ESC 2011 - Bertinoro

Understanding Low-level Performance Tuning

Sverre Jarp, Andrzej Nowak

CERN openlab

October 28th 2011

Contents

1. Rationale and background
2. Software performance tuning on a modern PC
3. Drilling down on performance figures
4. Practical tips
5. Tools

- In this talk, we focus on processor performance
- We focus on Intel Core processors, but the same techniques can often be applied to other case

CERN openlab

Rationale and background

Seven dimensions of performance

First three dimensions:

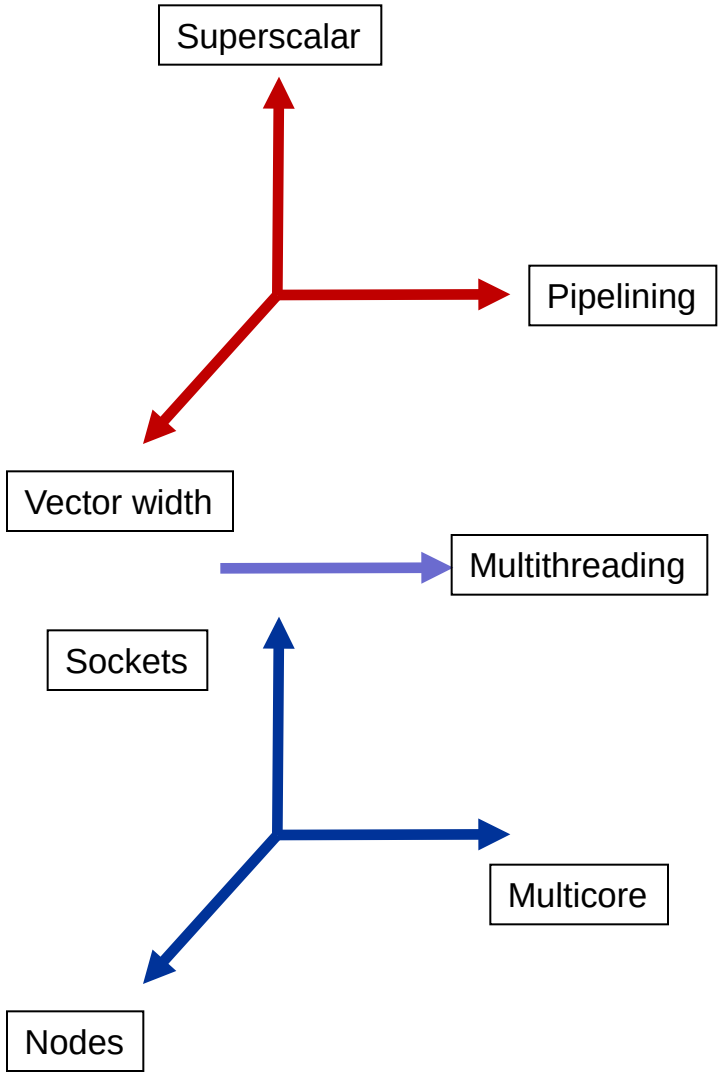
- Hardware vectors/SIMD
- Pipelining
- Superscalar

Next dimension is a “pseudo” dimension:

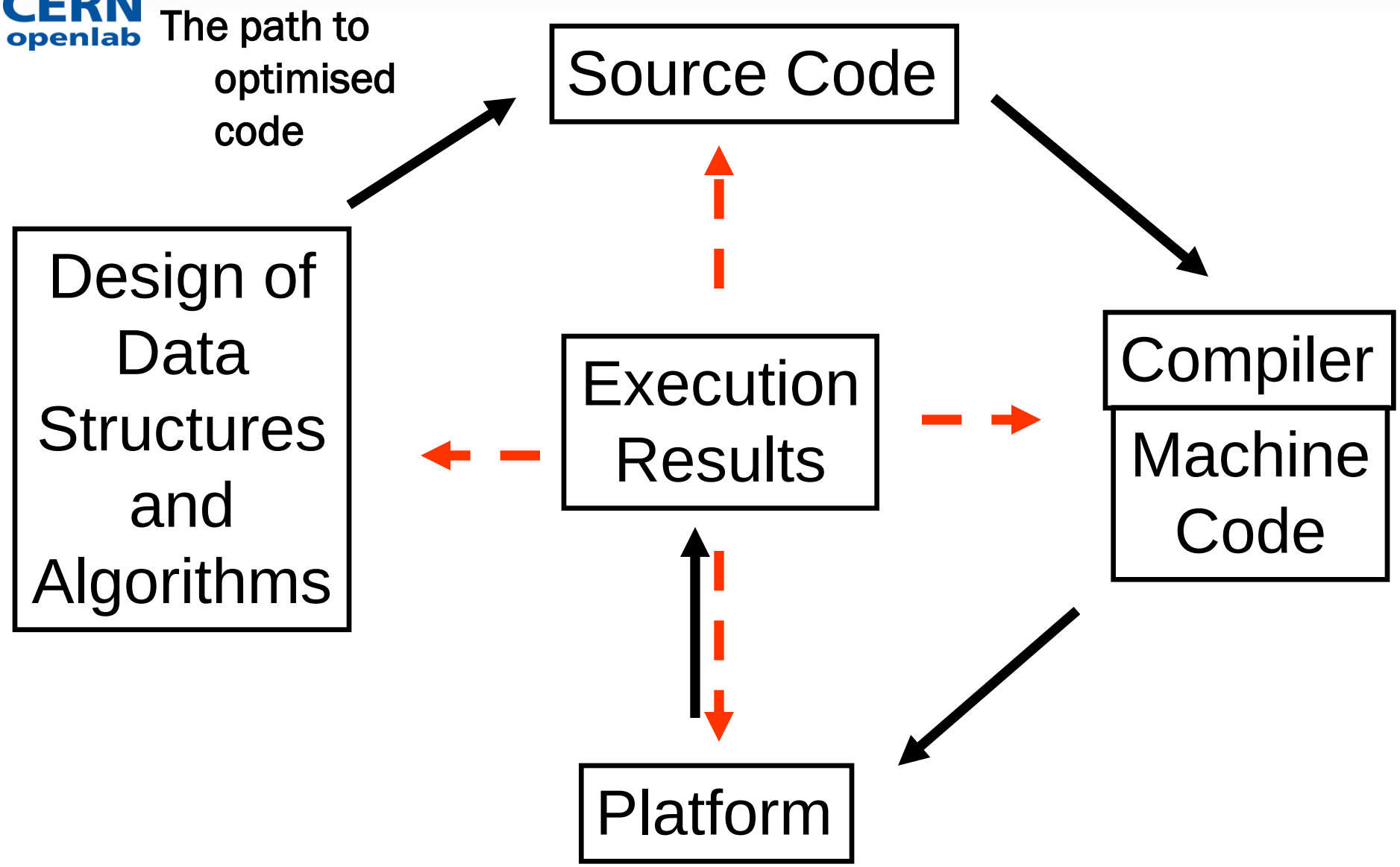
- Hardware multithreading

Last three dimensions:

- Multiple cores
- Multiple sockets
- Multiple compute nodes



An iterative approach



Performance tuning levels – reality check

Level	Potential gains	Estimate
Algorithm	Major	~10x-1000x
Source code	Medium	~1x-10x
Compiler level	Medium-Low	~10%-20% (more possible with autovec or parallelization)
Operating system	Low	~5-20%
Hardware	Medium	~10%-30%

There is a facility that can monitor all of the items above and their interaction – hardware counters

CERN openlab

Performance tuning on a modern PC

Key techniques

Measuring performance

- **The most common performance measurement unit is time**
  - Wall clock time – "how long do I have to wait for it to be done?"
  - CPU time – "for how long is the computer busy?"
  - Latency – "how long do I have to wait to get an answer?"
  - Throughput – "how much of X in a period of time?"

- **Minimizing time/latency is not the same as maximizing throughput**
  - and vice versa – i.e. see Amdahl's and Gustafson's laws

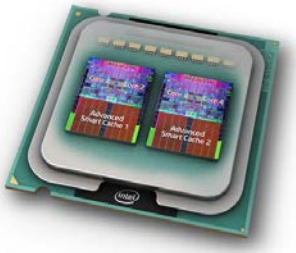
Performance monitoring in hardware

- Most modern CPUs are able to provide real-time statistics concerning executed instructions…
- …via a Performance Monitoring Unit (PMU)
- The PMU is watching your application in real time! (and everything else that goes on inside the CPU)
- Limited number of sentries (counters) available, but they are versatile
- Counters monitor events as they happen
- Recorded occurrences are called samples
- Core i7 and other (Nehalem and later):
  - 2-4 universal counters: #0, #1, (#2, #3)
  - 3 specialized counters: #16, #17, #18
  - Additional 8 uncore counters: #20-#27

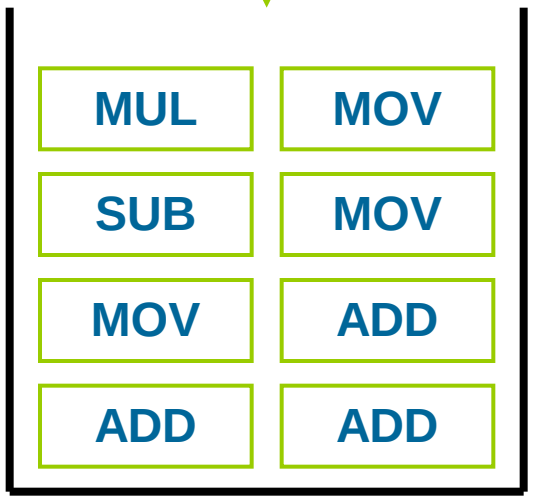
Events

- **Many events in the CPU can be monitored**
  - A comprehensive list is dependent on the CPU and can be extracted from the manufacturer's manuals or from relevant tools
  - Examples: cache misses, instructions executed, cycles, loads, vector operations
- **On some CPUs (i.e. Intel Core), some events have bit-masks which limit their range, called "unit masks" or "umasks"**
  - Example: memory instructions retired: "ALL" or "only LOAD" or "only STORE"
- **Extensive information: Intel Manual 248966-023a**
  - Intel Manual 248966-023a "Intel 64 and IA-32 Architectures Optimization Reference Manual"
- **AMD CPU-specific manuals**
  - i.e. "BIOS and Kernel Developer's Guide for AMD Family 10h Processors" or "Software Optimization Guide For AMD Family 10h and 12h Processors"

