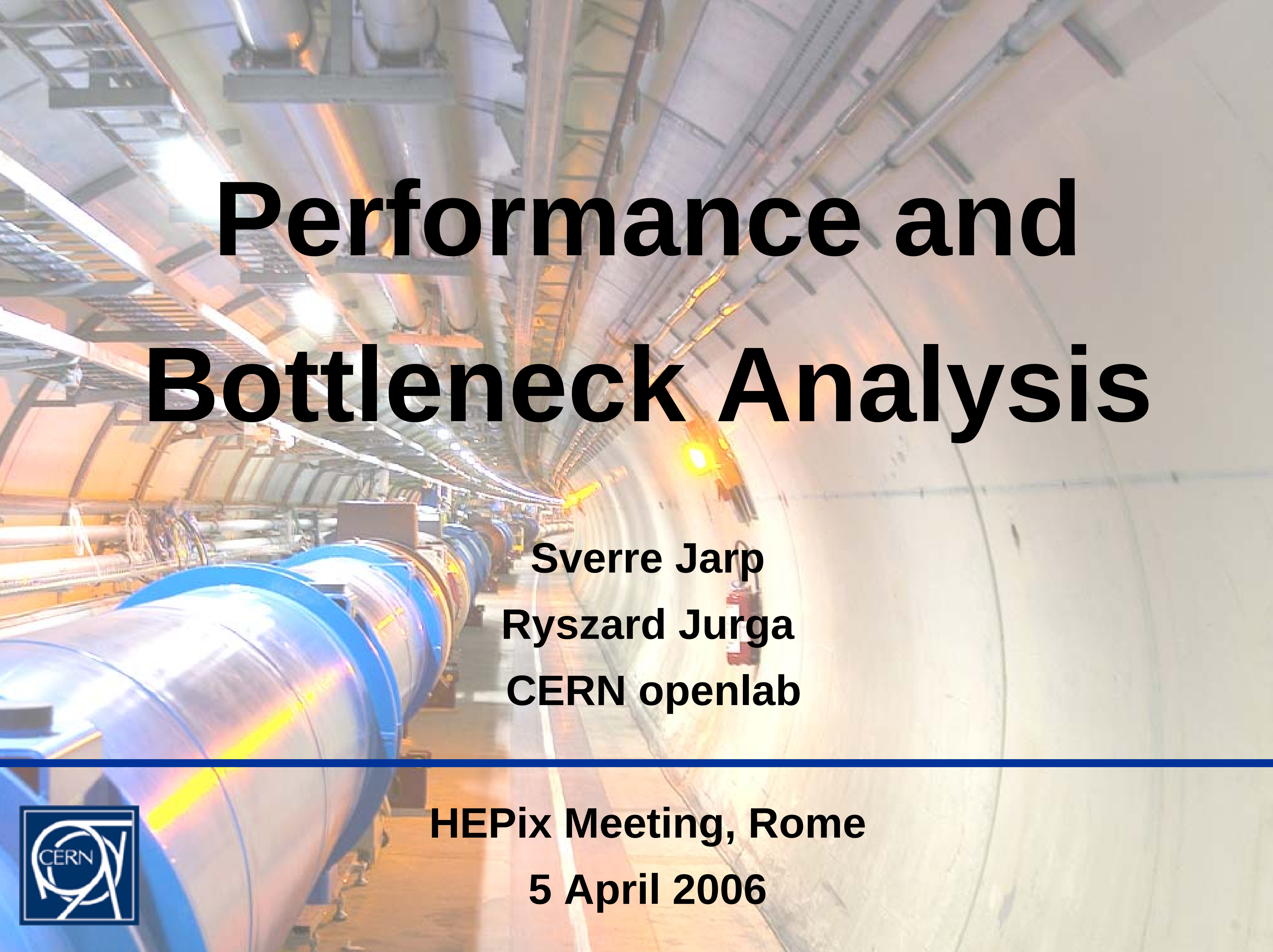


Performance and Bottleneck Analysis

Sverre Jarp

Ryszard Jurga

CERN openlab

HEPix Meeting, Rome

5 April 2006

Part 1
Methodology
Part 2
Measurements

Initial Question

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

Initial Answer

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

I thought “bank conflicts” could not occur in Switzerland.

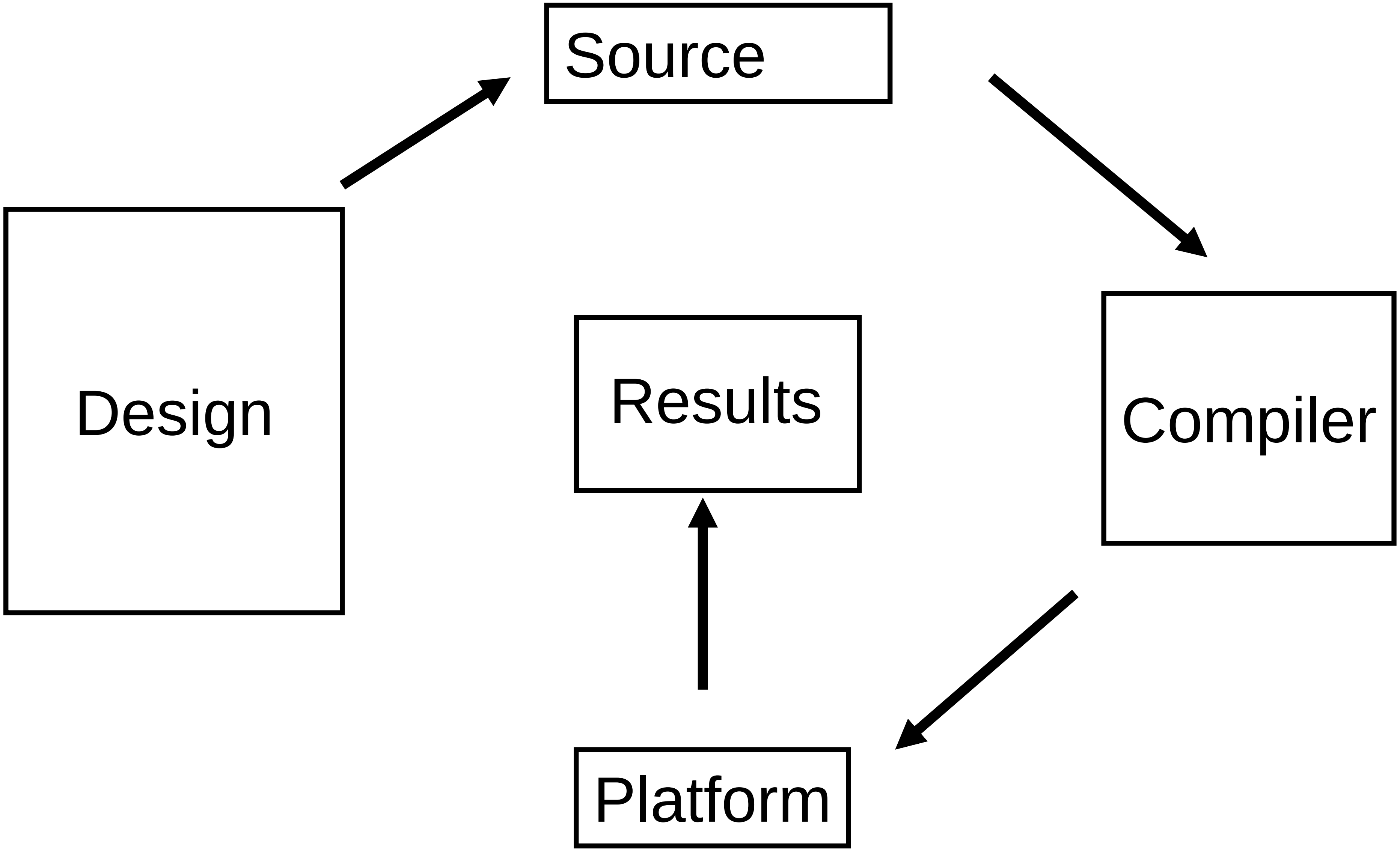
Before we start

- **This is an effort to pass in review a somewhat “systematic approach” to tuning and bottleneck analysis**
  - Main focus is on understanding the “platform”

- **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



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 shared?
  - Monolithic executable or dynamic loading?
  - Could influence via -march=xxx

