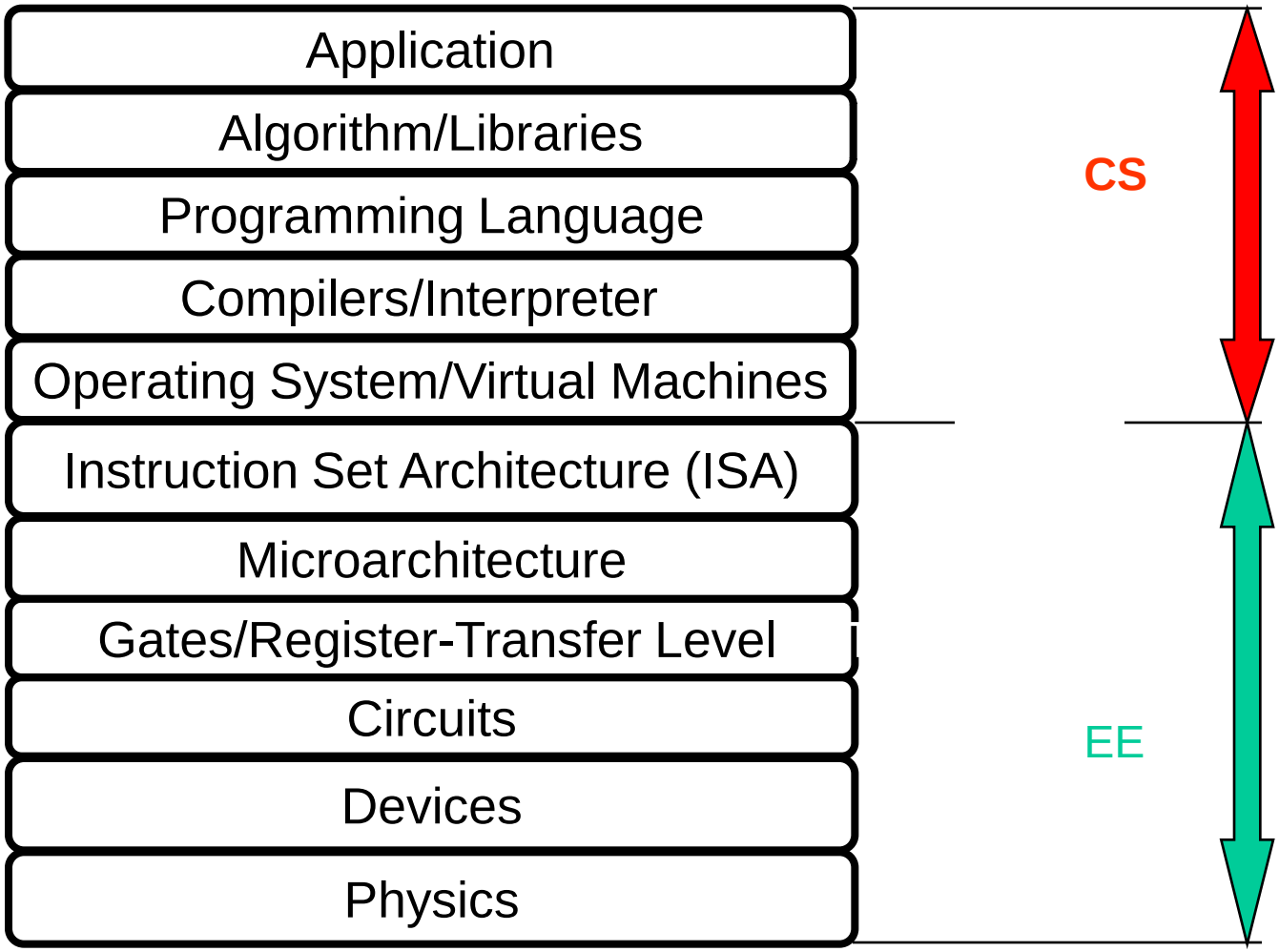
INTRODUCTION TO PERFORMANCE ANALYSIS

William Jalby, UVSQ,
Exascale Computing Research

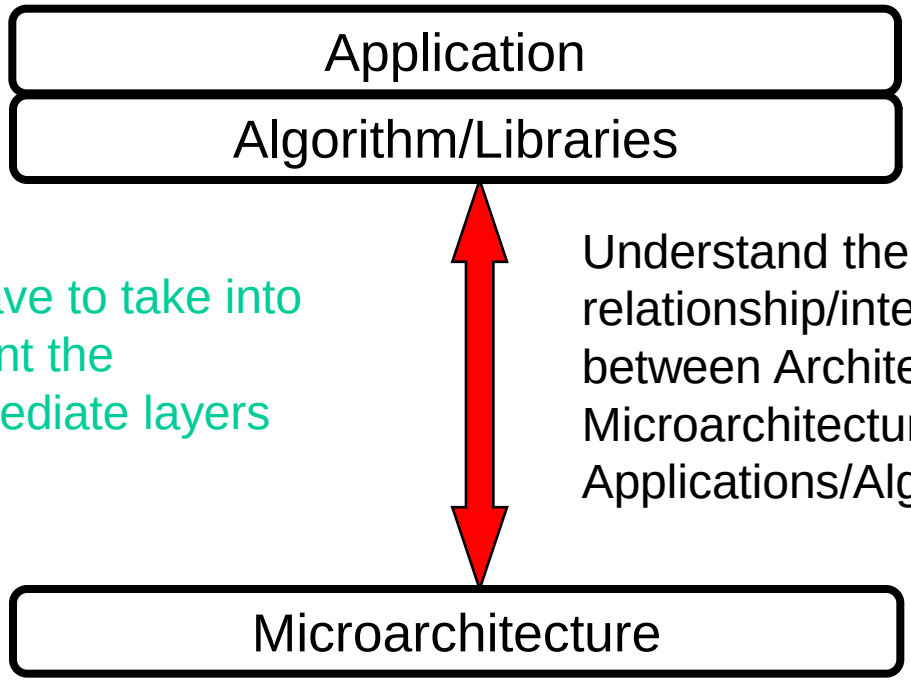
Overview

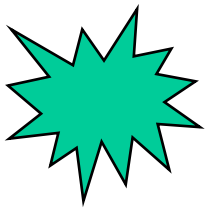
1. The stage/actors
2. Measurement Techniques
3. A brief microarchitecture overview
4. Microbenchmarking
5. DECAN

Abstraction Layers in Modern Systems



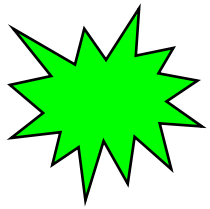
OUR OBJECTIVE/POSITIONNING





We have to take into account the intermediate layers

Understand the relationship/interaction between Architecture Microarchitecture and Applications/Algorithms



Don’t forget also the lowest layers

KEY TECHNOLOGIES:

