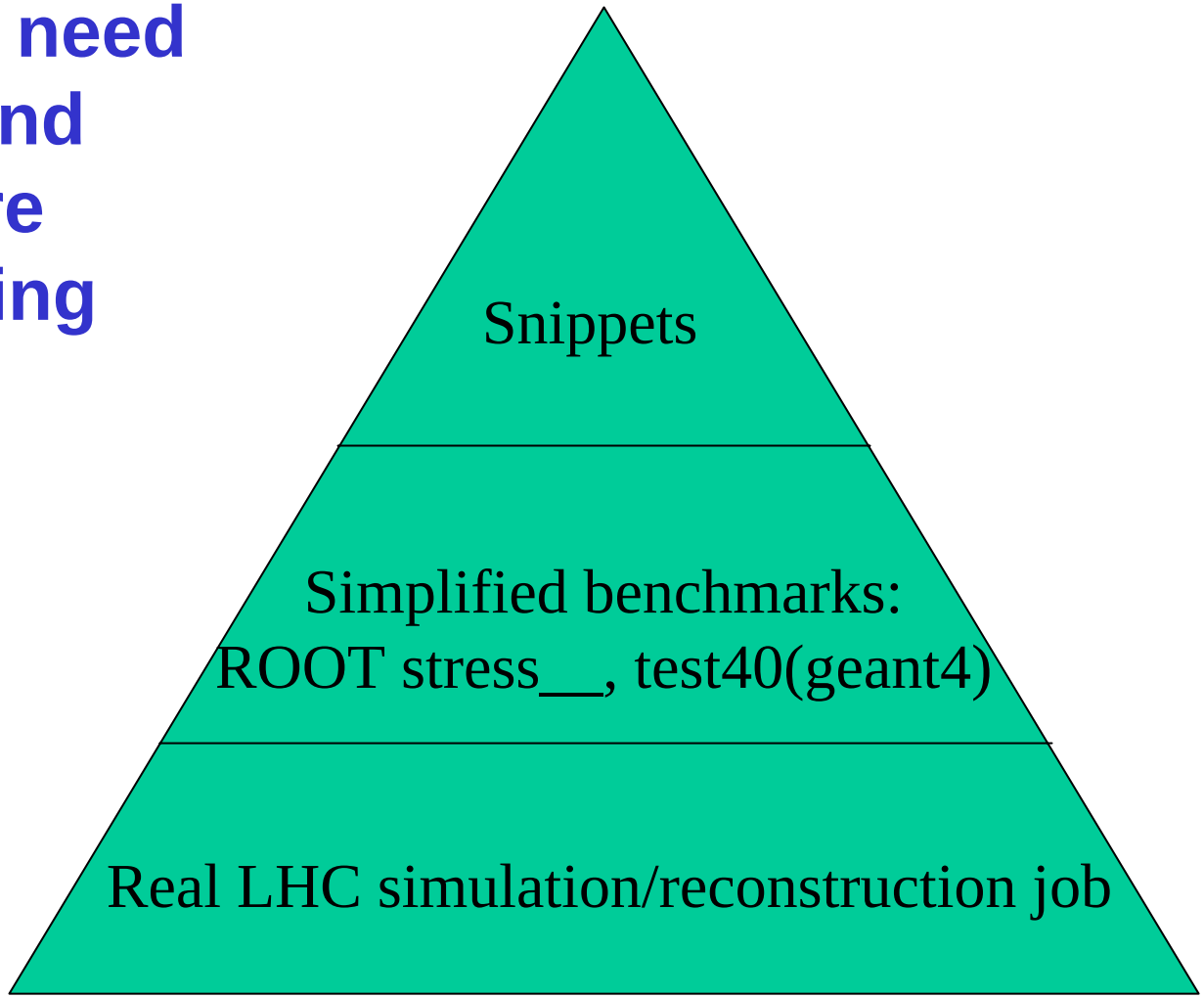
- **The incompressible part ends up dominating:**