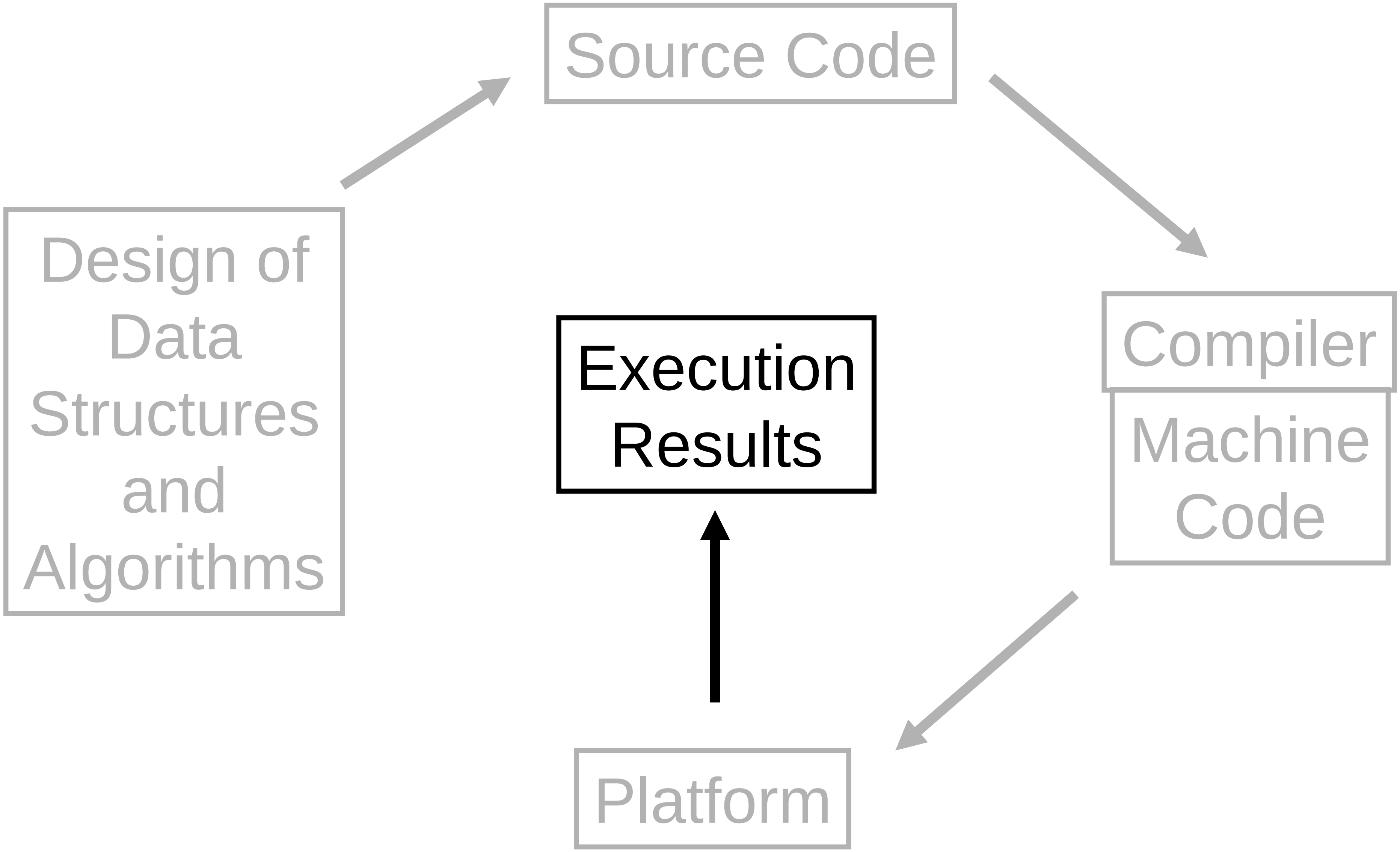
By the way, who knows: -xK/W/N/P/B ?

Fourth step: Platform

- **Choice of:**
  - Manufacturer
  - ISA
  - Processor characteristics
    - Adressing
    - Frequency
    - Core layout
    - Micro-architecture
    - Cache organization
  - Further configuration characteristics

- **Multiple options:**
  - AMD, Intel, Via, IBM, SUN, ..
  - IA32, AMD64/EM64T, IA64, Power, Cell, SPARC, ..
  - Could be:
    - 32- or 64-bit
    - 3.8 GHz Netburst or 2 GHz "Core" ?
    - Single, dual, quad core, ..
    - Pentium 4, Pentium M, AMD K7, ..
    - Different sizes, two/three levels, ..
  - Bus bandwidth, Type/size of memory, ….

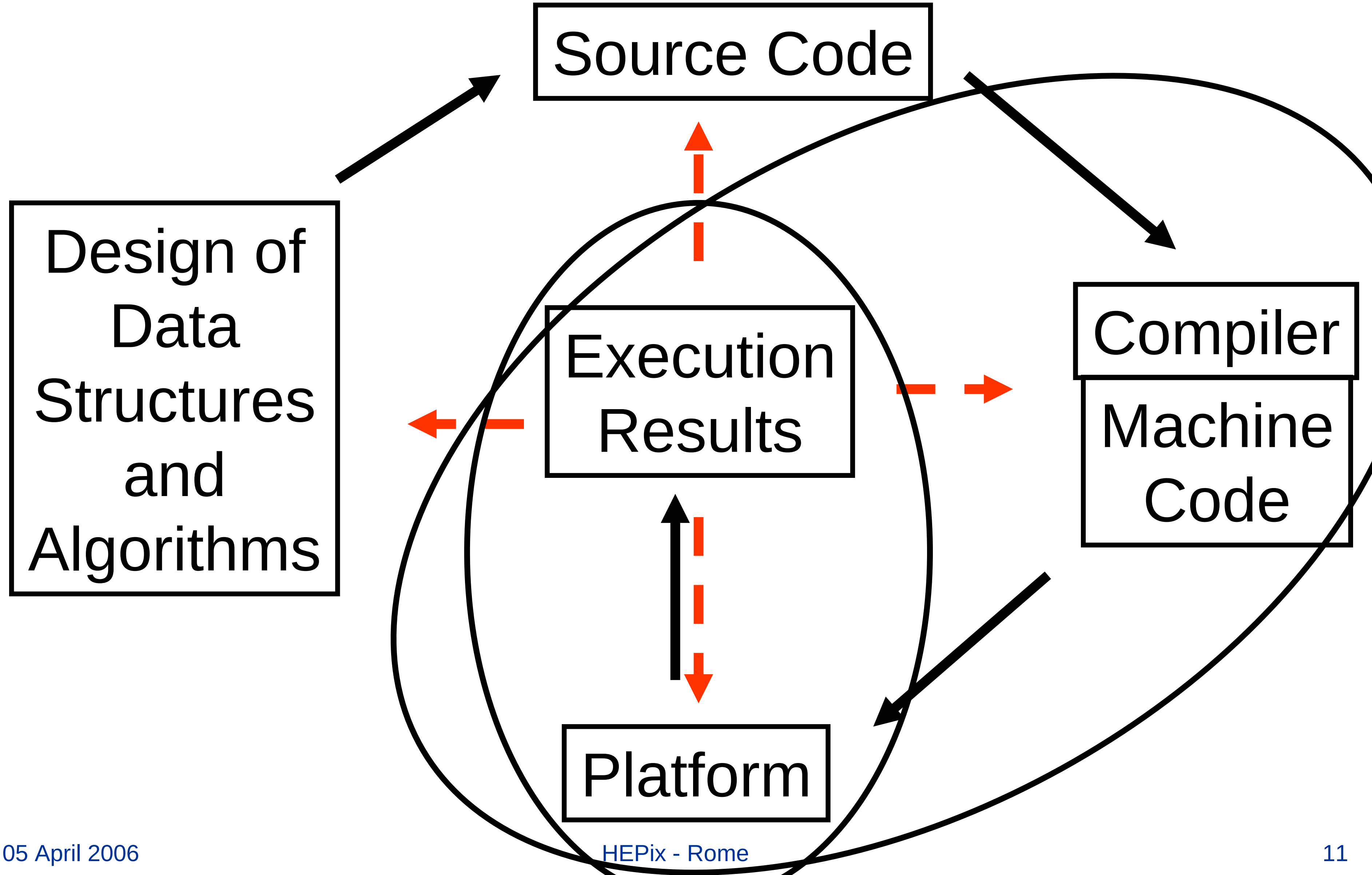
In the end: Execution Results



Back to our cartoon

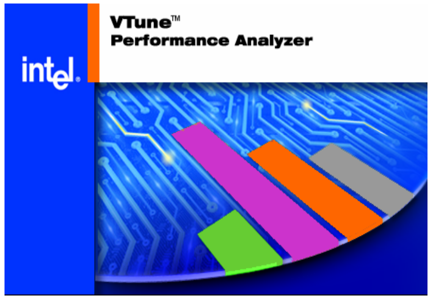
- **First of all, we must guarantee correctness**
- **If we are unhappy with the performance**
  - … and by the way, how do we know when to be happy?
- **We need to look around**
  - Since the culprit can be anywhere

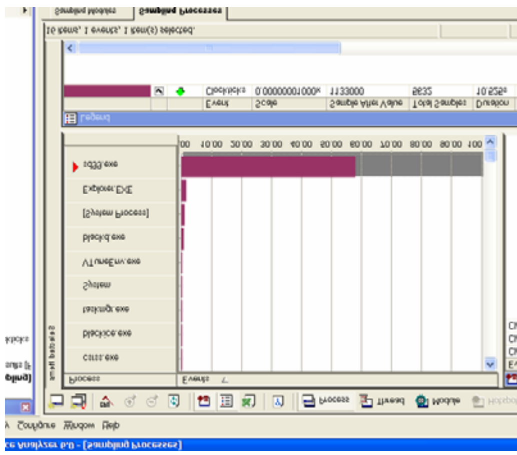
Where to look ?

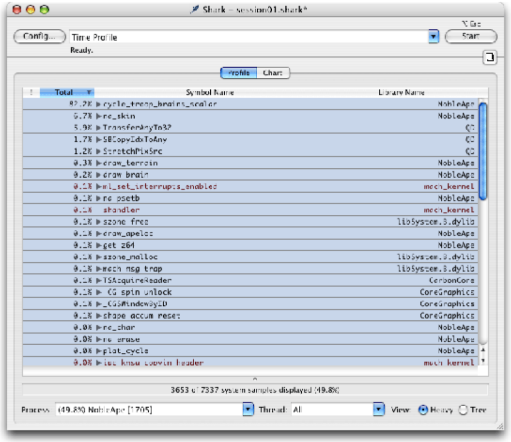


Need a good tool set

- **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, perfsuite, caliper, oprofile, TAU







Time Profile
Ready.