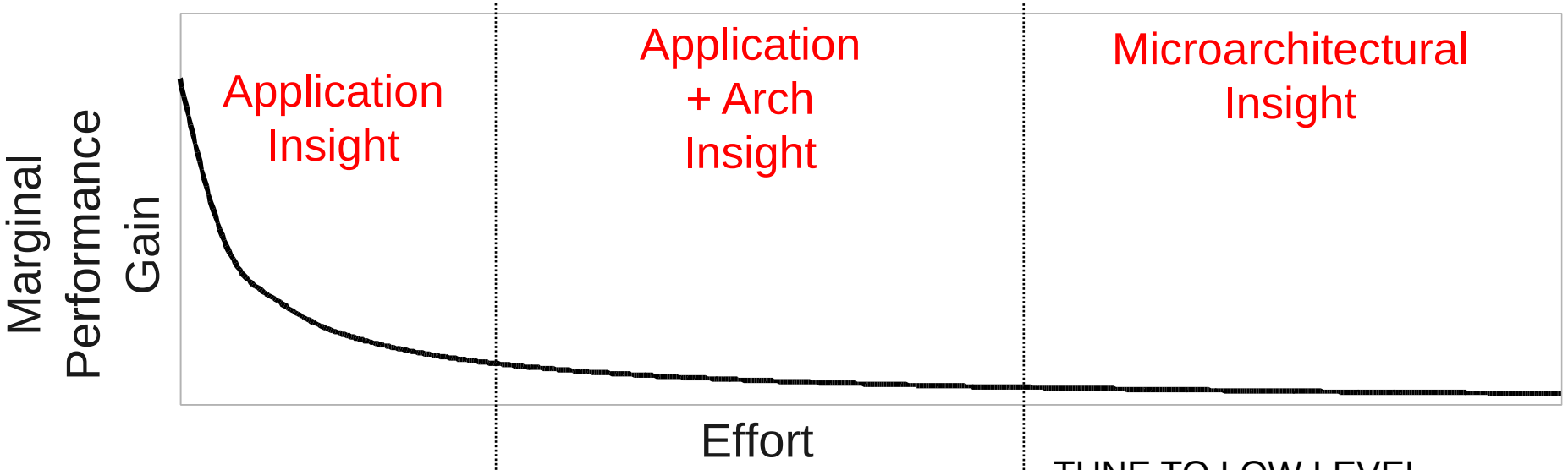
- Performance Measurement and Analysis

- Compilers

Standard goals for Performance Analysis

- For a given architecture and application, improve application performance: tune performance and/or change algorithms.
- For a given set of applications, try to determine best architecture including its variants (cache size, memory/core organization etc …)
- For Computing Center managers, optimize resource usage
- For hardware/system designers, understand bottlenecks on current architectures and derive guidelines for next generation
- NEW: For a given architecture and application, improve its energy consumption

