Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → **Motivation for serial code tuning**
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Scalability Myth: Code scalability is the key issue

Lore 1

In a world of highly parallel computer architectures only highly scalable codes will survive

Lore 2

Single core performance no longer matters since we have so many of them and use scalable codes

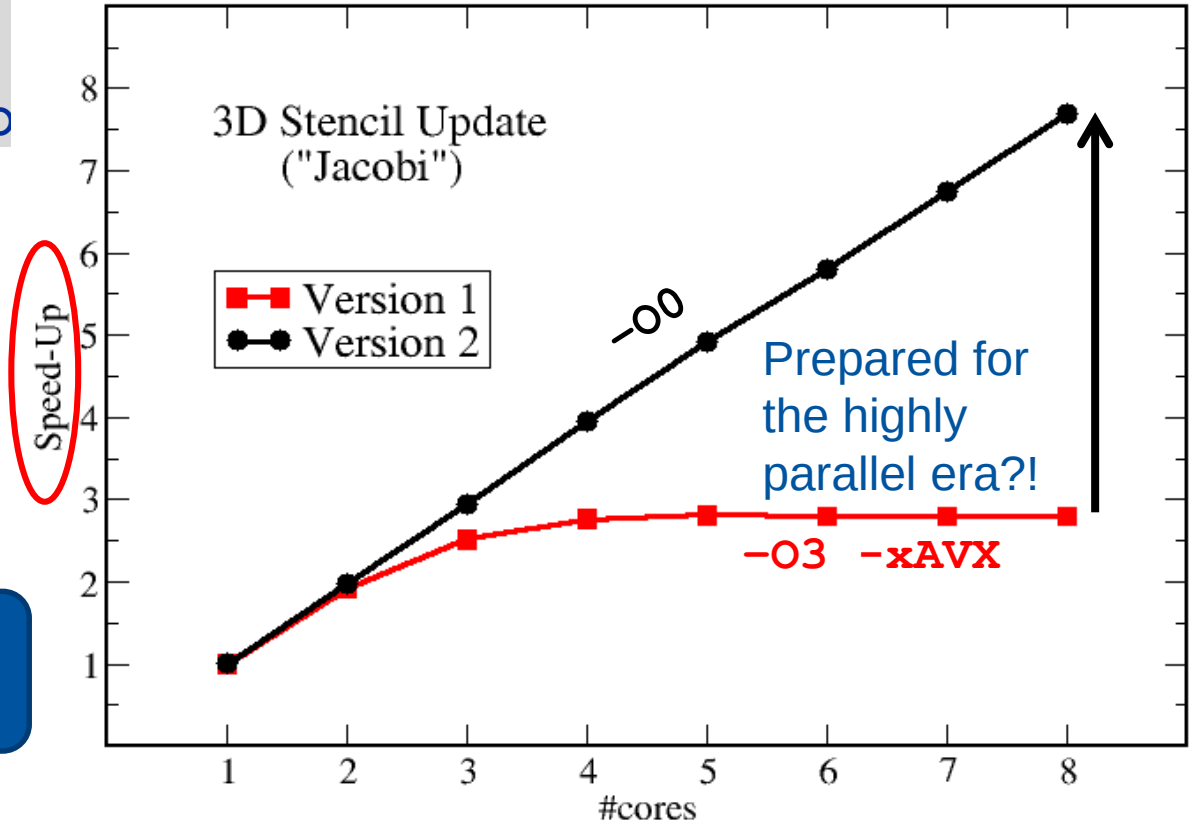
Courtesy of Erlangen Regional Computing Center (RRZE)

Scalability Myth: Code scalability is the key issue

parallel program

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
     y(i,j,k)= b*(  x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                    x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```