Total	Symbol Name	Library Name
82.2%	cycle_troop_brains_scalar	NobleApe
5.7%	ne_skin	NobleApe
5.9%	TransferAnyTo32	CG
1.7%	SBCopyIdxToAny	CG
1.2%	StretchPixSrc	CG
0.3%	draw_terrain	NobleApe
0.2%	draw brain	NobleApe
0.1%	ml_set_interrupts_enabled	mach_kernel
0.1%	ne psetb	NobleApe
0.1%	shandler	mach_kernel
0.1%	szone free	libSystem.B.dylib
0.1%	draw_apelcc	NobleApe
0.1%	get z64	NobleApe
0.1%	szone_malloc	libSystem.B.dylib
0.1%	mach msg trap	libSystem.B.dylib
0.1%	TSAcquireReader	CarbonCore
0.1%	CG spin unlock	CoreGraphics
0.1%	_CGSWindowByID	CoreGraphics
0.1%	shape accum reset	CoreGraphics
0.0%	ne_char	NobleApe
0.0%	ne erase	NobleApe
0.0%	plat_cycle	NobleApe
0.0%	iac knsu copvin header	mach kernel

3653 of 7337 system samples displayed (49.8%)

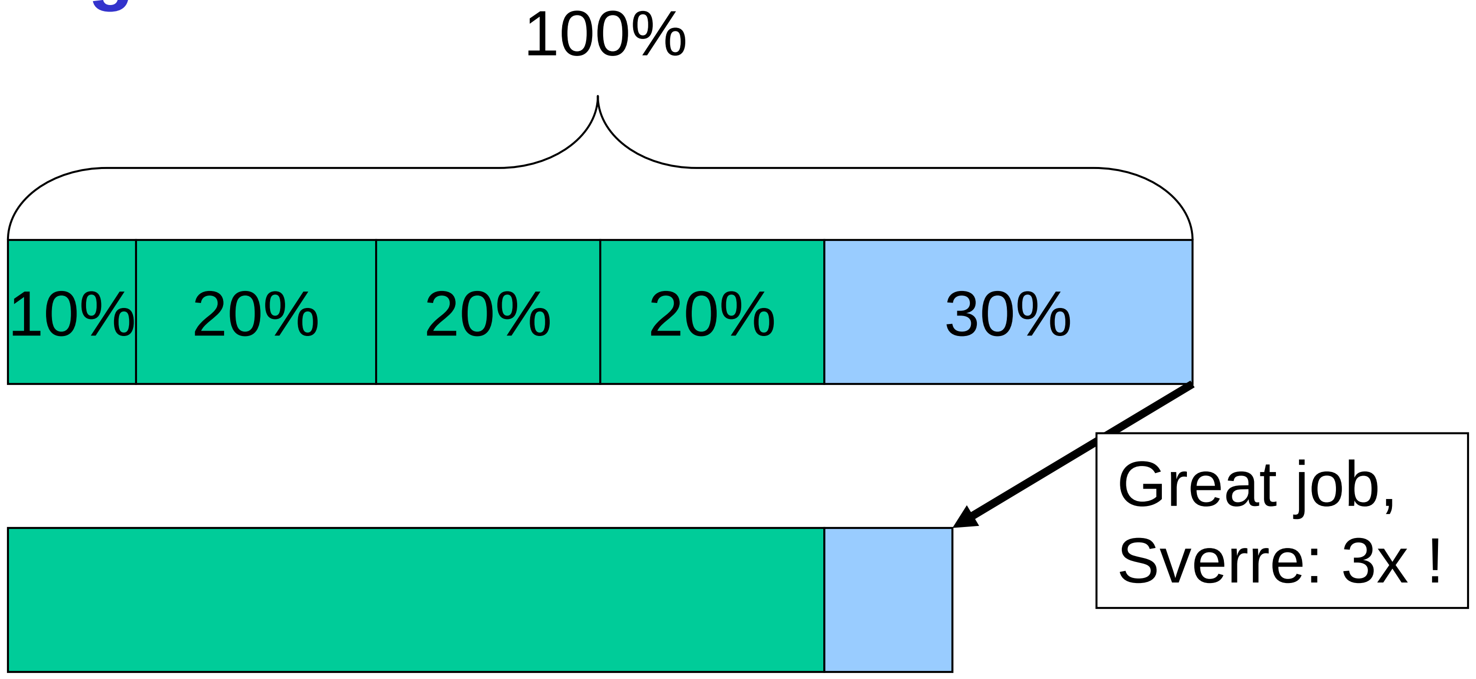
Process: (49.8%) NobleApe [1705] Thread: All View: [✓ Heavy] Tree

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

Amdahl's Law

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



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

Typical profile
Stop press: G4ATLAS

```
     Samples    Self %   Total %   Module


   11767458    36.64%    36.64%   libG4geometry.so
    5489494    17.09%    53.73%   libG4processes.so
    2283674     7.11%    60.85%   libG4tracking.so
    2146178     6.68%    67.53%   libm-2.3.2.so
    2057144     6.41%    73.93%   libstdc++.so.5.0.3
    1683623     5.24%    79.18%   libc-2.3.2.so
     933872     2.91%    82.08%   libCLHEP-GenericFunctions-1.9.2.1.so
     685894     2.14%    84.22%   libG4track.so
     655282     2.04%    86.26%   libCLHEP-Random-1.9.2.1.so
     524236     1.63%    87.89%   libpthread-0.60.so
     283521     0.88%    88.78%   libCLHEP-Vector-1.9.2.1.so
     265656     0.83%    89.60%   libG4materials.so
     205836     0.64%    90.24%   libG4Svc.so
     197690     0.62%    90.86%   libG4particles.so
     190272     0.59%    91.45%   ld-2.3.2.so
     150757     0.47%    91.92%   libCore.so (ROOT)
     149525     0.47%    92.39%   libFadsActions.so
     126111     0.39%    92.78%   libG4event.so
     123206     0.38%    93.16%   libGaudiSvc.so
```

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,-funroll-loops, -ffast-math (g++
  - -O2 → -O3, -ipo (icc)

Some options can compromise accuracy or correctness

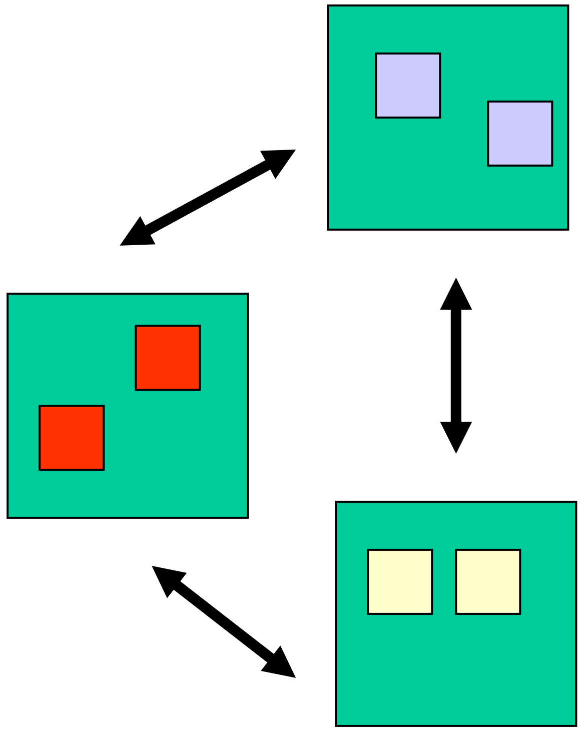
- **More recent compiler versions**
  - g++ version 3 → g++ version 4
  - icc version 8 → icc version 9

- **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!

Useful to know the hardware ?

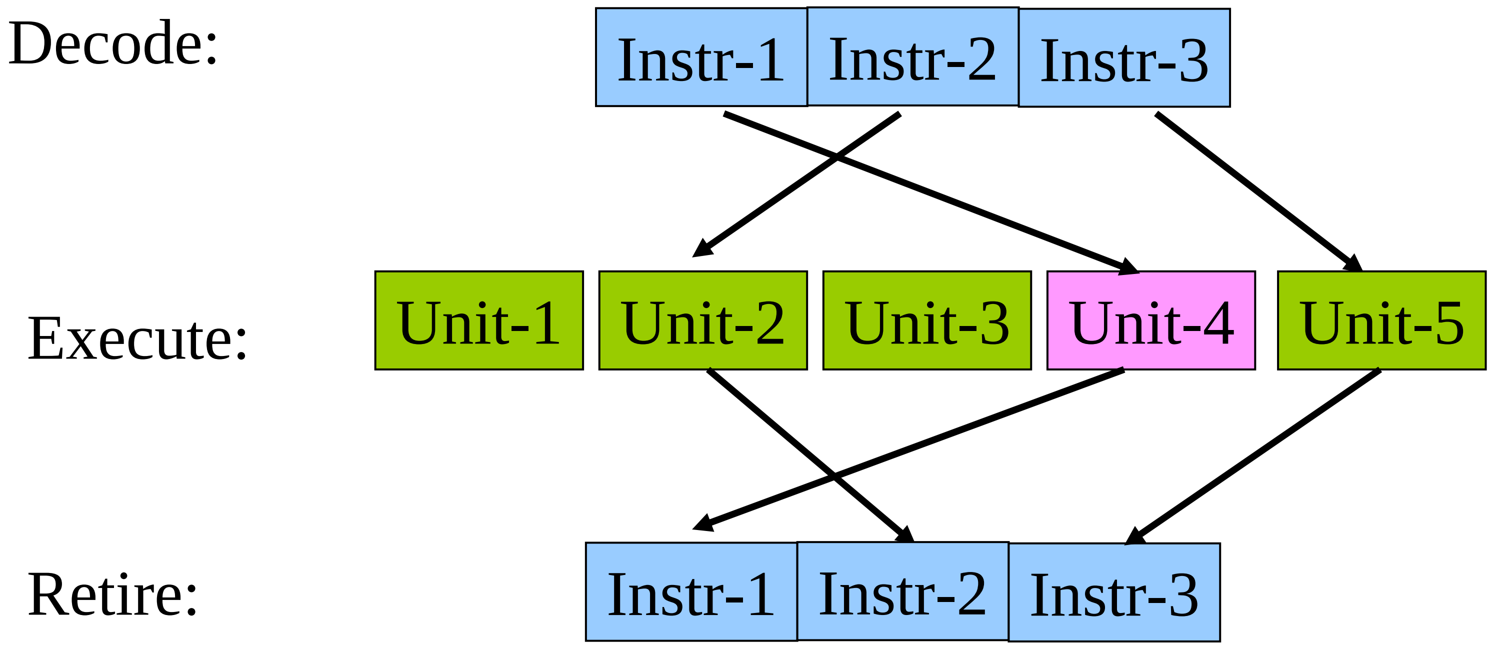
- **Matrix multiply example (IPF)**
  - From D.Levinthal/Intel (Optimization talk at IDF, spring 2003)
    - Basic algorithm: $C_{ik} = SUM (A_{ij} * B_{jk})$

Simple compile	3%
Explicit prefetch	7%
Compile with O3	16%
Loop unrolling	35%
Avoid bank conflicts	54%
Transpose	75%
Data blocking	91%
Data TLB aware	95%