Total speedup is "only": (100/80): 1.25

The testing pyramid

- **You always need to understand what you are benchmarking**



Mersenne Twister

```
Double_t TRandom3::Rndm(Int_t){
    UInt_t y;
    const Int_t  kM = 397; const Int_t  kN = 624; const UInt_t kTemperingMaskB =  0x9
    const UInt_t kTemperingMaskC =  0xefc60000; const UInt_t kUpperMask =       0x800
    const UInt_t kLowerMask =       0x7fffffff; const UInt_t kMatrixA =         0x990

    if (fCount624 >= kN) {
       register Int_t i;
      for (i=0; i < kN-kM; i++) {  /* THE LOOPS */
        y = (fMt[i] & kUpperMask) | (fMt[i+1] & kLowerMask);
        fMt[i] = fMt[i+kM] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
      }
      for (   ; i < kN-1    ; i++) {
        y = (fMt[i] & kUpperMask) | (fMt[i+1] & kLowerMask);
        fMt[i] = fMt[i+kM-kN] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
      }
      y = (fMt[kN-1] & kUpperMask) | (fMt[0] & kLowerMask);
      fMt[kN-1] = fMt[kM-1] ^ (y >> 1) ^ ((y & 0x1) ? kMatrixA : 0x0);
      fCount624 = 0;
    }
    y = fMt[fCount624++]; /*THE STRAIGHT-LINE PART*/
    y ^=  (y >> 11);  y ^= ((y << 7 ) & kTemperingMaskB );
    y ^= ((y << 15) & kTemperingMaskC );  y ^=  (y >> 18);
    if (y) return ( (Double_t) y * 2.3283064365386963e-10); // * Power(2,-32)
    return Rndm();
}
```

The “MT” loop is full

- **Highly optimized**
  - Here depicted in 3 Itanium cycles
    - But similarly dense on other platforms

0	**Load**	**Test Bit**	**XOR**	**Load**	**Add**	**No-op**
1	**AND**	**AND**	**Shift**	**Add**	**Load**	**Move**
2	**Store**	**OR**	**XOR**	**Add**	**Add**	**Branch**

The sequential part is not!

0	Add	Mov long	No-op	No-op	No-op	No-op
1	Load	Mov long	Mov long	No-op	No-op	No-op
2	Shift,11	Set float	No-op	No-op	No-op	No-op
3	XOR	Move	No-op	No-op	No-op	No-op
4	Shift,7	No-op	No-op	No-op	No-op	No-op
5	AND	No-op	No-op	No-op	No-op	No-op
6	XOR	No-op	No-op	No-op	No-op	No-op
7	SHL,15	No-op	No-op	No-op	No-op	No-op
8	AND	No-op	No-op	No-op	No-op	No-op
9	XOR	No-op	No-op	No-op	No-op	No-op
10	SHL,18	No-op	No-op	No-op	No-op	No-op
11	XOR	No-op	No-op	No-op	No-op	No-op
12	Set float	Compare	Branch	No-op	No-op	No-op
13	Bubble (no work disp					
14	Bubble (no work disp					
15	Bubble (no work disp					
16	Bubble (no work disp					
17	Bubble (no work dispatched, because of FP latency)					
18	Mult FP	No-op	No-op	No-op	No-op	No-op
19	Bubble (no work dispatched, because of FP latency)					
20	Bubble (no work dispatched, because of FP latency)					
21	Bubble (no work dispatched, because of FP latency)					
22	Mult FP	Branch	No-op	No-op	No-op	No-op

```
y = fMt[fCount624++]; /*THE STRAIGHT-LINE PART*/
    y ^=  (y >> 11);  y ^= ((y << 7 ) & kTemperingMaskB );
    y ^= ((y << 15) & kTemperingMaskC );  y ^=  (y >> 18);
    if (y) return ( (Double_t) y * 2.3283064365386963e-10);
```

“Low- hanging fruit”

- **Typically one starts with a given compiler, and moves to:**
- **More aggressive compiler options**
  - For instance:
  - -O2 → -O3 or -O3 → -O3,-ffast-math (g++
  - -O2 → -O3 or -O3 → -O3, -ipo (icc)