performance tuning curve*



CHOOSE RIGHT ALGORITHM CLASS

Parallelism: 1000X

TUNE TO GLOBAL ARCHITECTURE CHARACTERISTICS

Optimize communication: 10X

Vectorize: 2X to 8X

TUNE TO LOW LEVEL ARCHITECTURE CHARACTERISTICS

Optimize cache usage: 2X to 10X

Optimize unicore execution: 1,2 X to 3X

*fruit-pickers, compilers, and dynamic optimizers all follow this model

Performance Tuning

- Identify clearly performance issues:
  - Where ?? source code fragment (ideally a few statements)
  - Who ?? algorithm, compiler, OS, hardware
  - How much ?? exact cost of performance issue (determine optimal possible performance for a given code fragment)
- Three solution techniques
  - Analytical models
  - Simulation
  - Measurements

Analytical Models

Mathematical equations describing system (or more likely subsystem) performance in function of key parameters

- ✚ Allows to exactly capture impact of parameters and ideal for performance tuning
- ✚ Fast
- ▬ Requires very strong simplifying assumptions to remain tractable/usable: low accuracy
- ▬ Has to validated/calibrated against simulation/experiment

- Exemples
  - ➢ Amdahl's law
  - ➢ L1/L2 equation: Tav = h T1 + (1-h) T2 h : Hit Ratio

Simulation

Software tool modeling hardware behavior of system or subsystem

- Explicit direct relation between hardware and software
- Slow: accuracy versus speed trade off (OS impact often not taken into account)
- Has to validated/calibrated against experiment
- To be accurate requires deep knowledge on target architecture

- Examples
  - Cache simulators: good tool to apprehend program temporal locality

Measurements

Direct measurement of running programs

- ✚ Excellent accuracy (if measurements done correctly): everything taken into account, no simplifying assumption: IDEAL
- ✚ Fast (not so fast if good measurement methodology is used)
- ▬ Difficult to vary parameters
- ▬ Difficult separate parameters impact (aggregate effect) s

- Examples
  - ➢ Analytical models built using measurement (microbenchmarks)

Metrics

- What can be measured:
  - Counts of a given hardware event occurrences: cache miss, instruction stalls, etc …
  - Time: time interval
  - Values: value profiling: stride of memory access, loop length, message size etc ….
- Difficulties:
  - Accuracy
  - Correlation with source code: aggregate values (otal number of cache misses for the whole loop not for individual statements)

TIME

- Wall clock time: it includes everything: I/O, system etc ….. Including other programs running simultaneously but it corresponds to response time
- CPU Time:
  - Time spent by CPU to execute programs
  - Real target
- How to measure time ?? recommendation use RDTSC: Read Time Stamp Counter (assembly instruction with good accuracy). However small durations (less than 100 cycles are extremely difficult to measure if not impossible)

Derived Metrics

- Rates: obtained by dividing number of occurrences by time
  - GIPS Billions of Instructions per second
  - GFLOPS Billions of Floating point instructions per second
  - MBYTE/s number of Mbytes per second (useful for characterizing stress on various memory levels)
  - THROUGHPUT: how many job instances executed per second
- Rates are useful to assess how well some hardware parts are used.
- A useful derived metric: SPEEDUP: T1/Tp Where T1 (resp. Tp execution time on 1 (resp. p) core(s).

How to perform measurements ??

- How to trigger measurements ??
  - Hardware Driven: sampling
  - Code Driven: tracing
- For tracing, how to insert probes ??
  - Source level
  - Binary level
  - Static/dynamic instrumentation
- Three key questions:
  - How much perturbation is introduced ??
  - How to correlate with source ??
  - How to Record/Display information??