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

CPU execution flow

- **Simplified:**

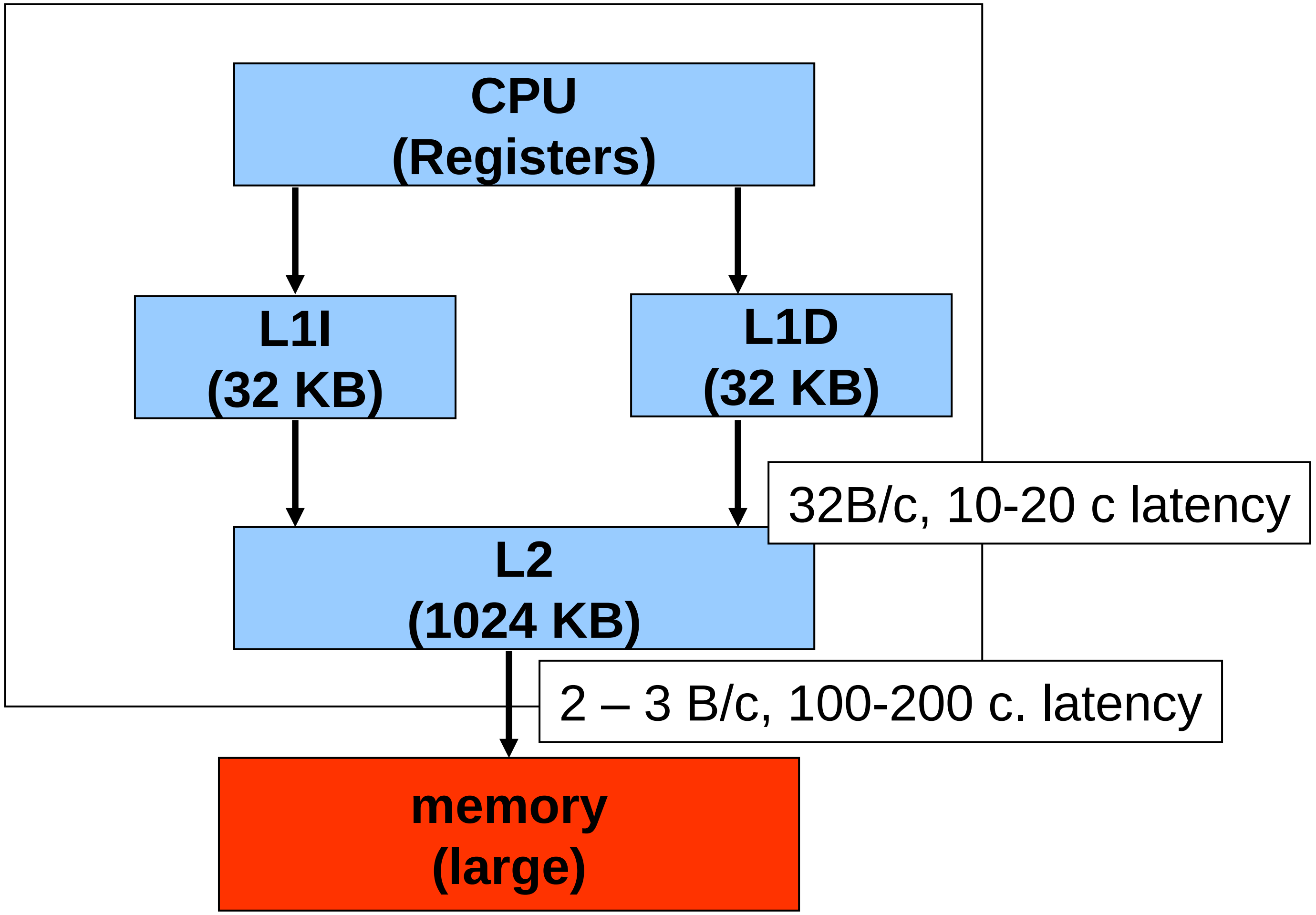


You also need to know how the execution units work: their latency and throughput.

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

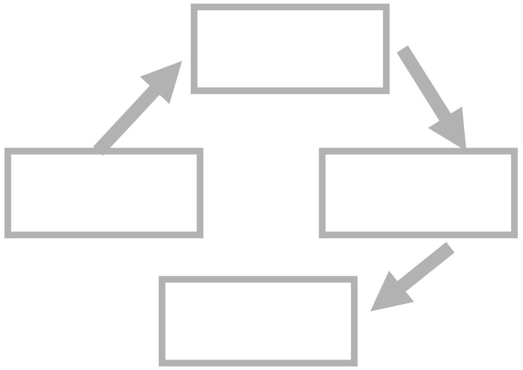
Memory Hierarchy

- From CPU to main memory



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

Summing up



- **Understand which parts of the "circle" you control**
- **Equip yourself with good tools**
  - **Get access to hw performance counters**
  - Use IDE w/integrated performance tools
  - 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!

Part 1

Methodology

Part 2

Measurements

(first results)

Monitoring H/W

- **Special on-chip hardware of modern CPU**
  - Direct access to CPU counts of branch prediction, data and instruction caches, floating point instructions, memory operations
  - Event detectors, counters
    - Itanium2: 4 (12) counters, 100+ events to monitor
    - **Pentium 4, Xeon**: 44 event detectors, 18 counters
  - Linux interfaces and libraries:
    - Part of kernel in order to per-thread and per-system measurements
    - Perfmon2
      - uniform across all hardware platforms; event multiplexing
      - Full support mainly in the Itanium (integrated w/2.6 kernel)
    - **Perfctr**
      - per-thread and system-wide measurements
      - user and kernel domain; kernels 2.4 & 2.6; No multiplexing
      - Support for a lot of CPUs (P Pro/II/III/IV/Xeon), no support for Itanium
      - Almost no documentation apart from comments in source files
      - Require a deep understanding of performance monitoring features of every processors

Pentium 4 Performance Monitoring Features

- 44 event detectors, 9 pairs of counters
- 2 control registers (ESCR, CCCR)
- 2 classes of events:
    - Non-retirement events – those that occur any time during execution (1 counter)
    - At-retirement events – those that occurred on execution path and their results were committed in architectural state (1 or 2 counters)
- multiplexing

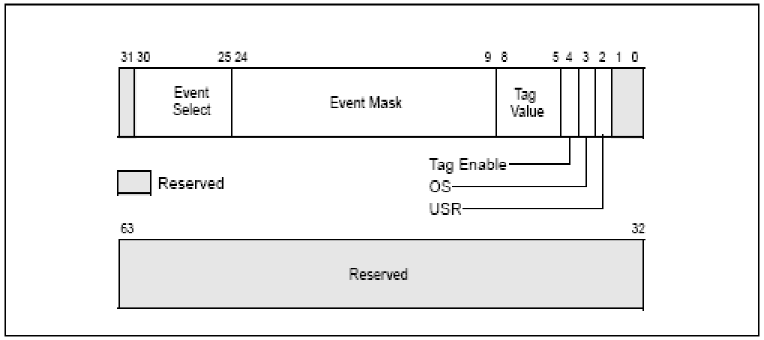


Figure 18-9. Event Selection Control Register (ESCR) for Pentium 4 and Intel Xeon Processors without HT Technology Support

from Intel documentation

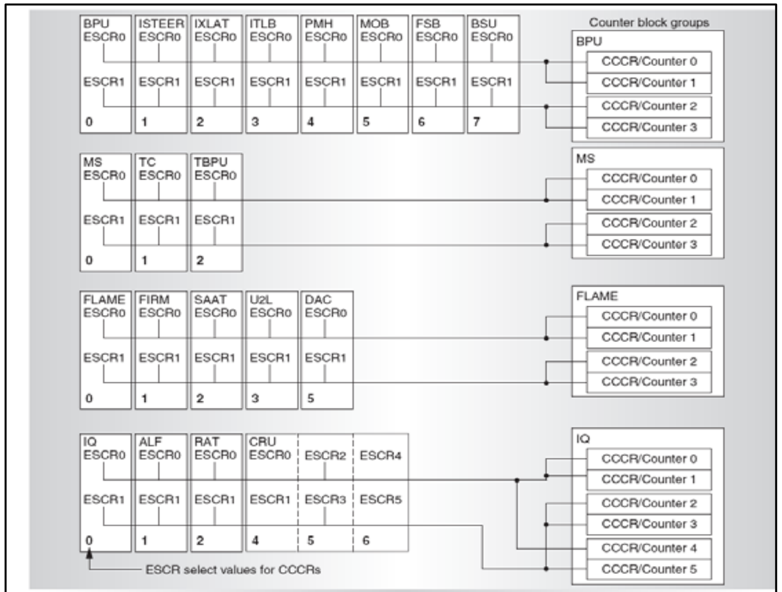


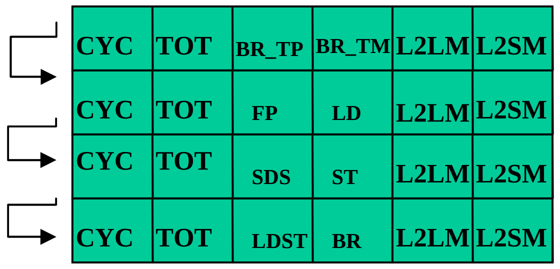
Figure 2. Interconnections among event detectors and event select control registers (ESCRs), and their associated counters and counter configuration control registers (CCCRs).

from B. Sprunt "Pentium 4 Performance-Monitoring Features"

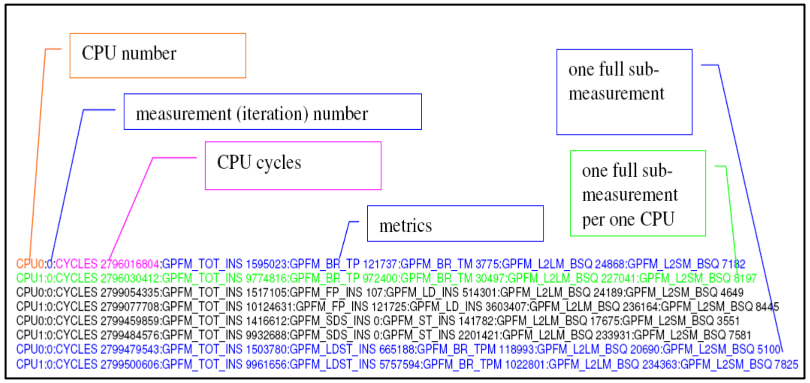
Monitoring tool - *gpfmon*

- uses perfctr
- enables multiplexing
- user and kernel domain
- per single or total CPU
- events:

CYC = CPU cycles
TOT = Instructions completed
BR_TP = Branch taken predicted
BR_TM = Branch taken mispredicted
L2LM = L2 load missed
L2SM = L2 store missed
FP = Floating point instructions
SDS = scalar instructions
LD = load intstructions
ST = store instructions
BR = BR_TP+BR_TM
LDST = LD+ST



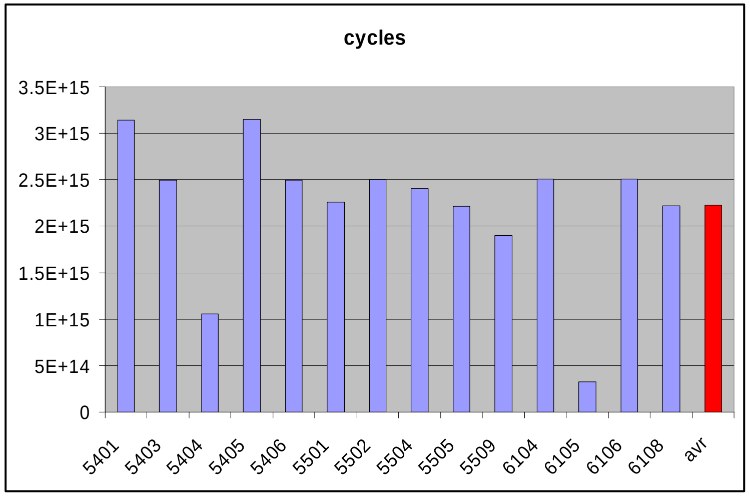
CYC	TOT	BR_TP	BR_TM	L2LM	L2SM
CYC	TOT	FP	LD	L2LM	L2SM
CYC	TOT	SDS	ST	L2LM	L2SM
CYC	TOT	LDST	BR	L2LM	L2SM


CPU number
measurement (iteration) number
CPU cycles
metrics
one full sub-measurement
one full sub-measurement per one CPU

```
CPU0:0:CYCLES 2796016804:GPFM_TOT_INS 1595023:GPFM_BR_TP 121737:GPFM_BR_TM 3775:GPFM_L2LM_BSQ 24868:GPFM_L2SM_BSQ 7182
CPU1:0:CYCLES 2796030412:GPFM_TOT_INS 9774816:GPFM_BR_TP 972400:GPFM_BR_TM 30497:GPFM_L2LM_BSQ 227041:GPFM_L2SM_BSQ 8197
CPU0:0:CYCLES 2799054335:GPFM_TOT_INS 1517105:GPFM_FP_INS 107:GPFM_LD_INS 514301:GPFM_L2LM_BSQ 24189:GPFM_L2SM_BSQ 4649
CPU1:0:CYCLES 2799077708:GPFM_TOT_INS 10124631:GPFM_FP_INS 121725:GPFM_LD_INS 3603407:GPFM_L2LM_BSQ 236164:GPFM_L2SM_BSQ 8445
CPU0:0:CYCLES 2799459859:GPFM_TOT_INS 1416612:GPFM_SDS_INS 0:GPFM_ST_INS 141782:GPFM_L2LM_BSQ 17675:GPFM_L2SM_BSQ 3551
CPU1:0:CYCLES 2799484576:GPFM_TOT_INS 9932688:GPFM_SDS_INS 0:GPFM_ST_INS 2201421:GPFM_L2LM_BSQ 233931:GPFM_L2SM_BSQ 7581
CPU0:0:CYCLES 2799479543:GPFM_TOT_INS 1503780:GPFM_LDST_INS 665188:GPFM_BR_TPM 118993:GPFM_L2LM_BSQ 20690:GPFM_L2SM_BSQ 5100
CPU1:0:CYCLES 2799500606:GPFM_TOT_INS 9961656:GPFM_LDST_INS 5757594:GPFM_BR_TPM 1022801:GPFM_L2LM_BSQ 234363:GPFM_L2SM_BSQ 7825
```

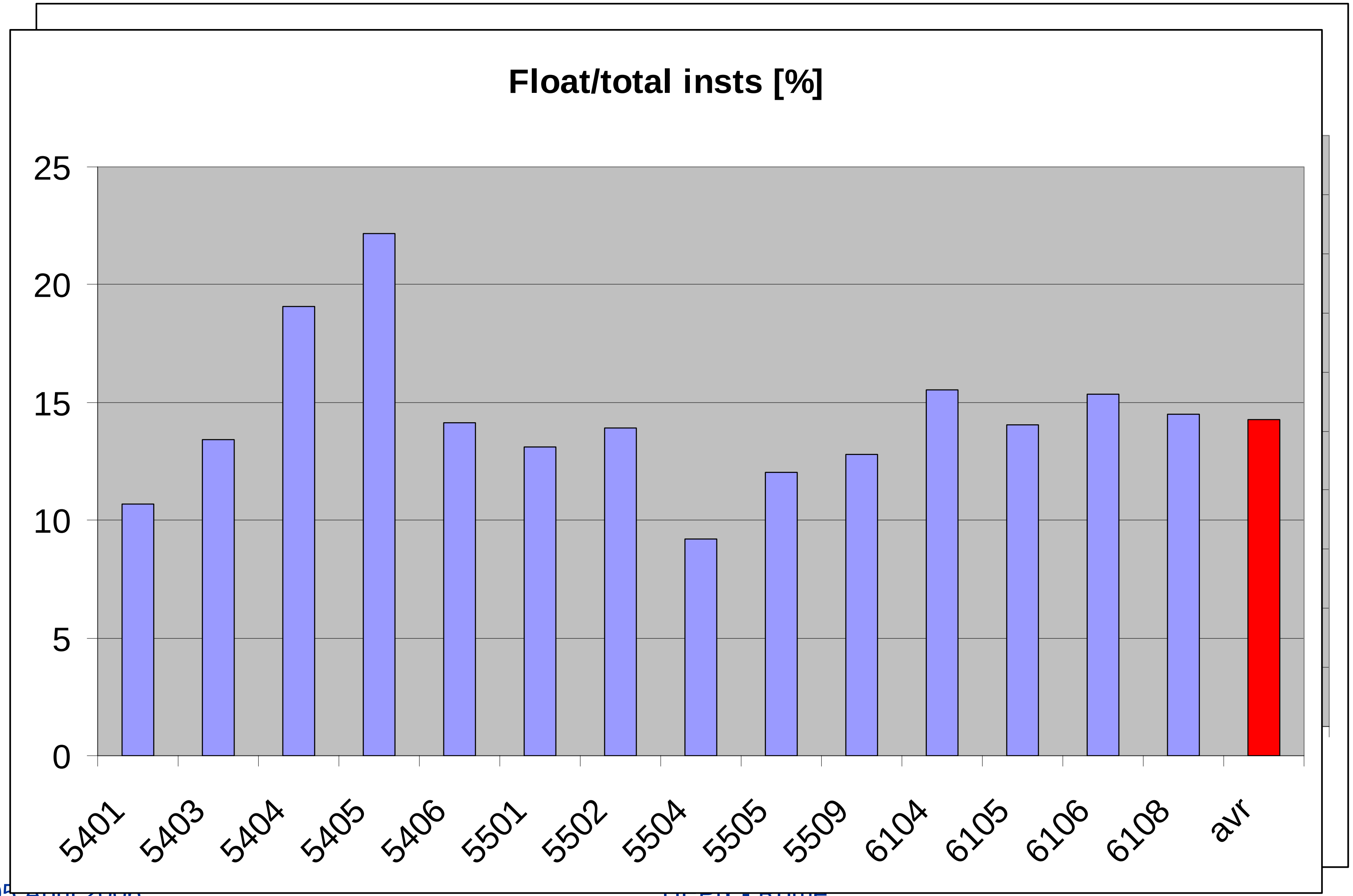
lxbatch monitoring

- 14 machines
- Running from 2 days to 2 weeks
- Nocona (10) – lxb5xxx
- Irwindale (4) – lxb6xxx
- 2.8 GHz
- Cache: 1 MB L2 (10) 2 MB L2 (4)
- SL3 (kernel 2.4)