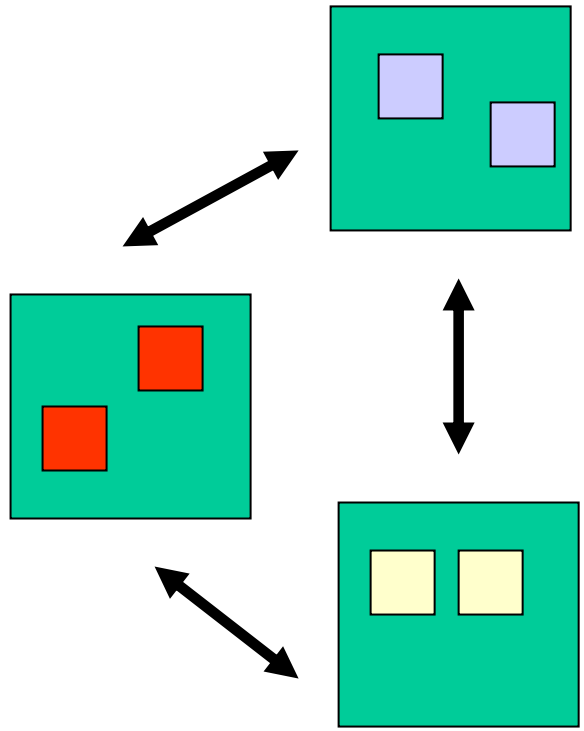
Some options can compromise accuracy or correctness

- **More recent compiler versions**
  - g++ version 3 → g++ version 4
  - icc version 8 → icc version 9
- **Different compilers**
  - GNU → Intel or Pathscale
  - Intel or Pathscale → GNU

May be a burden because of potential source code issues

Interprocedural optimization

- **Let the compiler worry about interprocedural relationship**
  - "icc –ipo"
- **Valid also when building libraries**
  - Archive
  - Shared
- **Cons:**
  - Can lead to code bloat
  - Longer compile times



Probably most useful when combined with heavy optimization for "production" binaries or libraries!