Sampling (1)

- OPERATION MODE (hardware driven):
  1. Focus on a given hardware event: clock ticks, FP operations, cache miss,
  2. At each event occurrence, counter is incremented
  3. When threshold is reached (counter overflow), interrupt occurs and counter reset to 0
- What happens on interrupt ??
  - ➢ Record instruction pointer and charge the whole occurrences count to that IP
  - ➢ Advanced mechanism on INTEL processors: PEBS (Precise Event Based Sampling): record processor state (register values etc …)

Sampling (2)

KEY PRINCIPLE: general statistical measurement techniques relying on the assumption that a subset of the population being monitored is representative of the whole population

- CORRELATION WITH SOURCE CODE:
  - Function level, Basic Block Level, Loop level but NOT AT THE INSTRUCTION LEVEL (reasonably)
  - IP is not enough, whole call stack is needed which is not easy ☺
  - Inclusive Versus Exclusive issue
  - Call site issue

  EXCELLENT EXAMPLE: XE Amplifier (VTUNE/PTU) : INTEL

Inclusive versus Exclusive

Subroutine toto1 (.....)

Basic Block 1 (BB1)

Call toto2

Basic Block 2 (BB2)

Return

Toto2 is leaf in the call graph

INCLUSIVE TIME:

Tinc = T(BB1) + T(toto2) + T(BB2)

EXCLUSIVE TIME

Texc = T(BB1) + T(BB2)

Exclusive time is easy but Inclusive time needs call stack

Issue with call sites

Subroutine toto1

……

call toto2 (4)

…….

call toto2 (10000)

……

Return

Usually, all of the counts relative to the different occurrences of toto2 will be lumped together: bad correlation with source code.

TRICK: use toto2short and toto2long to distinguish the two!!

SAMPLING: pros and cons

PROS

- Binary used as is (no recompile/no modifications)
- User transparent
- Low overhead if sampling period is large
- PEBS offers very interesting opportunities (whole processor state)

CONS

- Accuracy
- Correlation with source code
- Difficult to assert its quality

TRACING

- OPERATION MODE (code driven):
    1. Insert probes (source/binary, static/binary) at point of interest (POI)
    2. Measurement performed when probe is executed
    3. Record tracing event/build trace
- Trace format
    - VTF : used by TAU
    - OTF: Open Trace format

Instrumentation: Probe Insertion

- Source level: EXAMPLE: TAU source code instrumenter
- Library level
- Binary level: EXAMPLE: MAQAO/MIL
- Probe Insertion
  - Manual: tedious, error prone
  - Automatic: preprocessor, binary rewrite: Might be difficult to select meaningful POI.
  - Automatic by compiler: specification can be done at source level but instrumentation done by compiler: INTEL IFC/ICC 12.0

Source Instrumentation Issue

```
DO I = 1, 200
   DO J = 1, 1000
      ……
   ENDDO
ENDDO
```

Loop Interchange can be performed by compiler

```
DO I = 1, 200
   Start Clock
   DO J = 1, 1000
      ……
   ENDDO
   Stop Clock
ENDDO
```

Loop interchange no longer possible!!

Source Instrumentation: Pros and Cons

PROS

- Portable
- Good correlation with source code

CONS

- Needs recompile
- Interaction with compiler
- Difficult interaction with high level abstractions (C++)
- Requires access to source code

Binary Instrumentation: Pros and Cons

PROS

- No recompile
- Instrument the real target code
- No need to access source code
- Lowest overhead possible
- OK correlation with simple source code constructs.

CONS

- Not portable
- Need access to specialized tooling (disassembler)
- Might be difficult to correlate with High Level abstractions in source code (C++)

Tracing: pros and cons

PROS

- Excellent correlation with source code
- Excellent accuracy
- Traces preserve temporal and spatial relationships between events
- Allows reconstruction of dynamic behavior
- Most general technique

CONS

- Traces can be huge
- How to select POI and events to be measured a priori ??
- Writing large trace files can induce measurement perturbation
- Aggregate view at loop level at best