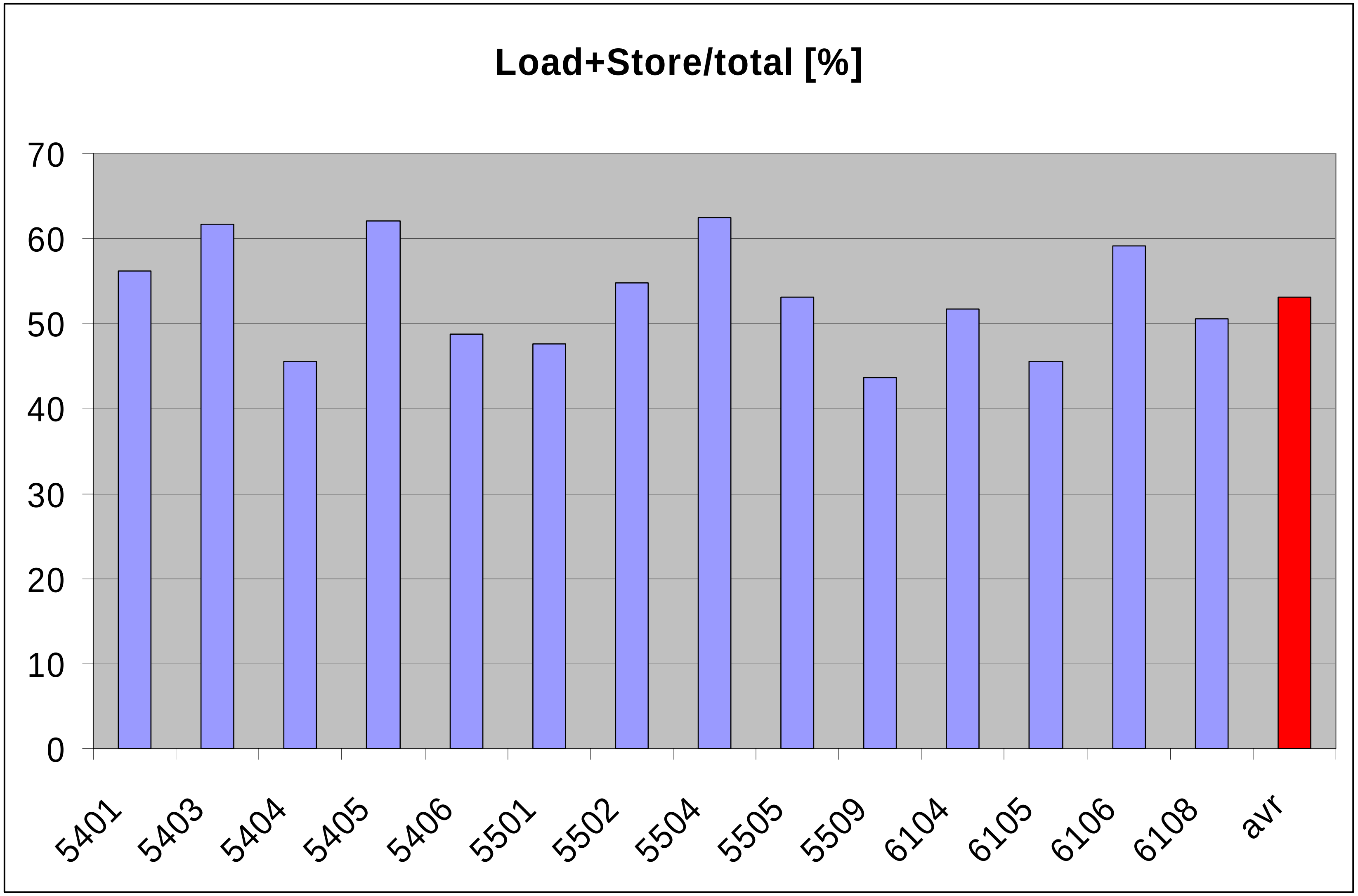


lxbatch monitoring

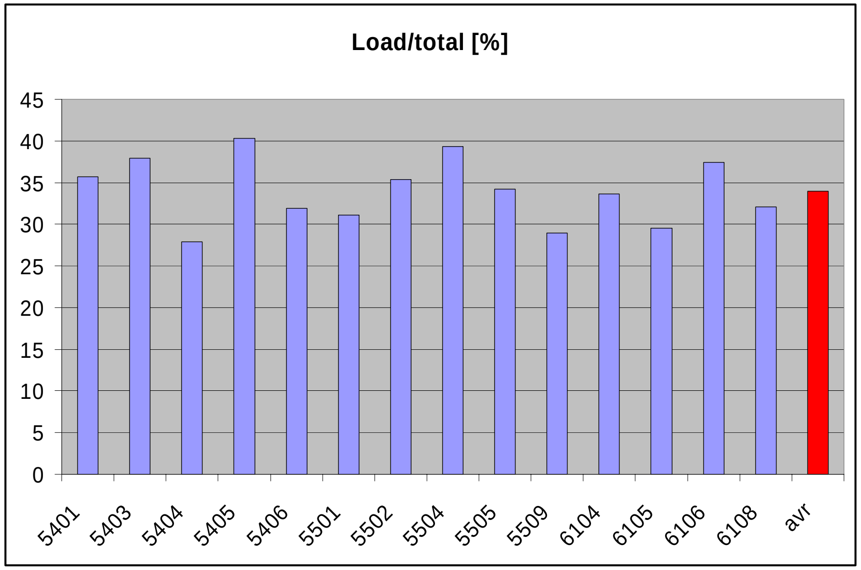
- **Monitors everything: Kernel, User, both CPUs**