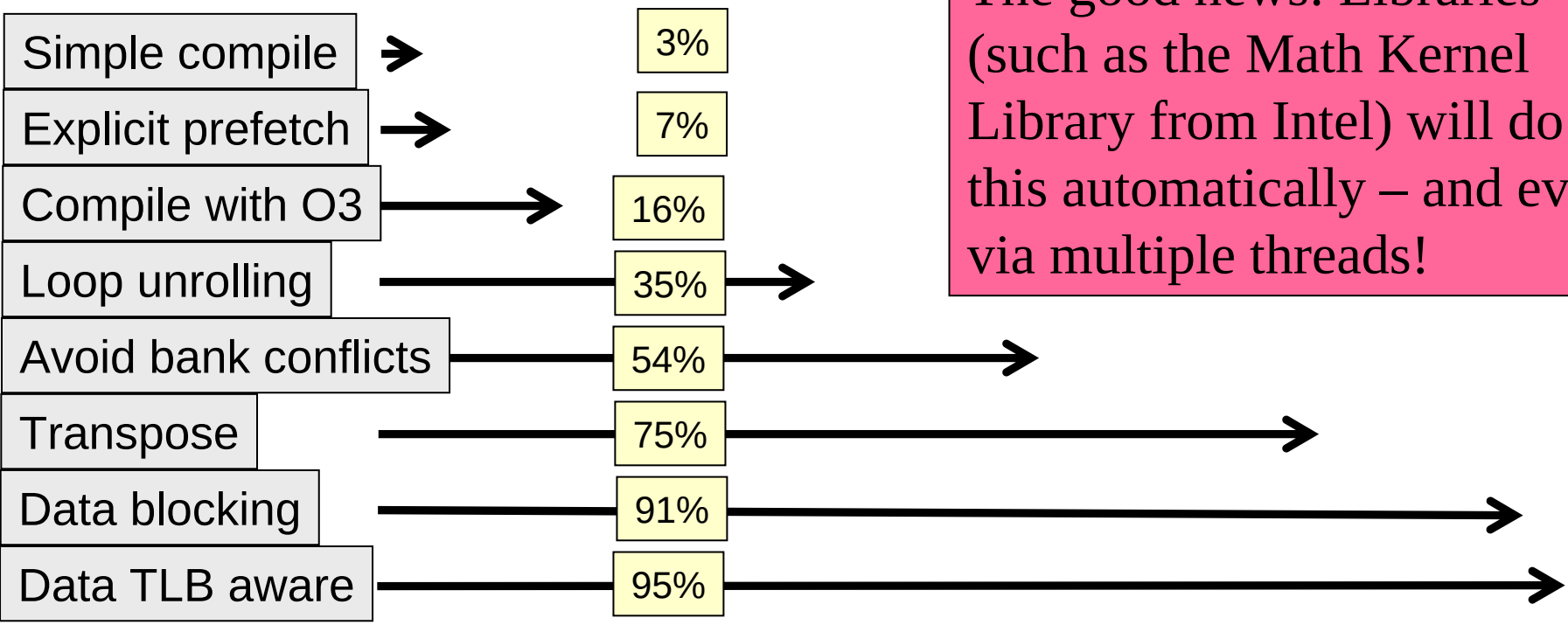
Feedback Optimization

- **Many compilers allow further optimization through training runs**
  - Compile once (to instrument binary)
    - g++ -fprofile-generate
    - icc -prof_gen
  - Run one (or several test cases)
    - ./test40 < test40.in (will run slowly)
  - Recompile w/feedback
    - g++ -fprofile-use
    - icc -prof_use (best results when combined with -O3,-ipo

With icc 9.0 we get ~20% on root stress tests on Itanium, but only ~5% on x86-64

Useful to know the hardware ?

- **Matrix multiply example**
  - From D.Levinthal/Intel (Optimization talk at IDF, spring 2003)
    - Basic algorithm: $C_{ik} = SUM (A_{ij} * B_{jk})$



The good news: Libraries (such as the Math Kernel Library from Intel) will do this automatically – and even via multiple threads!