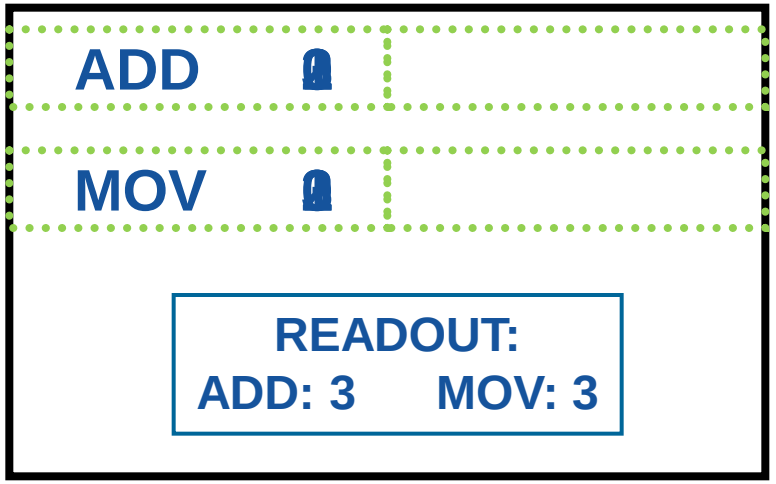
The Performance Monitoring Unit



Let's monitor ADD and MOV instructions



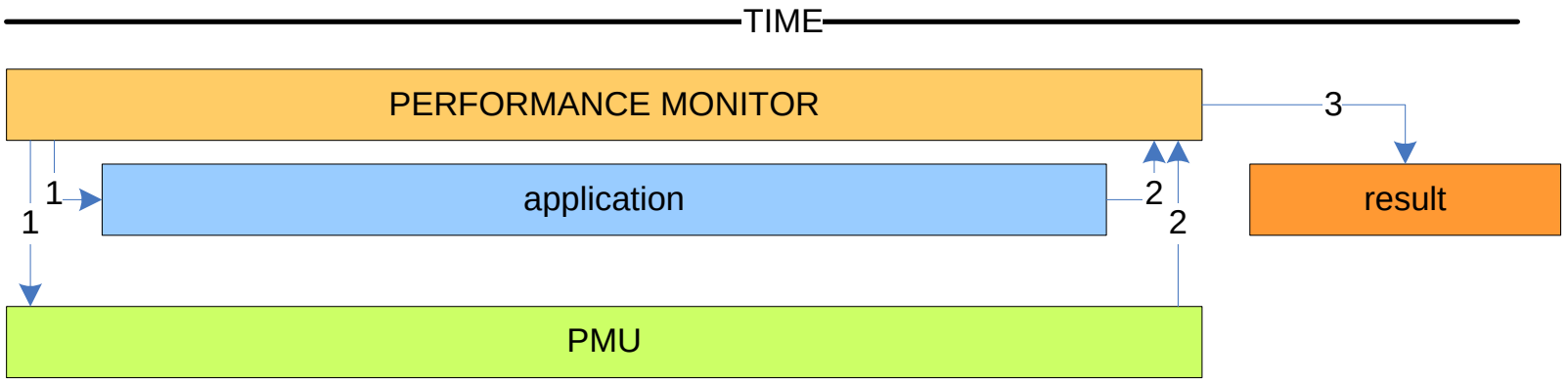
RETIRED INSTRUCTIONS (=successful & useful execution)



PERFORMANCE MONITORING UNIT (PMU)

Popular methods for performance monitoring: Counting

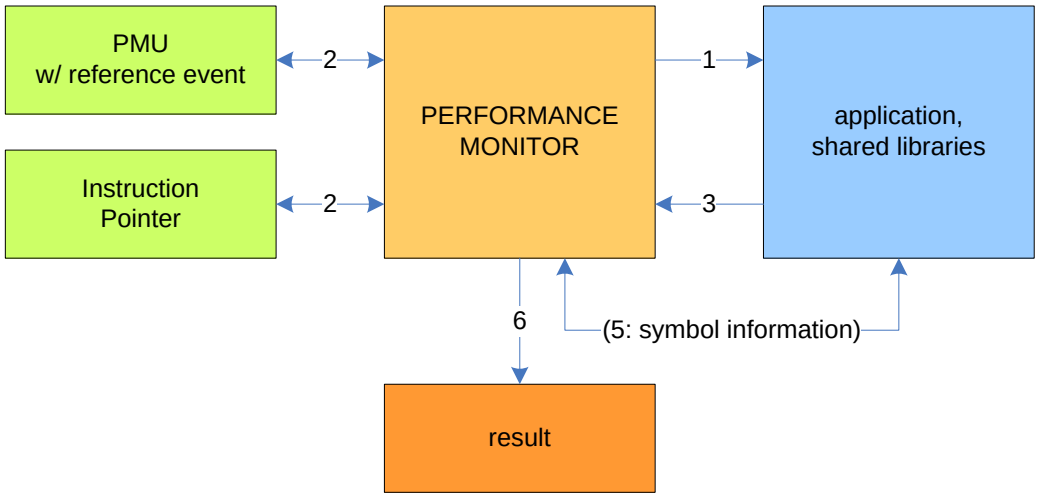
1. Programming the PMU with the specified event(s)
2. Reading the elapsed counts
3. Producing the result



Popular methods for performance monitoring: Sampling (a.k.a. “Profiling”)

1. **The PMU is programmed with the event(s) of interest**
2. **The application is started**
3. **Hardware performance counters count in the background**
   - When a pre-programmed value is reached, a performance monitoring interrupt is sent
   - The interrupt handler can perform a variety of operations, but most of the time it will try to register the state of the machine, especially the instruction pointer – this is called a sample

4. **Finalization**
   - Once the application finishes, the performance monitoring software consolidates and renders the results. There are different ways to present the captured data, and they also depend on the type of counters used.



Popular methods for performance monitoring: Other useful mechanisms

> Event multiplexing

- Since the amount of simultaneously monitored events is limited, a technique called “multiplexing” can be used. It consists of frequent, periodical time-based switching of the monitored events on the counters. Final results are produced through extrapolation, and the choice of events in each group is important for accuracy.

> System wide profiling vs. application profiling

- Some tools monitor the whole operating system and later allow you to restrict the results just to your application or library. Some tools (like pfmon) have built in context switching logic which allows for direct monitoring of applications.