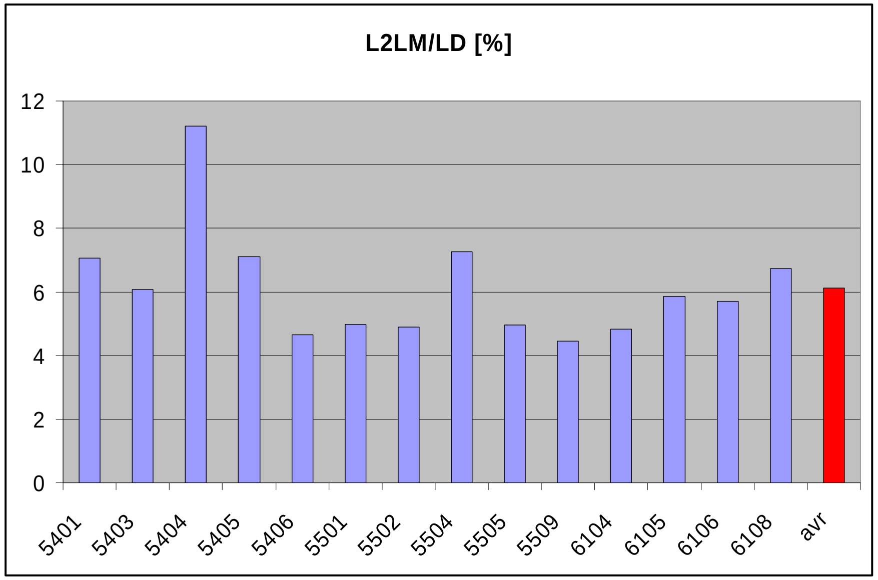


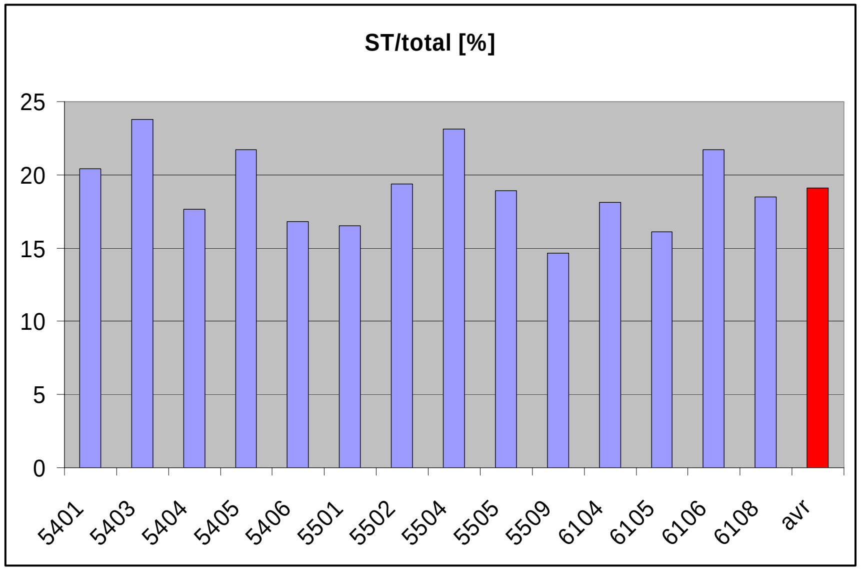
lxbatch memory operations

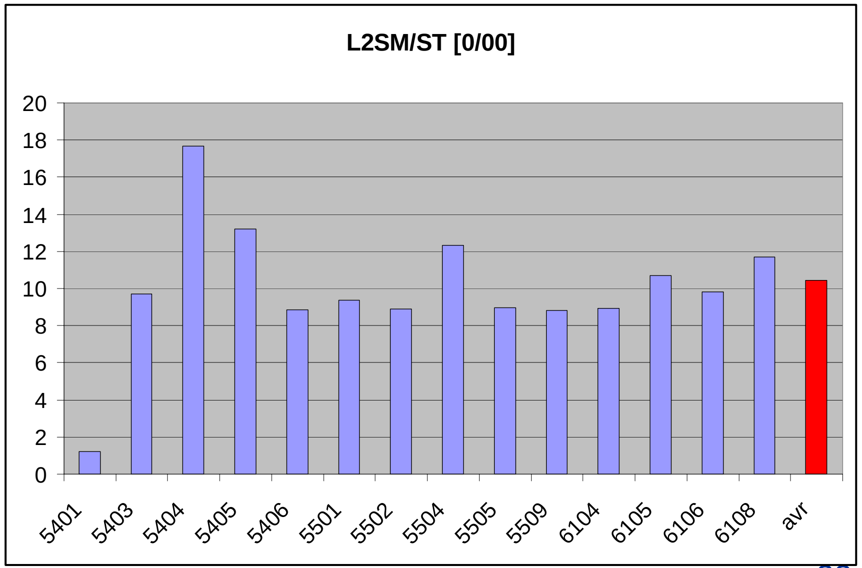


lxbatch memory operations

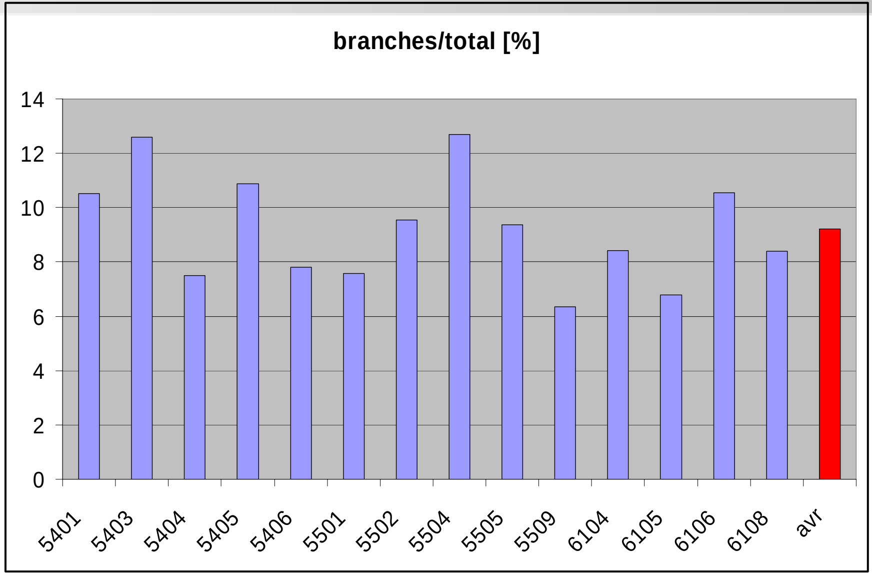


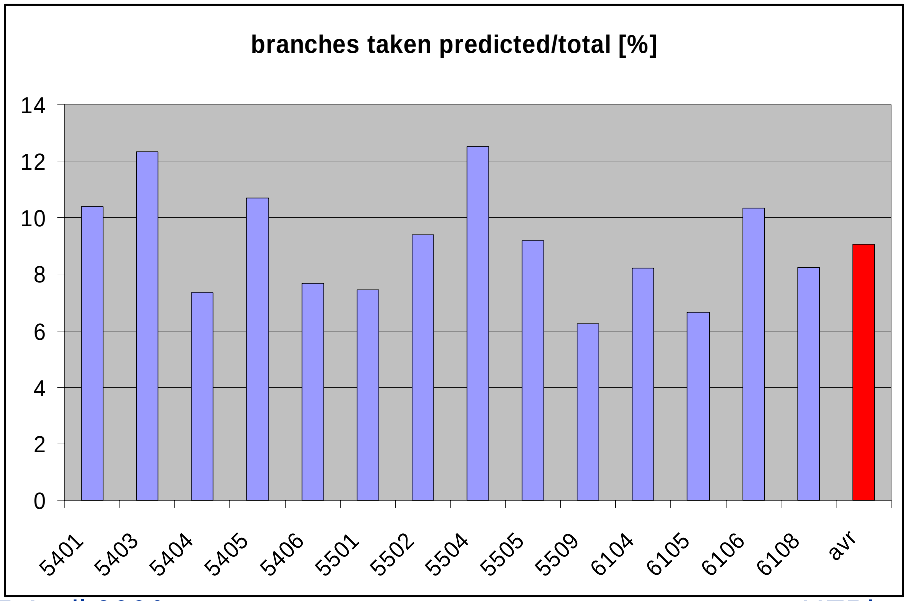


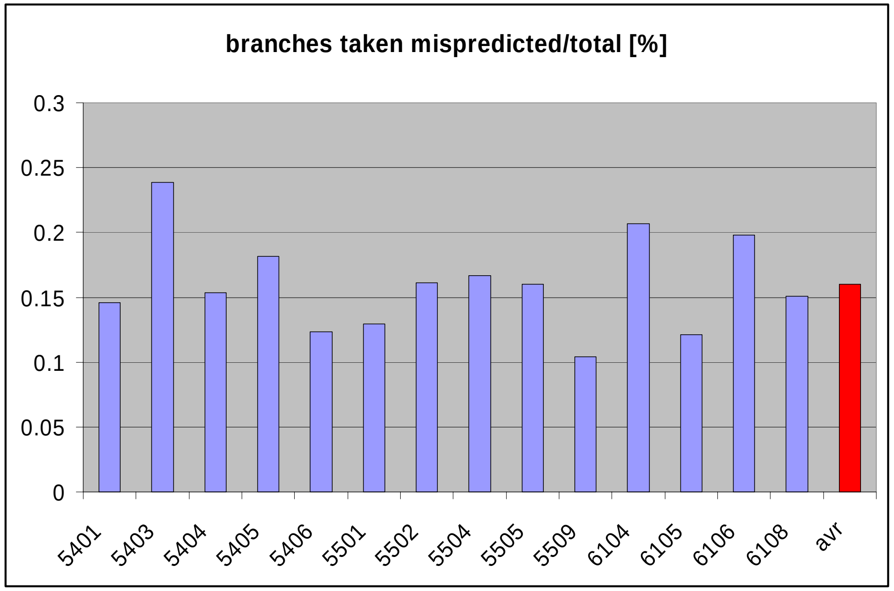




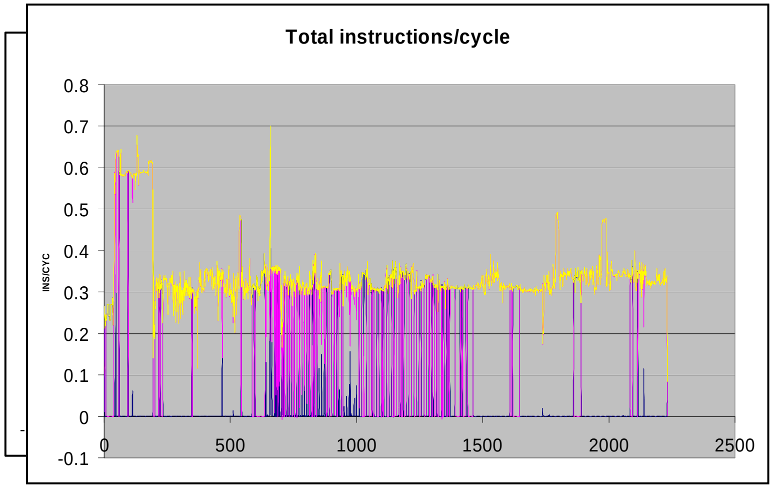
lxbatch - branches







G4Atlas simulation (3 events)

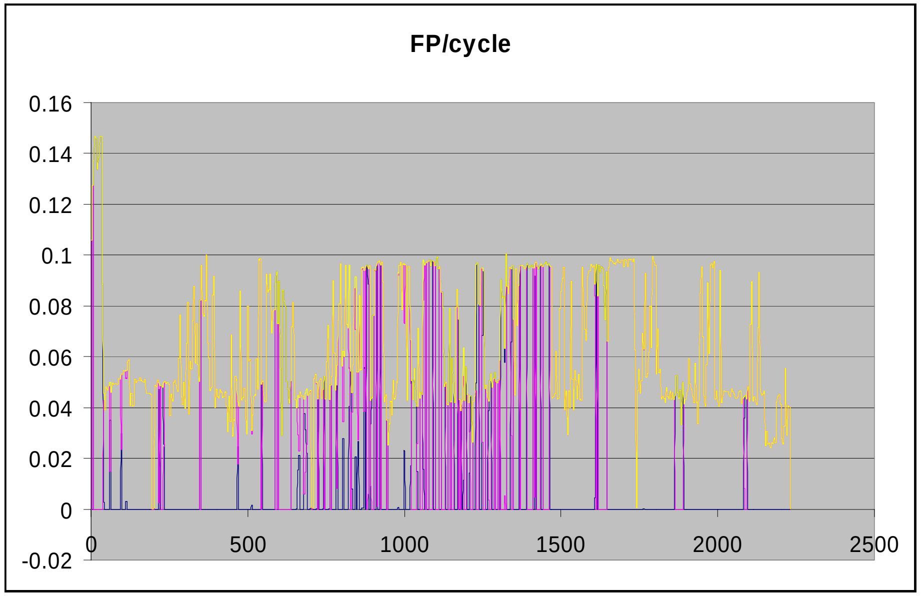


Total instructions

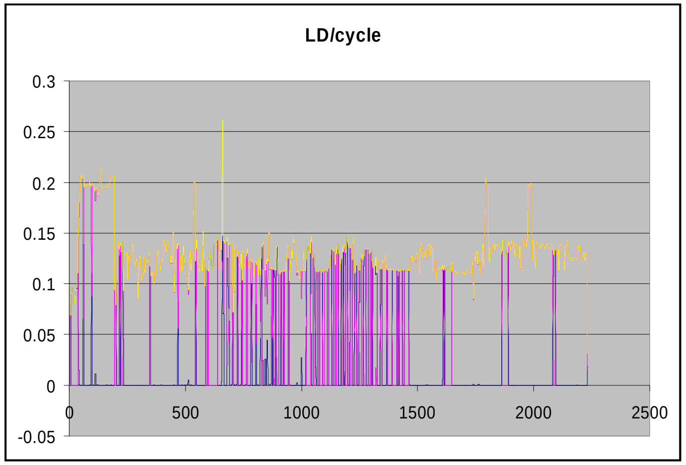
Cycles	**6252 * 10^9**
Total inst	**2136 * 10^9**
TOT INS/CYC	**0.342**

Floating-point instructions

FP	**397 * 10^9**
FP/TOT	**0.186**

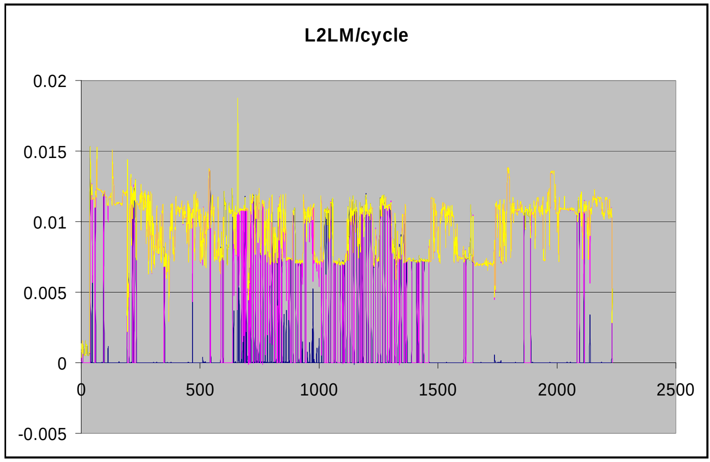


G4Atlas simulation

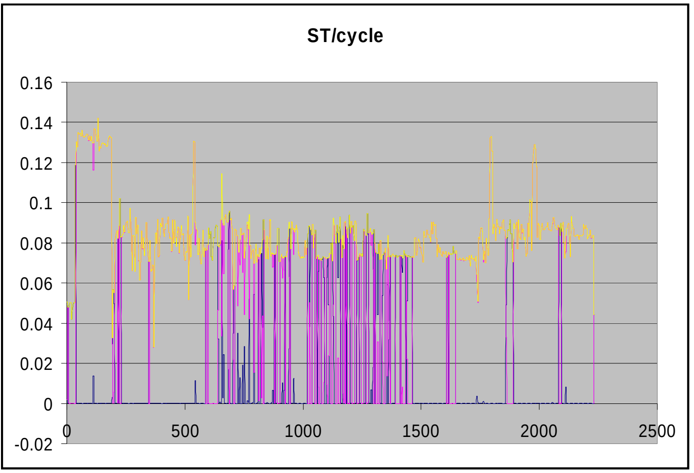


Loads 38%

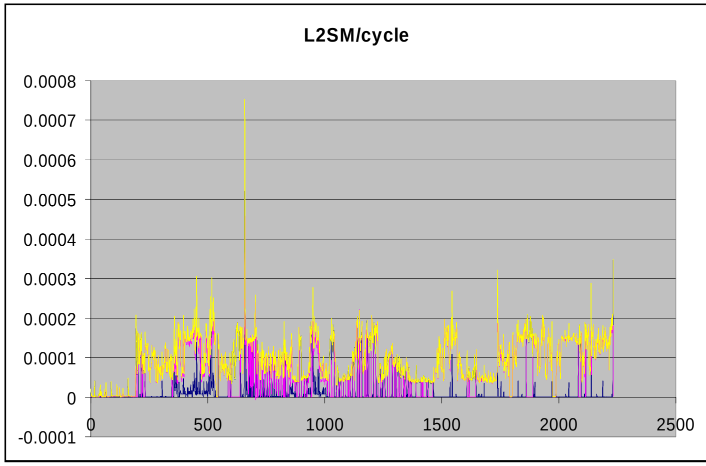
LD	814 * 10^9
LD/TOT	0.38
L2LM	60 * 10^9
L2LM/LD	0.074