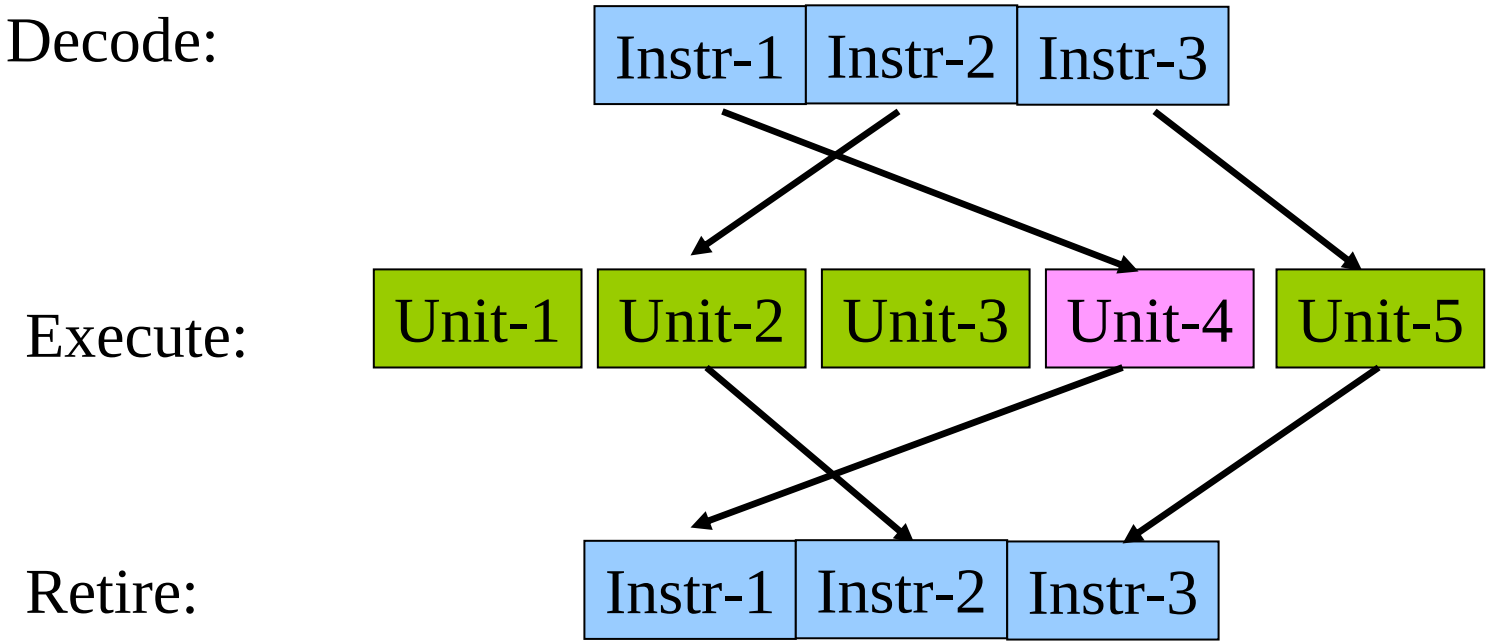
Machine code

- It may be useful to read the machine code

```
.LFB1840:
        movsd   16(%rsi), %xmm2
        movsd   .LC2(%rip), %xmm0
        xorl    %eax, %eax
        movsd   8(%rdi), %xmm4
        andnpd  %xmm2, %xmm0
        ucomisd %xmm4, %xmm0
        ja      .L22
        movsd   (%rsi), %xmm5
        movsd   8(%rsi), %xmm0
        movapd  %xmm2, %xmm3
        movsd   32(%rdi), %xmm1
        addsd   %xmm4, %xmm3
        mulsd   %xmm0, %xmm0
        mulsd   %xmm5, %xmm5
        addsd   %xmm0, %xmm5
        movapd  %xmm4, %xmm0
        mulsd   %xmm3, %xmm1
        subsd   %xmm2, %xmm0
        mulsd   40(%rdi), %xmm3
        movapd  %xmm0, %xmm2
        movsd   16(%rdi), %xmm0
        mulsd   %xmm2, %xmm0
        mulsd   24(%rdi), %xmm2
        addsd   %xmm0, %xmm1
        movsd   .LC3(%rip), %xmm0
        addsd   %xmm2, %xmm3
        mulsd   %xmm0, %xmm1
        mulsd   %xmm0, %xmm3
        divsd   %xmm4, %xmm1
        divsd   %xmm4, %xmm3
        mulsd   %xmm1, %xmm1
        ucomisd %xmm5, %xmm1
        ja      .L28
        mulsd   %xmm3, %xmm3
        movl    $1, %eax
        ucomisd %xmm3, %xmm5
        jbe     .L22
.L28:
        xorl    %eax, %eax
.L22:
        ret
```