Popular methods for performance monitoring: Other useful mechanisms

> Instrumented vs. non-instrumented monitoring

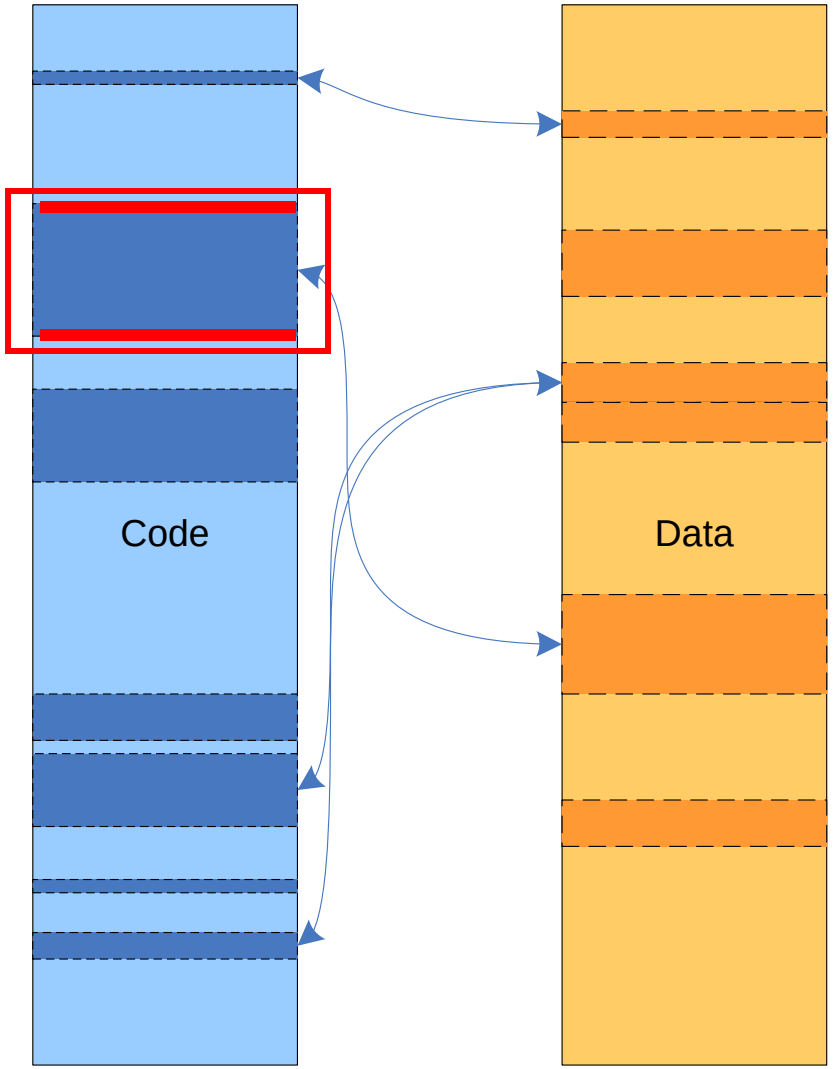
- Some analysis types require binary instrumentation. That means that the monitoring tool inserts probes into the binary code and changes the way the monitored application is running. Such activities might slow down the application by orders of magnitude!
- Other analysis types don't require binary instrumentations and can be performed in the background without disturbing the application. The performance penalty is usually minor, in the order of 1-2%.

> User level vs. kernel level

- Most tools will allow you to choose whether you wish to monitor on the user level or the kernel level (e.g. pfmon). Kernel level monitoring is very handy to debug drivers or system call abuse. You can get a profile and counts just as you would with a userland application.

Popular methods for performance monitoring: Triggers and triggering

- **Automatically start or stop monitoring**
- **Trigger types:**
  - Code
  - Data
- **A symbol name...**
  - i.e. “foobar”
- **...or an address**
  - i.e. 0x8103b91e



CERN openlab

Common performance figures

And how to interpret them

Basic information about your program: Recap

- **The amount of:**
  - instructions executed
  - processor cycles spent on the program
  - transactions on the bus
- **The amount/percentage of:**
  - memory loads and stores
  - floating point operations
  - vector operations (SIMD)
  - branch instructions
  - cache misses

Advanced information about your program

- **The amount and type of:**
  - micro-ops executed
  - SIMD instructions executed (and the kind)
  - resource stalls within the CPU

- **Cache access characteristics**
  - A rich set on Intel Core CPUs
  - Requests (missed / hit / total / exclusive or shared / store or read)
  - Lines modified / evicted / prefetched

Derived events

> Too much information available?

- Low level and fine grained events can be combined to produce ratios (so called "derived events")

> Examples (discussion follows):

- Cycles per instructions [CPI]
- Cache miss ratio or impact
- Branch misprediction ratio
- Modified data sharing ratio
- Percentage of wasted cycles
- Bus occupancy

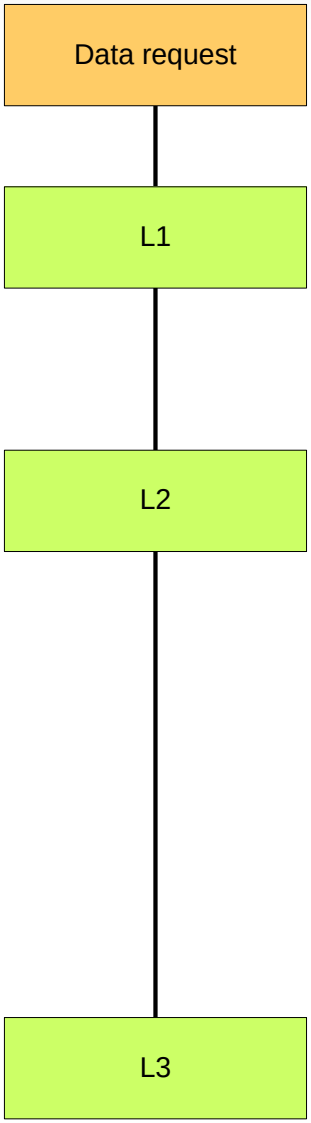
A word for the future

Mapping performance monitoring data onto your source code and environment requires care and experience

Cache misses

- **If the requested item is not in the polled cache, a higher level has to be consulted (cache miss)**
- **Significant impact on performance**
  - Memory access issues are very common, yet hard to fix
- **Ratio:**

  LAST LEVEL CACHE MISSES / LAST LEVEL CACHE REFERENCES
- **Tips:**
  - A last level cache hit ratio below 95% is considered to be catastrophic!
  - Usually the figure should be above 99%
  - The overall cache miss rate might be low (misses / total instructions), but the resource stalls figure might be high; always check the cache miss percentage