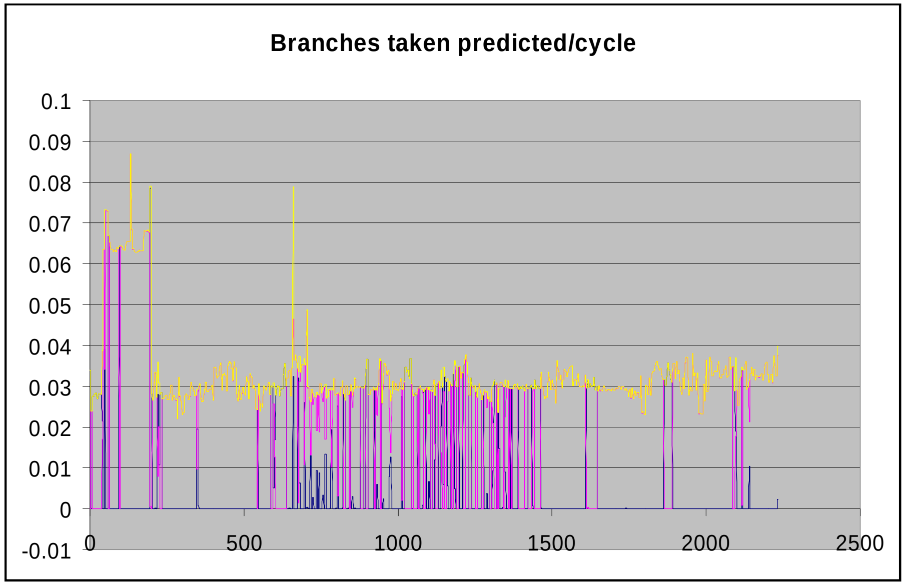
Stores 25%

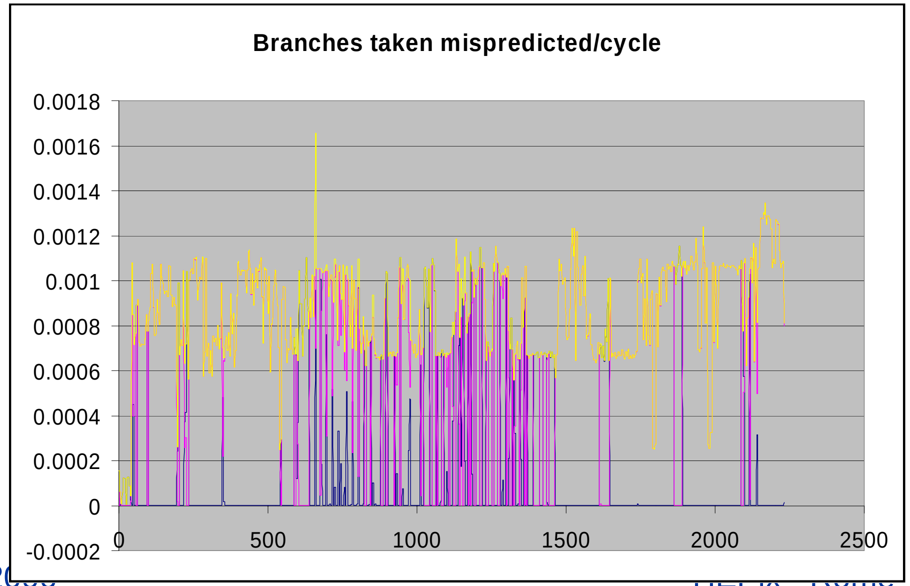


ST	528 * 10^9
ST/TOT	0.247
L2SM	0.60 * 10^9
L2SM/ST	0.00113



G4Atlas simulation





Branches 10%

BR_TP	218 * 10^9
BR_TM	5.4 * 10^9
BR_TP/TOT	0.097
BR_TM/TOT	0.00252

Initial conclusions (1)

- **Counter by counter we see:**
  - Inst/Cycle:
    - Average 0.5 (from LXBATCH)
    - When G4ATLAS has received input file: 0.6 - 0.7
    - In any case, very far from 3 !!
  - Load + Store
    - 34 % + 18 % (38 % + 25 % for G4ATLAS)
    - Too many stores: Are jobs doing too much copying?
  - Total mix (lxbatch)
    - L + S: 52 %
    - FLP: 14 %
    - Branches taken: 9 %
    - Other: 25% → What are these?

Initial conclusions (2)

- **Counter by counter we see:**
  - Branches Taken Predicted incorrectly
    - 2.7 % (of Branches Taken – G4ATLAS)
    - Probably OK, but need to check Branch-Not-Taken counts
  - L2 Store Misses:
    - 0.1% (of stores – G4ATLAS)
    - Very low, so OK
  - L2 Load Misses
    - 6 – 7 % (of loads)
    - Need to understand in more detail
    - Could be multiple hits for a single cache line
  - Cache size
    - 1 MB or 2 MB ?
    - Do not see fewer L2LD misses

CERN openlab
for DataGrid applications

QUESTIONS?

Backup

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

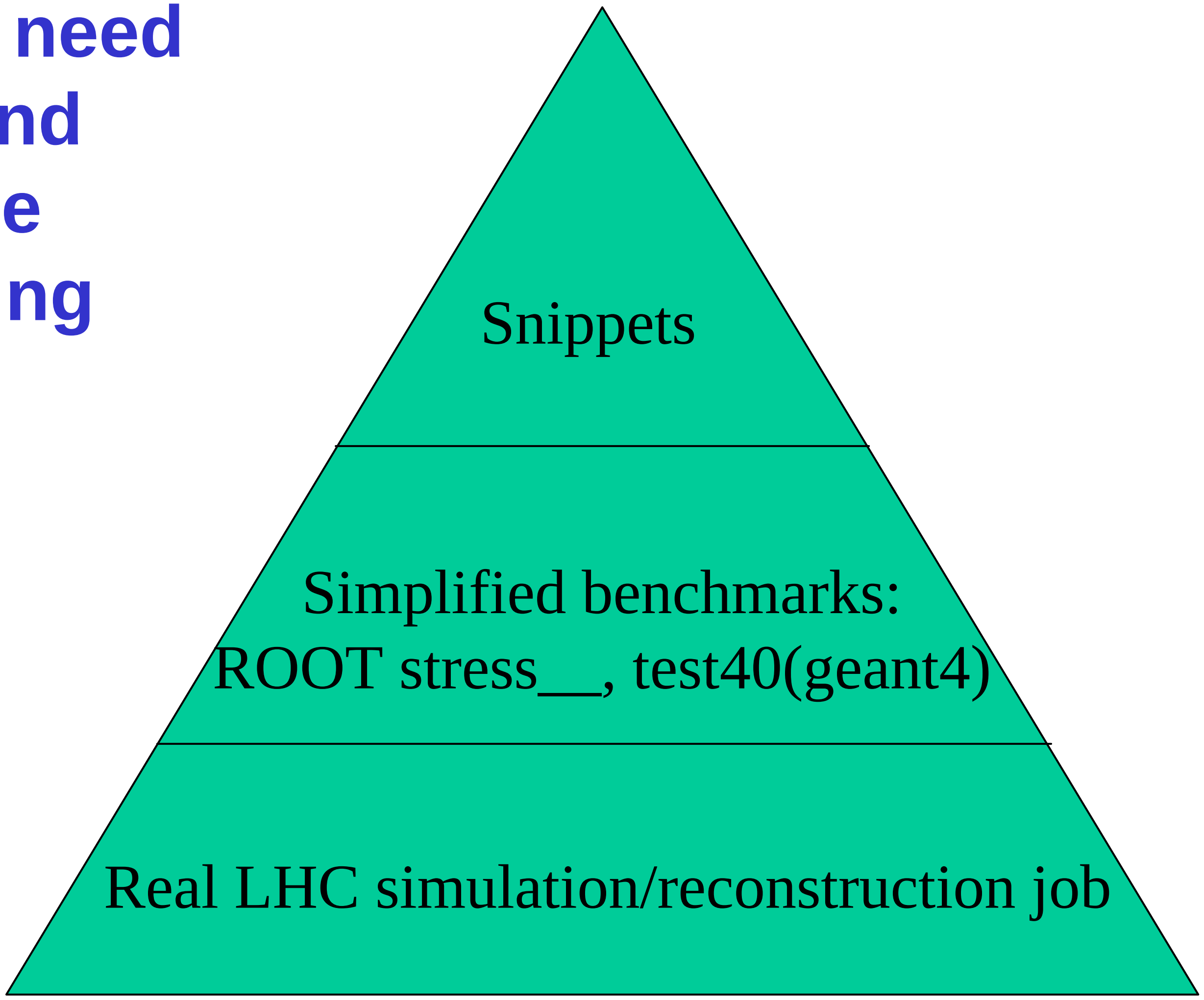
Second step: Source

- **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, …

The testing pyramid

- You always need to understand what you are benchmarking


Snippets
Simplified benchmarks:
ROOT stress__, test40(geant4)
Real LHC simulation/reconstruction job

Machine code

- It may be necessary 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
```

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

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