Analysis and modeling of

# Computational Performance

# Computational Performance

- Performance (efficiency) is (besides correctness, reliability, security, maintainability, user friendliness, etc.) one of the most important software qualities
- Performance, as is understood in the current lectures, has its main related parameter: time-to-solution (execution time)
  - guideline: performance = 1/time-to-solution
- Analysis of computational performance is concerned with elements that influence the time of program execution
- Performance modeling tries to express the execution time in terms of mathematical formulas, using a set of theoretically or experimentally obtained parameters
- Performance optimization finds ways to improve the computational performance of programs and minimize its execution time

# Computational Performance

- In different application areas execution time depends on many different factors:
  - time for performing operations by CPUs
  - time for accessing data in DRAM memory
  - time for sending data over network
  - time for accessing disk drives, SSDs, etc.
  - time for performing transactions with databases
  - time for displaying images and graphics primitives
  - time for creating and displaying video frames
- In the current lectures we are concerned with programs for which execution time depends on the three first factors above

# Computational Performance

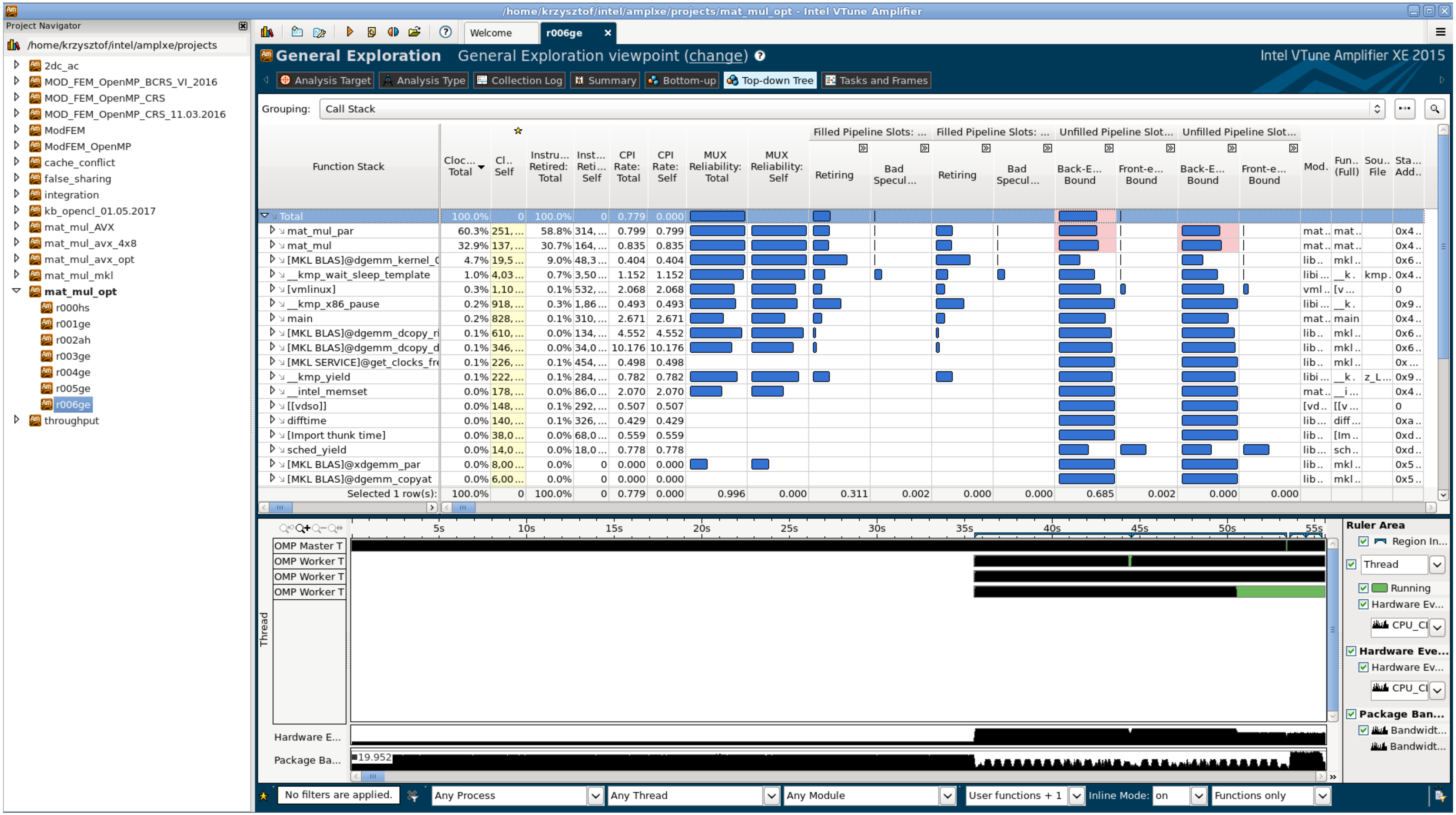
- Current lectures:
  - simple programs in C
    - micro-benchmarks for individual system components and simple operations
    - operations on vectors and matrices – numerical linear algebra
  - hardware-software interaction
    - assembly code
  - benchmarks
  - optimization
    - classical – manual and automatic by compilers
    - parallel
      - multithreading (CPU, GPU)
      - message passing