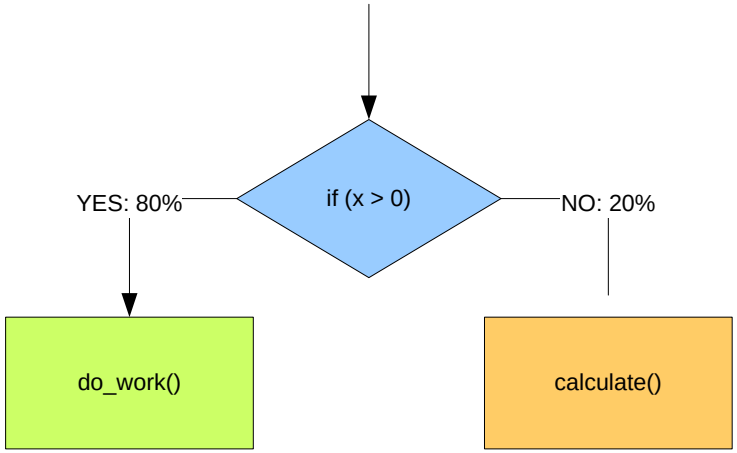


Branch prediction

- **Branch prediction** is a process inside the CPU which determines whether a conditional branch in the program is anticipated by the hardware to be taken or not
- Typically: prediction based on history
- The effectiveness of this hardware mechanism heavily depends on the way the software is written
- The penalty for a mispredicted branch is usually severe (the pipelines inside the CPU get flushed and execution stalls for a while)

Branch prediction ratios

- **The percentage of branch instructions**
  BRANCH INSTRUCTIONS / ALL INSTRUCTIONS
- **The percentage of mispredicted branches**
  MISPREDICTED BRANCHES / BRANCH INSTRUCTIONS
  - The number of incorrectly predicted branches is typically rather low: 20%, 10%, 5%, ... (?)



Floating point operations

- **Often a significant portion of work of an application**
- **May be accelerated using AVX or SSE (SIMD)**
- **Related events on the Intel Core microarchitecture:**
  - "traditional" x87 FP ops
  - Packed/Scalar single computational SIMD
  - Packed/Scalar double computational SIMD
  - SIMD micro-ops in total
- **Non-computational SIMD instructions can also be counted**

CERN openlab

Performance tuning tools

An update on popular performance monitoring facilities

Popular performance tuning software (1)

- **Linux "perf" – the new performance monitoring subsystem in the Linux kernel**
  - Pros:
    - Low level access to some counters
    - No patching needed for the application nor the kernel
    - Growing community and toolset
  - Cons:
    - A new (and somewhat crude) implementation that still needs time
    - Initial version: dangerously oversimplified
- **perfmon2 – a powerful legacy performance monitoring subsystem for Linux**
  - Pros:
    - Low level access to all counters
    - No recompilation needed for the application
    - Well established toolset
  - Cons:
    - Recompiled kernel or kernel patch needed
    - Development slowed down because of "perf", shrinking community

Popular performance tuning software (2)

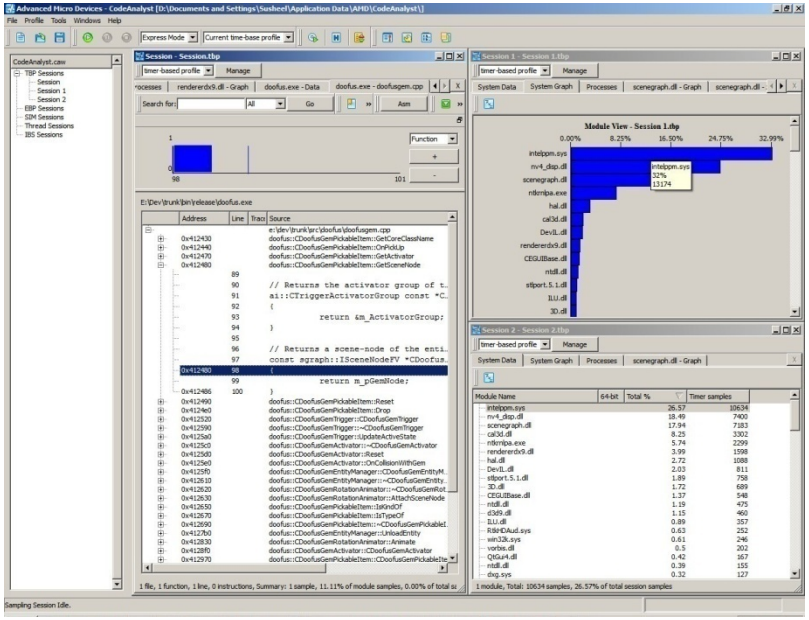
- **lgprof – Covered by Lassi earlier in the week**
- **gprof – flat profiler**
  - Recompilation needed
- **oprofile – flat profiler and event based sampling**
  - Flat profiles
  - Antique; Kernel driver needed
- **Valgrind**
  - Synthetic software CPU
  - Simulates cache misses and branch mispredictions, memory space profiler, function call relationships
- **Intel tools - new redesigned tools, native to Linux**
  - “Inspector” - Memory and threading inspector (extended functionality of Thread Checker)
  - “VTune Amplifier” - Performance inspector (extended functionality of VTune and Thread Profiler)

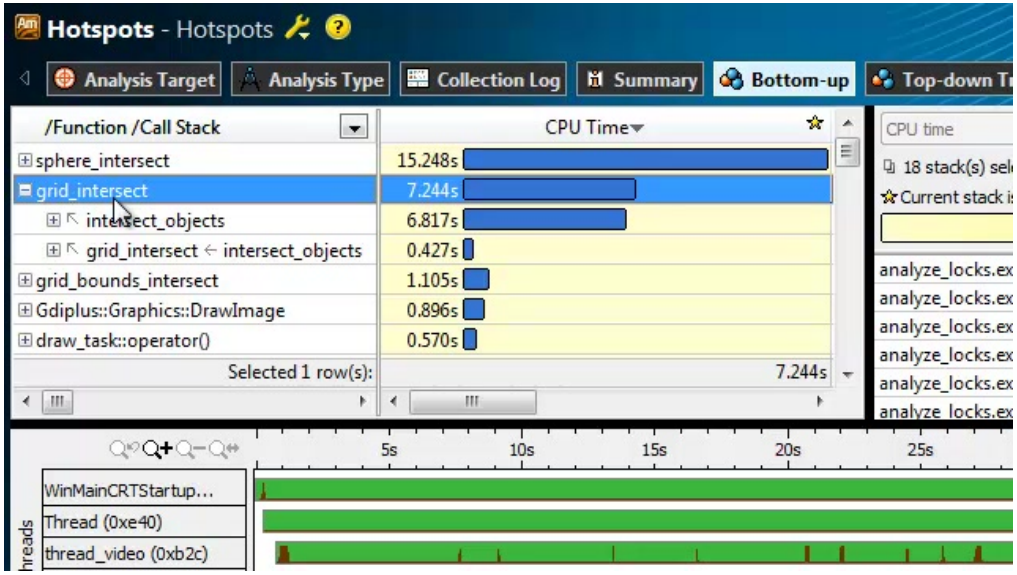
Popular performance tuning software (3)