A simplified view at X86 architecture

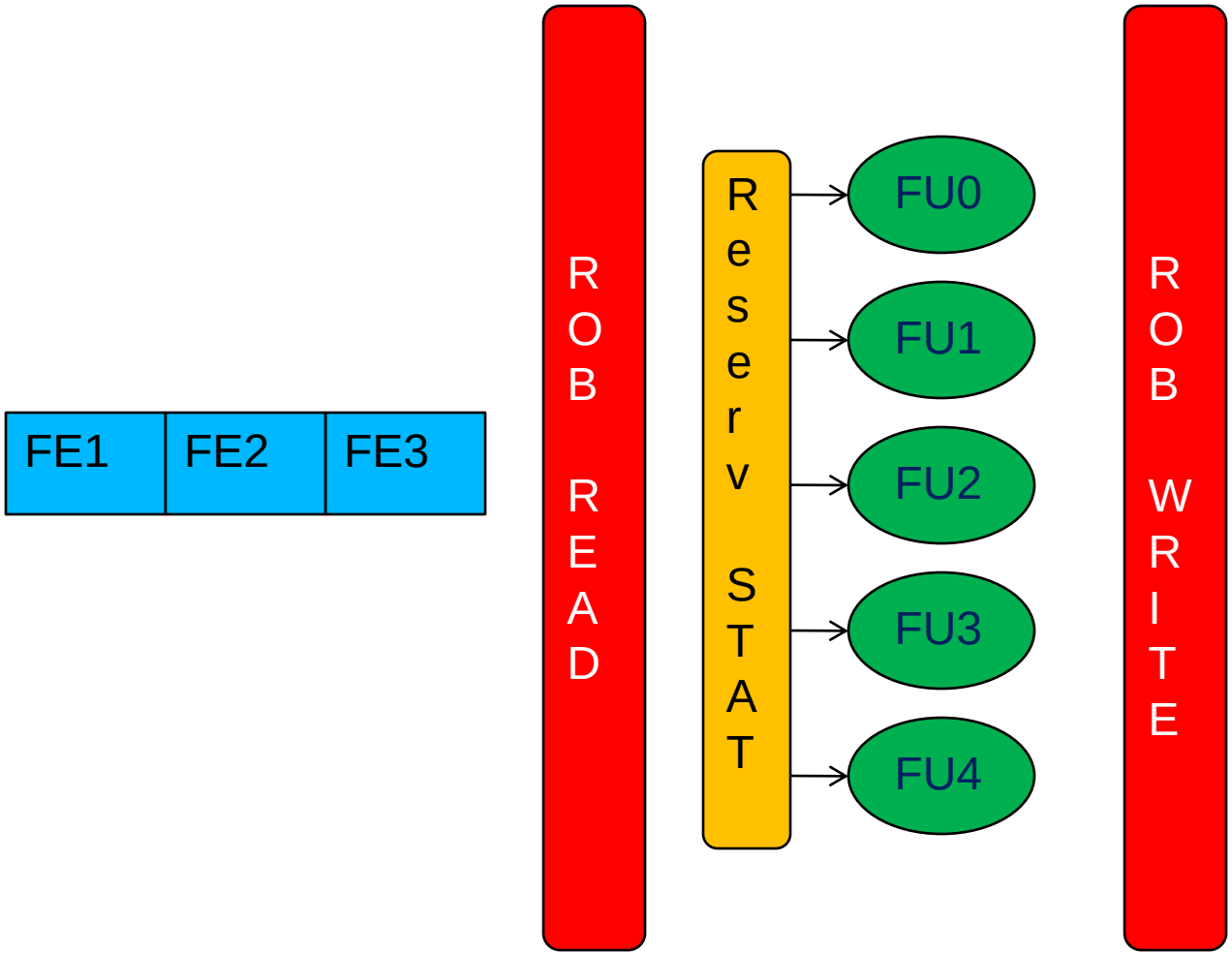
Three key components

- Front End pipeline: prepares instructions for execution; IN ORDER
- Execution pipeline: OUT OF ORDER
- Completion pipeline: retires completed instructions: IN ORDER
- Library level

These three components coupled through buffers:

- REORDER BUFFER (ROB): keep track of instruction status
- RESERVATION STATION (RS): store instructions ready to execute
- MEMORY ORDER BUFFER: make sure memory instructions are executed in an OK order.

Simplified view of X86 Pipeline



Typical Front End Pipeline

- Branch Prediction
- Instruction Fetch (16 B/cycle): fetches instructions
- Predecode (6 instruc /cycles): find instruction boundaries
- Decode (4 instruct/cycle): indentify operands; operations, generate micro operations
- Register allocation/rename (4 micro/cycle)
- Read Operands (4 micro/cycle)
- Wait for operands from ROB
- Get into reservation station

INSTRUCTIONS FLOW ALL OF THESE STAGES IN ORDER

Execution Units

Functional units are grouped into clusters with PORT as entry points

- PORT0: ALU Operation/SSE FMUL
- PORT1: ALU Operation/SSE FADDD
- PORT5: ALU Operation/Branch
- PORT2: Loads
- PORT3: Store Address
- PORT4: Store Data

In general ports can accept one new instruction every cycle: max of 6 instructions can be issued every cycle

OUT OF ORDER EXECUTION: all dependencies have been resolved earlier except between memory address (MOB)

Between 50 and 100 instructions simultaneously in flight

Completion/Retirement

- Once an instruction finishes its execution, results are provide to the ROB so subsequent instructions can use directly these results
- Write back to register file
- Retire instructions in order (4 / cycle)

STRANGE EFFECT: a long latency instruction (divide) can induce a quickly fill up of the ROB and freeze pipeline

Analysis of Out of Order

- Simplified version of Little's Law: Operation Latency = L cycles, L operations have to be in flight for ensuring a sustained rate of 1 operation per cycle.
- Out of Order will buy you a few cycles (at most 10 or 20) not hundreds of cycle of main memory latency.