# Performance tools

- Execution time:
  - wall clock time, elapsed time, real time – external time measure, the most important for software users
  - CPU time – time when CPU was executing program instructions
    - user time – time in user mode
    - system time – time in kernel mode
- Tools for measuring wall clock and CPU time
    - wrist watches, stopwatches
    - *top* utility, system monitors
    - *time* utility in Linux
    - profilers: *gprof*, *valgrind*
    - hardware counters
    - special performance analysis applications
      - Intel VTune, Advisor, NVIDIA Visual Profiler, AMD uProf

# Performance tools

- Intel Vtune – a complex performance analysis tool

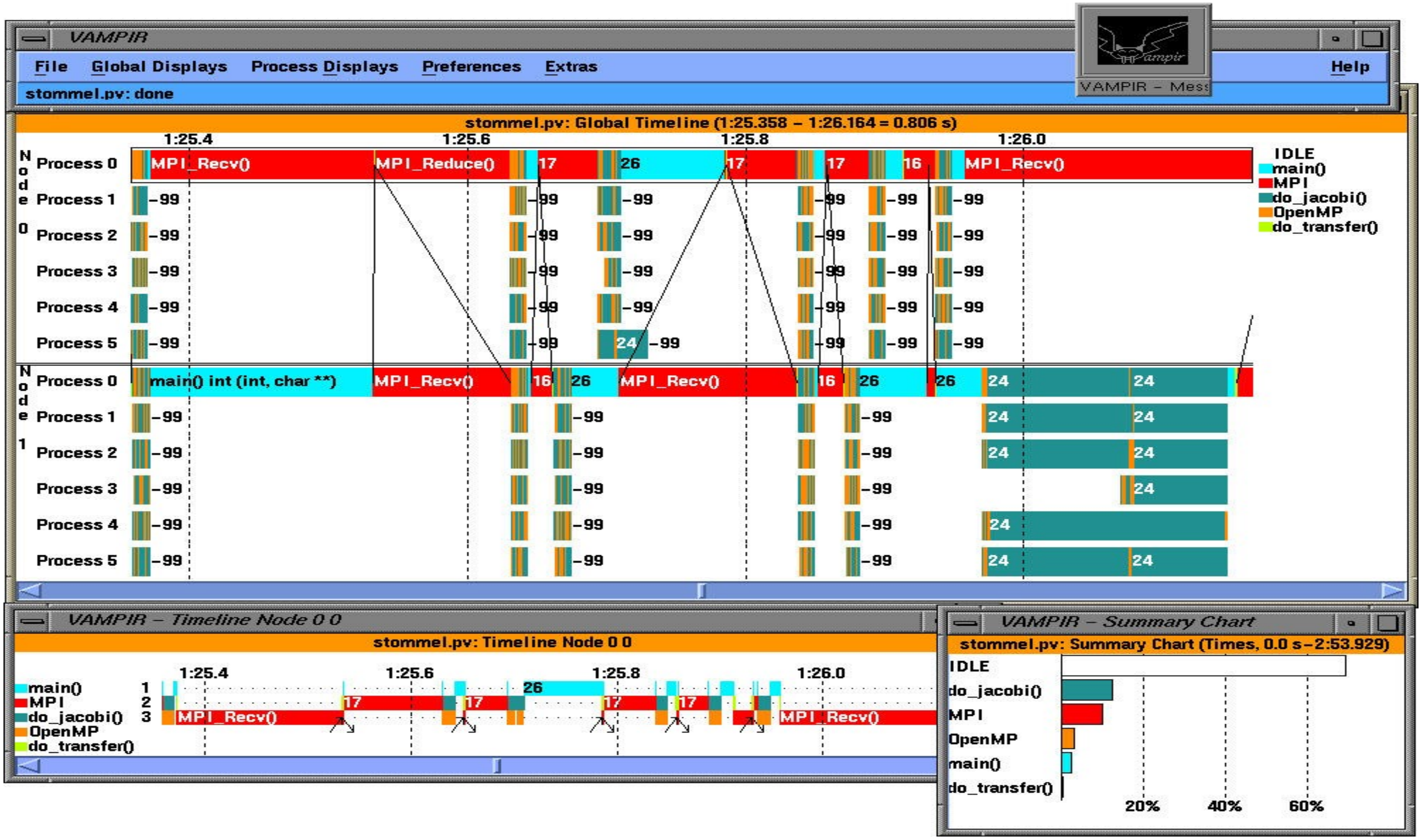


# Performance tools

- Profiling
  - collecting performance related data concerning a given program
    - the main usage of profiling is to give the time spent in different parts of the code
      - subroutines (functions)
      - blocks of code
      - individual lines of code
    - profiling can also report other events during program execution, that can be e.g. used to create:
      - call graph
      - instruction and subroutine (function) number of executions
  - profiling information can be stored and communicated in different ways
    - summary information
    - traces
    - on-line monitoring