> Intel products (soon available at CERN):

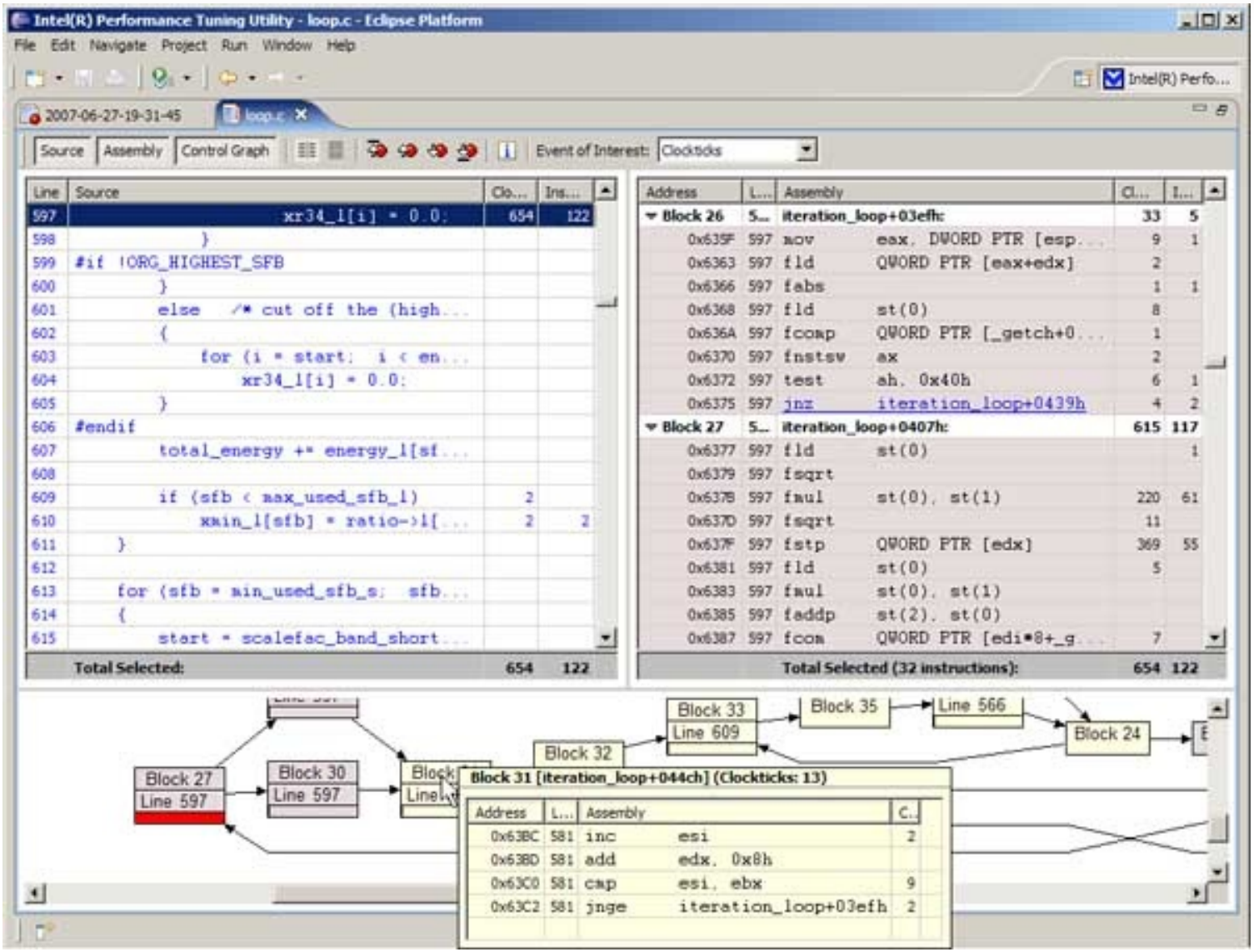
- VTune Amplifier (image on right), Inspector
- PTU (Performance Tuning Utility; next slide)
- Thread Profiler (legacy)

> AMD CodeAnalyst (image on left)





Popular performance tuning software (4)