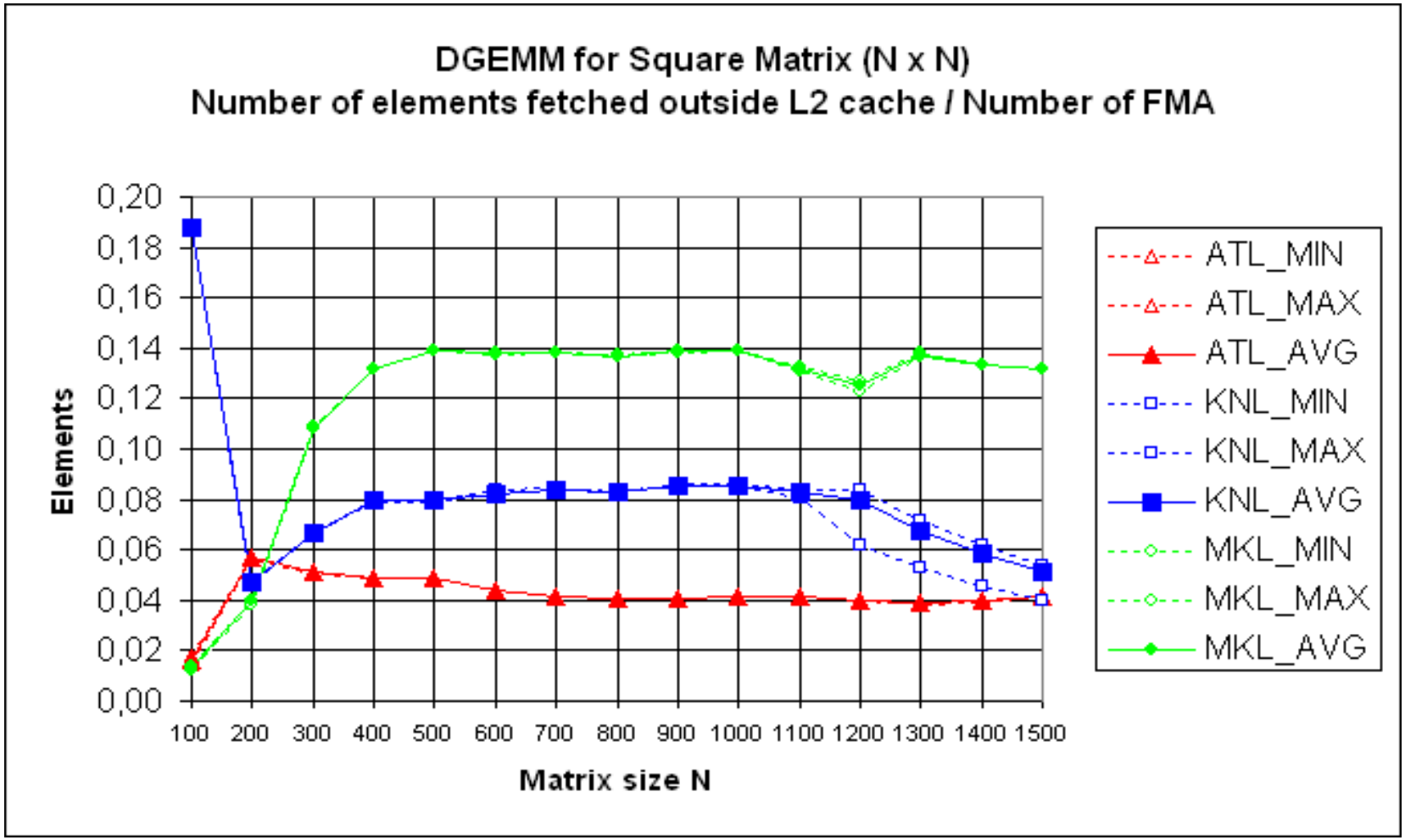
Hardware Performance Counters/Events

- A large number of hardware events (around 1200 on Nehalem processors) can be counted
- BUT DURING A SINGLE RUN, only 4 to 6 counters are available
- Therefore multiple runs are necessary to gather a good set of events
- Multiplexing can increase number of events monitored but at accuracy expense ☺
- Very precise
- Some nice feature: count number of loads exceeding a given latency threshold
- REAL GOAL: hardware debugging. SECONDARY GOAL: understand machine behavior