CPU execution flow

- **Simplified:**

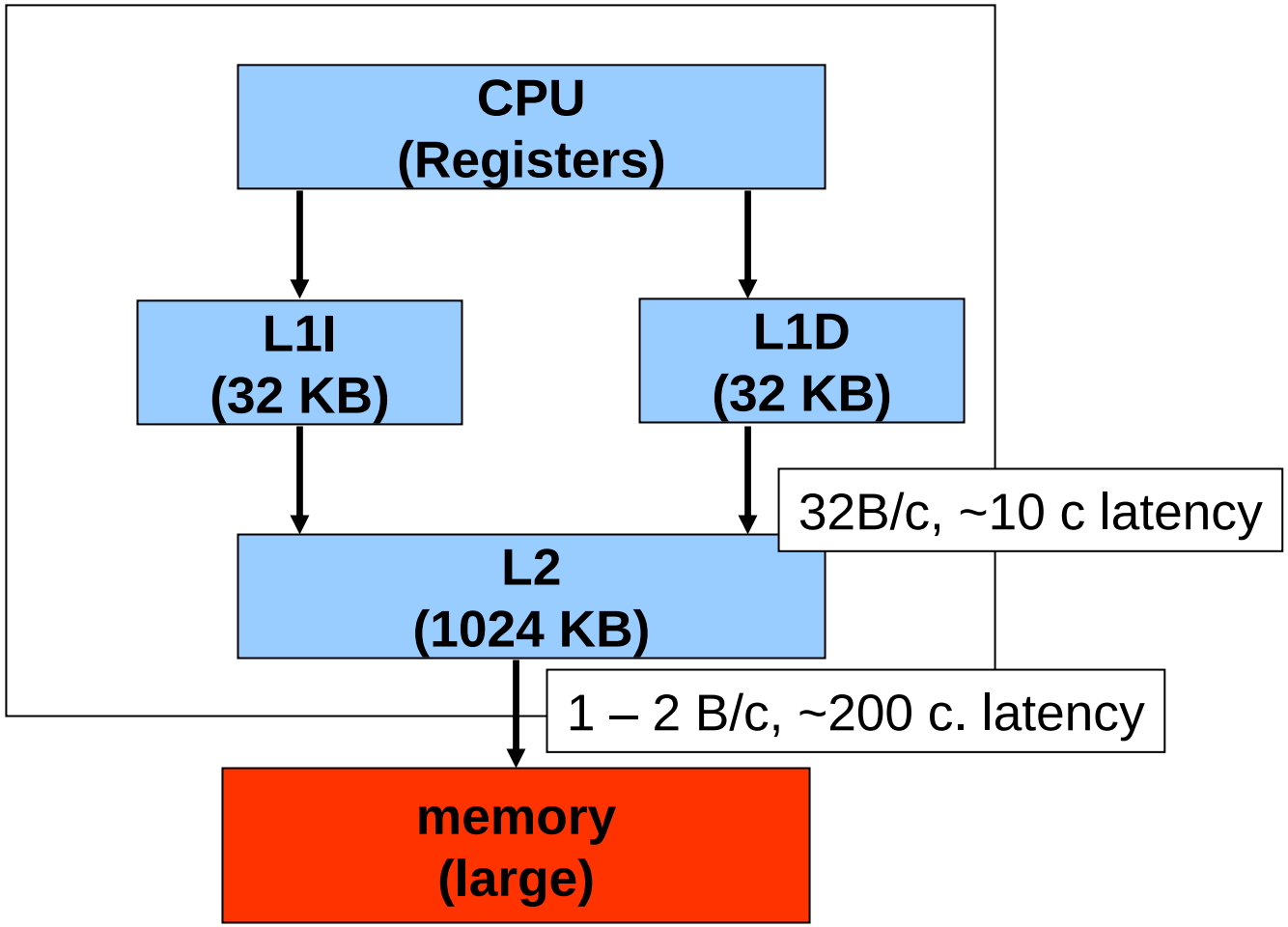


You also need to know how the execution units work: their latency and throughput.

Typical issue: Can the x86 processor issue a new SSE instruction every cycle or every other cycle?

Memory Hierarchy

- From CPU to main memory



You should avoid direct dependency on memory (both latency and throughput)

Cache organization

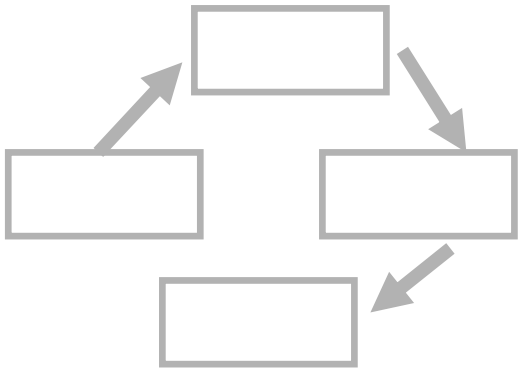
- **For instance: 4-way associativity**

0	64B cache line	64B cache line	64B cache line	64B cache line
1	64B cache line	64B cache line	64B cache line	64B cache line
2	64B cache line	64B cache line	64B cache line	64B cache line
3	64B cache line	64B cache line	64B cache line	64B cache line

• • •

60	64B cache line	64B cache line	64B cache line	64B cache line
61	64B cache line	64B cache line	64B cache line	64B cache line
62	64B cache line	64B cache line	64B cache line	64B cache line
63	64B cache line	64B cache line	64B cache line	64B cache line

Summing up



- **Understand which parts of the “circle” you control**
- **Equip yourself with good tools**
  - IDE + performance monitors
  - Threading analysis tools (?)
- **Check how key algorithms map on to the hardware platforms**
  - Are you at 5% or 95% efficiency?
  - Where do you want to be?
- **Cycle around the change loop frequently**
  - It is hard to get to “peak” performance!

Backup