CERN openlab

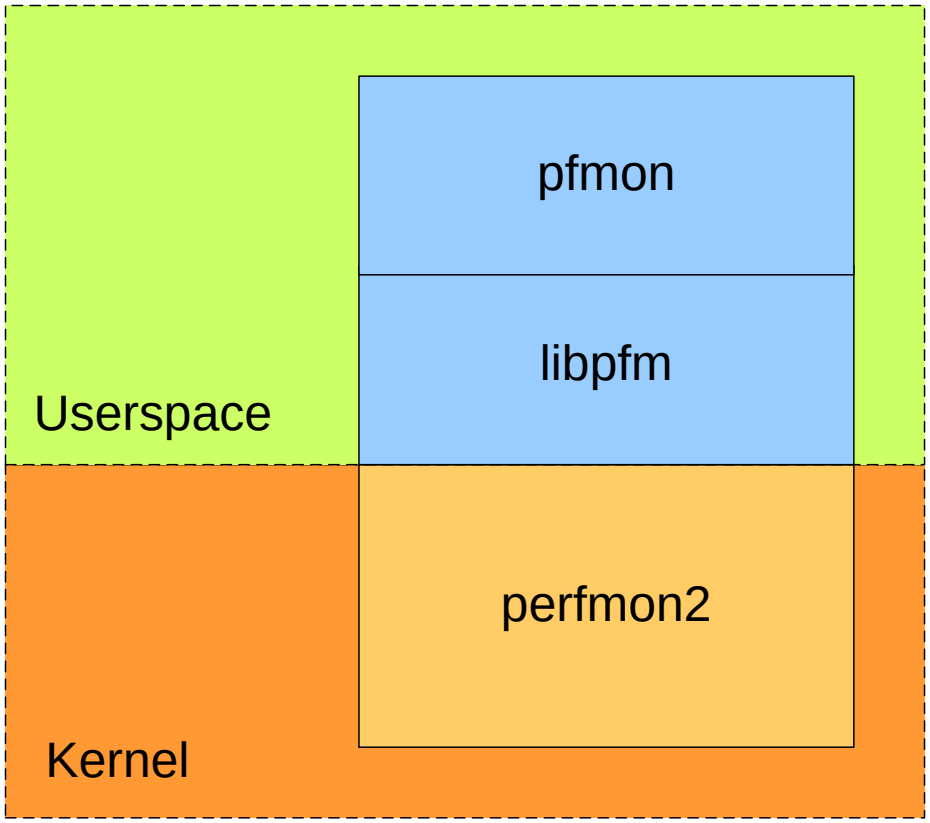
Perfmon2 & pfmon

A real-world performance monitoring framework example

... and a demo

Perfmon2 architecture

- We use it as an example of a robust performance monitoring framework for Linux
- perfmon2 – kernel part
- libpfm – userspace interface for perfmon
- pfmon – “example” userspace application, perfmon2 client



Perfmon2 – description and rationale

> Resides in the kernel

- Available as a kernel patch
- Very basic functionality

> Why was it chosen?

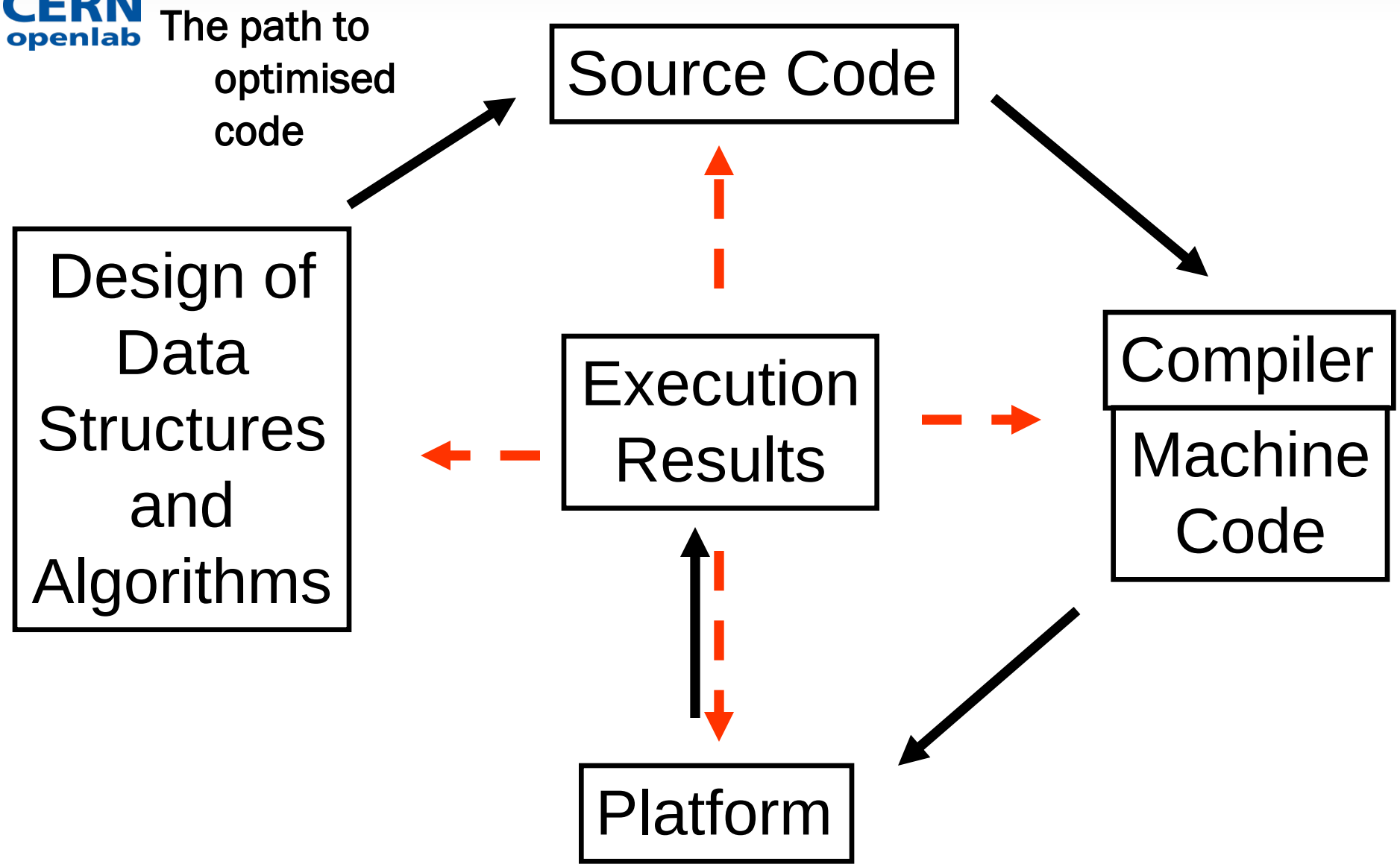
- Simple to use, lightweight, yet robust
- Support for numerous architectures:

  x86, x86-64, ia64, powerpc, cell / ps3, mips, sparc
- Supported by numerous hardware vendors:
  - HP Labs, AMD, IBM, Intel, Sony, Toshiba, Cray, SiCortex, Broadcom
  - Also received support from Google and RedHat
- Built on well established experience
- Long history of good support

Pfmon overview

- **Console based interface to libpfm/perfmon2**
- **Provides convenient access to performance counters**
- **Wide range of functionality:**
  - Counting events
  - Sampling in regular intervals
  - Flat profile
  - System wide mode
  - Triggers
  - Different data readout “plug-ins” (modules) available