# Performance tools - tracers

- A typical output of a popular Vampir tool for MPI tracing



# Performance tools

- Profiling
  - profilers can collect data using different mechanisms:
    - instrumentation (*gprof*)
      - inserting additional code to report the events related to execution and state of the program (e.g. call stack)
      - instrumentation requires special compilation
    - execution simulation (*valgrind*)
      - execution of the program using a special virtual machine
      - simulation incurs significant overhead
    - statistical sampling (*gprof*)
      - program execution is interrupted at specified time intervals and the state of the execution environment is stored (e.g. call stack)
    - event notification
      - for environments (virtual machines) equipped with suitable capabilities

# Performance tools - gprof

- Steps for gprof profiling (using gcc compiler):
  - compilation with instrumentation

    ```
    $ gcc -p source_file.c
    ```
  - standard execution (gmon.out file created)

    ```
    $ a.out
    ```
  - displaying results (binary file as argument, not gmon.out )

    ```
    $ gprof a.out
    ```
  - part of typical output:

```
Flat profile:
Each sample counts as 0.01 seconds.
  %    cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call   name
 45.72     0.48     0.48     31241     0.02     0.02    fun_1
 20.96     0.70     0.22        10    22.00    22.00    fun_2
 15.24     0.86     0.16     31241     0.01     0.01    fun_3
...
```

  - the output can be redirected to a file ( $ gprof a.out > file.txt )

# Hardware counters

- Hardware counters (performance monitoring counters) are special registers for storing the numbers of hardware events related to performance
  - hardware counters are specific for each processor architecture
  - hardware counters are mainly used to support the design and testing of new architectures, as well as fine tuning of compilers and system software
    - hardware events can be very detailed, reflecting the complex nature of contemporary processors
      - example: "IDQ_UOPS_NOT_DELIVERED.CORE - Counts the number of uops not delivered to Resource Allocation Table (RAT) per thread adding “4 – x” when Resource Allocation Table (RAT) is not stalled and Instruction Decode Queue (IDQ) delivers x uops to Resource Allocation Table (RAT) (where x belongs to {0,1,2,3})
  - there are hundreds of hardware events that can be reported by hardware counters

# Hardware counters

- The most important events are related to:
  - time measurements – clock cycles counters
  - instructions executed – especially branches and flops
  - cache and memory access related events – especially cache hits and misses
- There are several applications that provide the interface to hardware counters for different processors and programming environments
  - the basic one for Linux, for recent kernels, is *perf* utility (evolved from Performance Counters for Linux), based on *perf* Linux subsystem and kernel support
  - other popular for Linux:
    - o'profile
    - Performance Application Programming Interface (PAPI) – used during our course

# Performance tools – perf

- Standard usage of perf stat:

  $ perf stat a.out

- Typical output:

```
Performance counter stats for 'a.out':
      0,649995      task-clock (msec)        #    0,697 CPUs utilized
            21      context-switches         #    0,032 M/sec
             0      cpu-migrations           #    0,000 K/sec
           294      page-faults              #    0,452 M/sec
       2443055      cycles                   #    3,759 GHz
       2486027      instructions             #    1,02  insn per cycle
        490849      branches                 #  755,158 M/sec
         14307      branch-misses            #    2,91% of all branches
 ...
```

  - more details can be obtained with options

    $ perf stat -d -d a.out

# Optimizing compilers

- Contemporary compilers can have dozens of optimization options
  - examples (for *gcc*):
    - -fstrength-reduce, -fcse-follow-jumps, -ffast-math, -funroll-loops, -fschedule-insns, -finline-functions, -fomit-frame-pointer
  - important optimizations concern parallelization and vectorization
    - often in order to use particular optimizations for a given hardware (concerning e.g. vectorization) special options have to be passed explicitly to the compiler – e.g. **`-march=core-avx2`** – for cores with AVX2 instructions
    - often directives in source code help compilers to optimize
- In practice, most often compiler optimization is applied using options for optimization levels
  - typical levels and performed optimizations are:
    - -O0 – no optimization
    - -O1 – optimize for execution time and code size
    - -O2 – more optimization options applied, without sacrificing too much time and going into options that can alter the results of code execution
    - -O3 – the most aggressive optimization
    - (some compilers can have more levels, e.g. for vectorization, parallelization)

# „Numbers every programmer should know”

- Examples:
  - L1 cache reference 1 ns
  - Branch mispredict 5 ns
  - L2 cache reference 5 ns
  - Mutex lock/unlock 25 ns
  - Main memory reference 100 ns
  - Send 4K bytes over 10 Gbps network 10,000 ns
  - Transfer 1MB to/from PCI-E GPU 80,000ns
  - Round trip within same datacenter 500,000 ns
  - Read 1 MB sequentially from SATA SSD 2,000,000 ns
  - Read 1 MB sequentially from disk 5,000,000 ns
  - Read 1 MB sequentially from disk 30,000,000 ns
  - Send packet CA->Netherlands->CA 150,000,000 ns
- Current list: https://gist.github.com/eshelman/343a1c46cb3fba142c1afdcdeec17646