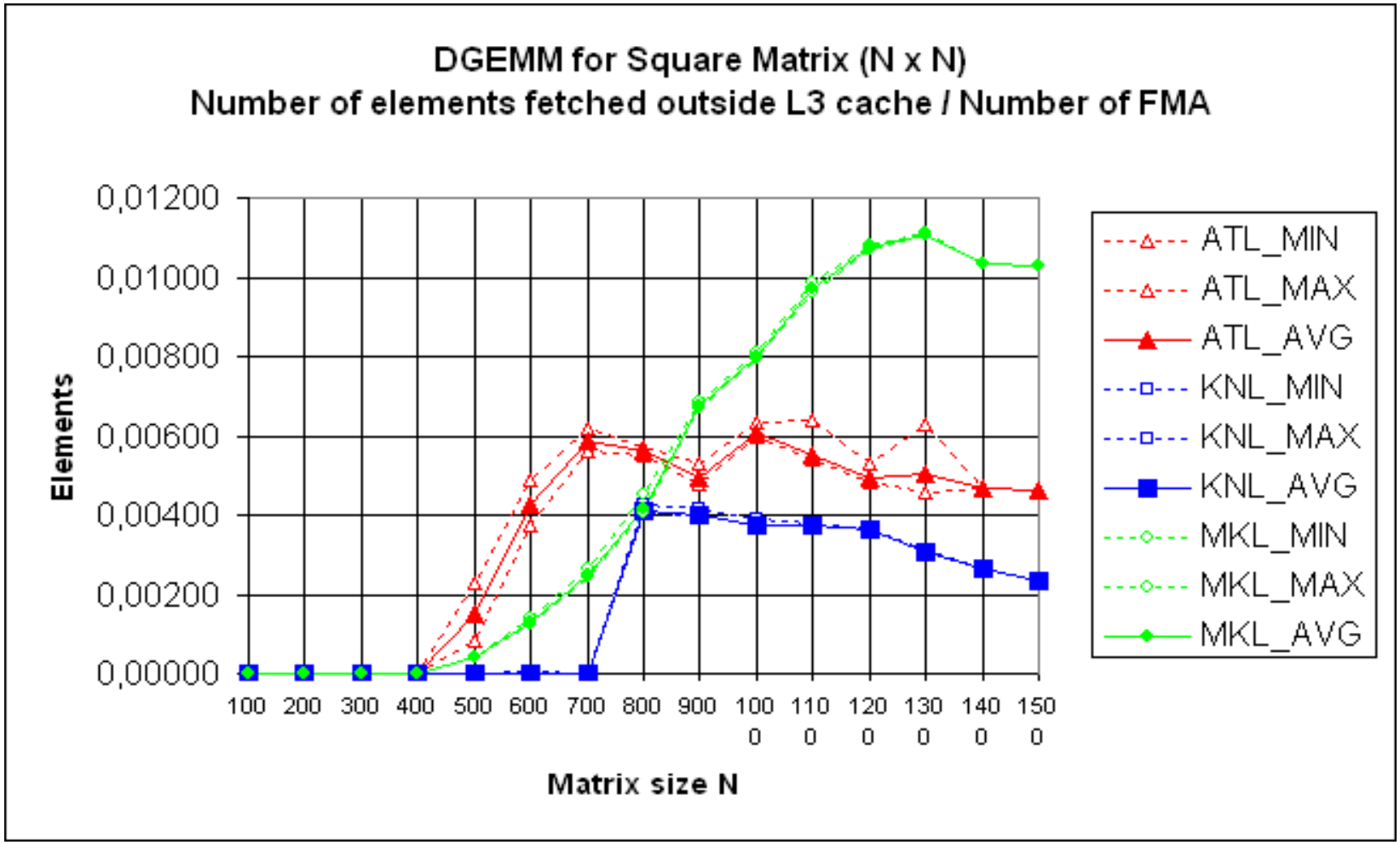
Critics on hardware performance events

- TOO LOW LEVEL: very local view at the hardware level
- NEEDS A DEEP UNDERSTANDING OF MICROARCHITECTURE: no good documentation available on microarchitecture
- CHANGE FROM ONE PROC GENERATION TO THE NEXT: different names designate similar events, same names designate different events
- NEED TO KNOW WHAT TO MONITOR: with 1200 events task is not easy
- HARD TO QUANTIFY: what is high ??
- ALMOST IMPOSSIBLE TO ACCURATELYCORRELATE WITH SOURCE CODE