Remember: An iterative approach



CERN openlab

An example: Matrix Multiplication

Why matmul?
1) Simple, analytically understandable
2) Computationally intensive
a) but memory accesses get in your way, if you are not careful
3) FLP-OPS: N^3 with SSE (DP)

Matmul (1): Source code

Two separate source files: matrix.c, multiply.c

```
#define NUM 1024
#define DIM 1024
 static double  a[DIM][DIM], b[DIM][DIM], c[DIM][DIM];
....
 start = _rdtsc();   // Shown on Monday
 multiply_d(a,b,c);
 stop = _rdtsc();
```

```
unsigned long i,j,k;
for(i=0;i<NUM;i++) {
            for(k=0;k<NUM;k++) {
                  for(j=0;j<NUM;j++) {
                        c[i][j] = c[i][j] + a[i][k] * b[k][j];
}}}
```

```
Minimum FP-OPS = 1.073.741.824
Minimum CYCLES =   536.870.912
```

Needed pfmon commands

- Find the counters of interest

```
pfmon -l
```

- Decide if “umasks” are required
- Use multiplexing

```
pfmon -iMEM_INST_RETIRED
Name     : MEM_INST_RETIRED
Desc     : Memory instructions retired
Umask-01 : 0x01 : [LOADS] : Instructions retired which cont
Umask-02 : 0x02 : [STORES] : Instructions retired which con
```

```
pfmon -ecpu_clk_unhalted:thread_p,inst_retired:any_p,
fp_comp_ops_exe:sse_fp,last_level_cache_references
      -emem_inst_retired:loads,mem_inst_retired:stores,
last_level_cache_misses --eu-c --switch-timeout=2
./matrix
```

Matmul (2): Run 1

Compile with icc and O2

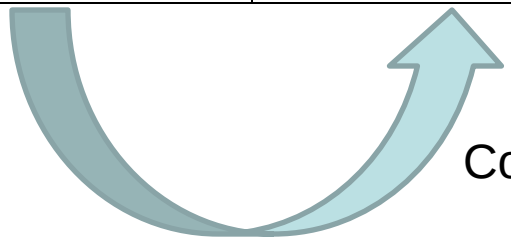
```
Fraction of theoretical throughput limit = 13.25%

4.428.551.833 CPU_CLK_UNHALTED:THREAD_P
7.527.231.268 INST_RETIRED:ANY_P
2.150.475.685 FP_COMP_OPS_EXE:SSE_FP
    7.234.570 LAST_LEVEL_CACHE_REFERENCES
3.221.299.472 MEM_INST_RETIRED:LOADS
1.074.821.795 MEM_INST_RETIRED:STORES
      555.421 LAST_LEVEL_CACHE_MISSES
```

Matmul (2): Run 3

Compile with icc and O3, ipo

Vector-friendly change

Fraction of th	Fraction of throughput limit = 27.02%	
4.428.551.833	2.153.265.512	CPU_CLK_UNHALTED:THREAD_P
7.527.231.268	2.683.340.732	INST_RETIRED:ANY_P
2.150.475.685	1.083.687.270	FP_COMP_OPS_EXE:SSE_FP
7.234.570	70.547.893	LAST_LEVEL_CACHE_REFERENCES
3.221.299.472	1.082.271.127	MEM_INST_RETIRED:LOADS
1.074.821.795	538.017.571	MEM_INST_RETIRED:STORES
555.421	170.399	LAST_LEVEL_CACHE_MISSES



Compared to previous run

Matmul (3): Run 6b

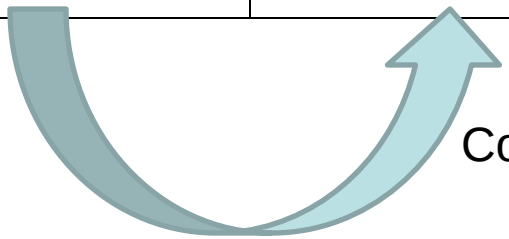
Compile with icc and O3, ipo

Transpose one matrix

Unroll by 4

} Cache-friendly changes

Fraction of th	Fraction of throughput limit = 66.35%
2.153.265.512	875.098.637 CPU_CLK_UNHALTED:THREAD_P
2.683.340.732	2.202.587.599 INST_RETIRED:ANY_P
1.083.687.270	1.078.769.120 FP_COMP_OPS_EXE:SSE_FP
70.547.893	4.455.041 LAST_LEVEL_CACHE_REFERENCES
1.082.271.127	512.036.721 MEM_INST_RETIRED:LOADS
538.017.571	240.648.551 MEM_INST_RETIRED:STORES
170.399	254.890 LAST_LEVEL_CACHE_MISSES



Compared to previous run

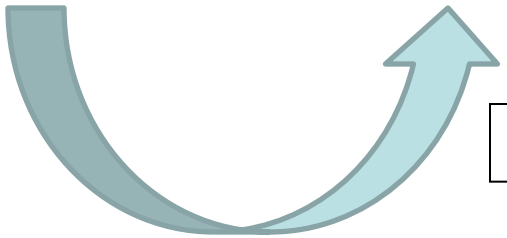
Matmul (4): Overall gain is 5x

Compile with icc and O3, ipo

Transpose b-matrix

Unroll by 4

Fraction of th	Fraction of throughput limit = 66.35%	
4.428.551.833	875.098.637	CPU_CLK_UNHALTED:THREAD_P
7.527.231.268	2.202.587.599	INST_RETIRED:ANY_P
2.150.475.685	1.078.769.120	FP_COMP_OPS_EXE:SSE_FP
7.234.570	4.455.041	LAST_LEVEL_CACHE_REFERENCES
3.221.299.472	512.036.721	MEM_INST_RETIRED:LOADS
1.074.821.795	240.648.551	MEM_INST_RETIRED:STORES
555.421	254.890	LAST_LEVEL_CACHE_MISSES



Compared to first run

Conclusion

- **Low-level performance monitoring works best inside-the-core**
  - On a restricted piece of code:
    - one central algorithm
  - Source code fully available
  - Good knowledge of compiler and platform
  - The theoretically best result is known

Q & A