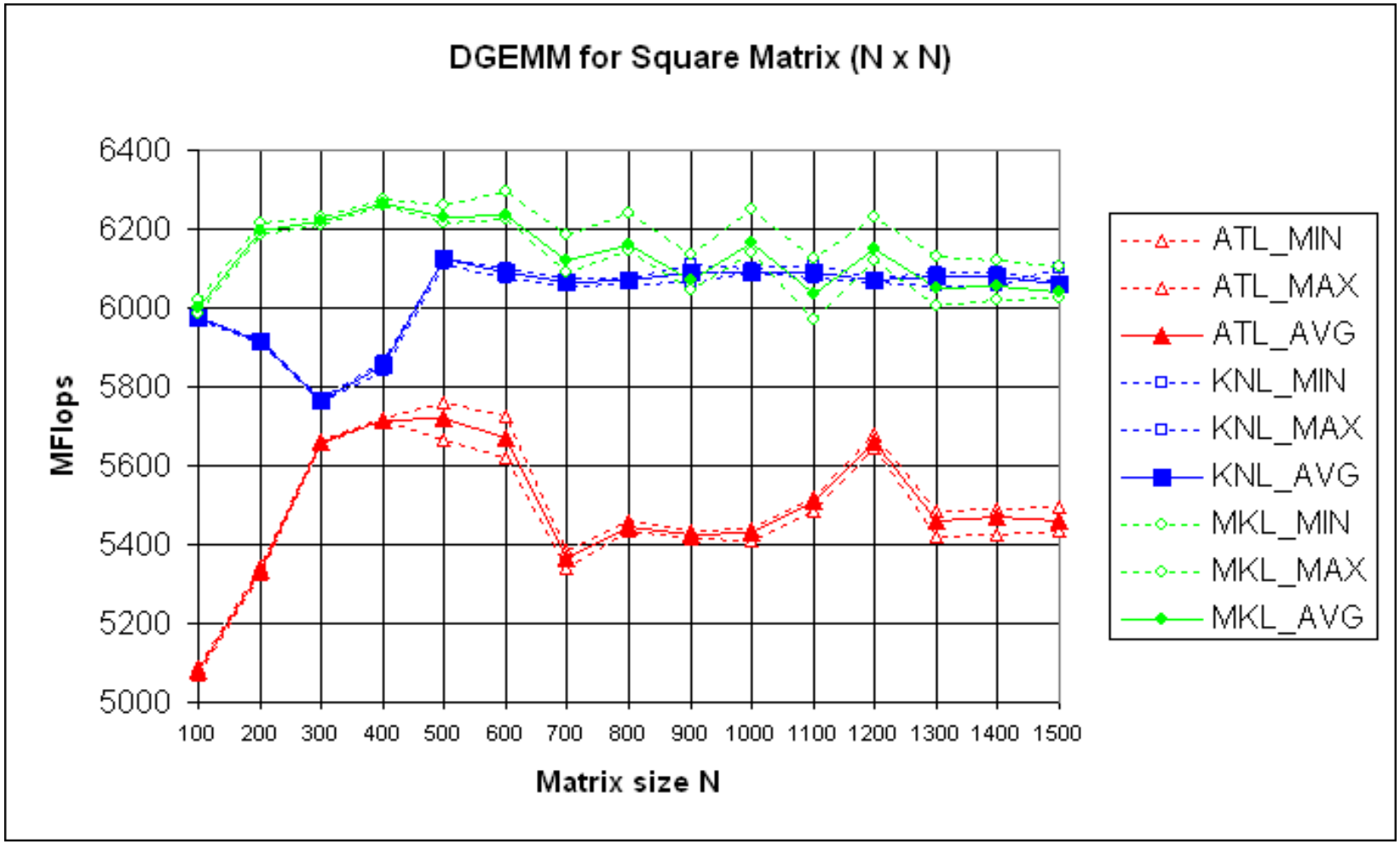
(NxN)(NxN) DGEMM L2 Behavior



DGEMM (NxN) (NxN) L3 Behavior



(NxN) (NxN) DGEMM Performance



THE 4 KEY ROADBLOCKS

- Algorithm
- Compiler (MAQAO on Wednesday)
- OS (Today with S. Valat)
- Hardware (next)