Changing only the compile options makes this code scalable on an 8-core chip



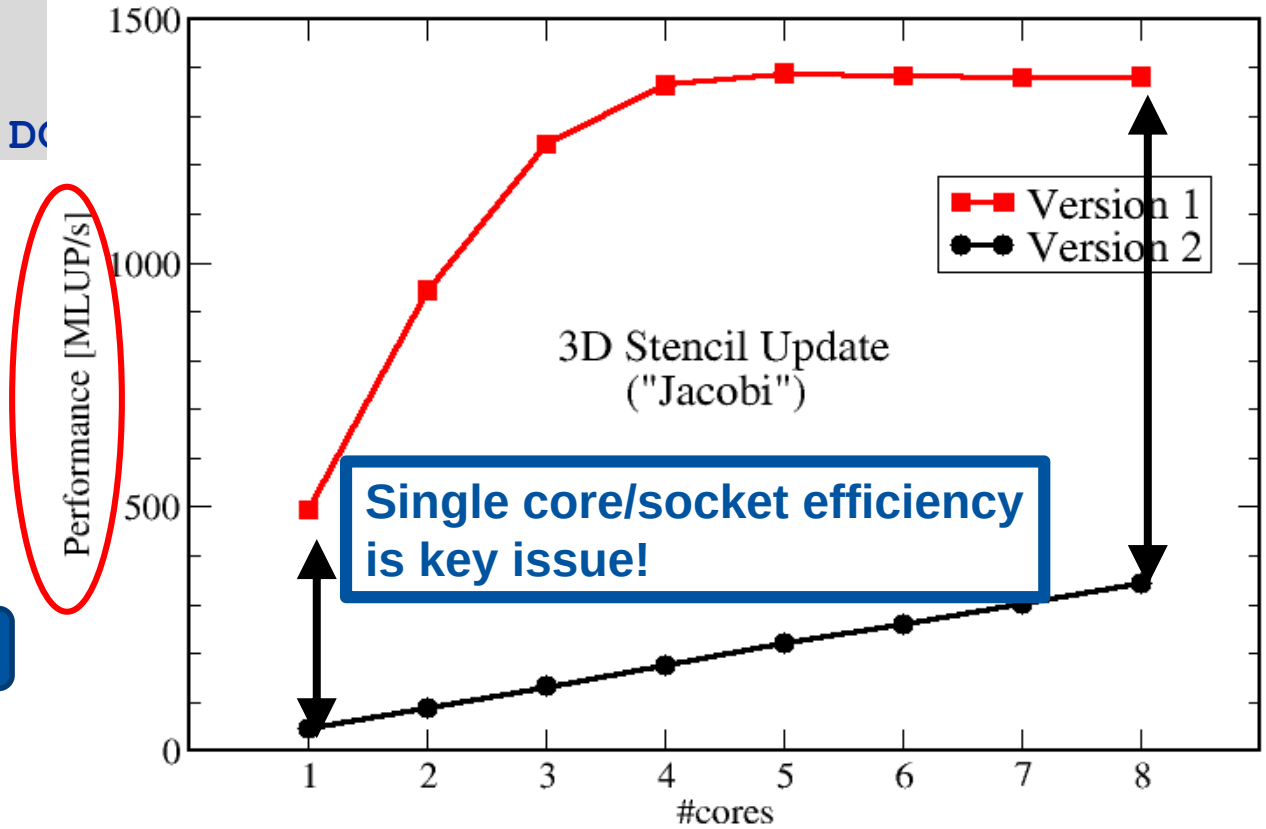
Parallel program is X times faster than serial program.

Courtesy of Erlangen Regional Computing Center (RRZE)

Scalability Myth: Code scalability is the key issue

parallel program

```
!$OMP PARALLEL DO
do k = 1 , Nk
  do j = 1 , Nj; do i = 1 , Ni
     y(i,j,k)= b*(  x(i-1,j,k)+ x(i+1,j,k)+ x(i,j-1,k)+
                    x(i,j+1,k)+ x(i,j,k-1)+ x(i,j,k+1))
    enddo; enddo
enddo
!$OMP END PARALLEL DO
```



Absolute performance

Courtesy of Erlangen Regional Computing Center (RRZE)

Improving single core performance

- **Serial code optimization/ tuning makes code more efficient on single processors and thus can reduce the overall amount of needed compute power**
  - → Resources can be put to better use
  - → Must be the first goal when optimizing an existing parallel code

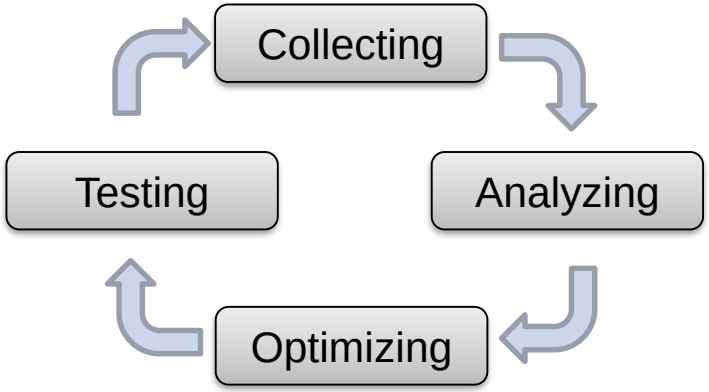
- **How to do?**

  1. Find out where most of the runtime is spent (collect data)

     → Statically analysis of code *or*

     → Retrieving information about a programs runtime behavior (e.g. by profiling) → dynamic approach

     → Usually one determines how much time is spent in certain functions to possibly identify *hot spots*



Improving single core performance

- **Hot spots**
  - → Parts of the program that require the dominant fraction of the total runtime
  - → 90/10 law: 90% of the runtime in a program is spent in 10% of the code
    - → From experience: nowadays, it's not so easy anymore
    - → Still: Hot spot analysis is important
- **How to do? - continued**

2. Find out why most of the runtime is spent there (analyze data)
   - → Determine which factors stall performance (e.g. by hardware counters)
3. Optimize your code to get a decreased runtime
4. Test the correctness of code & its performance
   - → Test size not too small, since performance behavior changes with the size of the memory consumption
   - → not too large, since the tests need to be done quite often to compare tuning steps
   - → Start with step (1) if test not successful

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Monitoring

- **A common method for optimization is monitoring a system's activities**

- **Reasons to monitor activities on a system**
  - → System programmer:
    - → Determine frequently executed portions of a program to target them for further optimization
  - → System manager:
    - → Measure resource utilization and determine performance bottlenecks
    - → System parameter tuning
  - → System Analyst:
    - → Use profiling data to characterize the workload for capacity planning

Monitoring terminology

- **Event**
  - → Pre-defined change in the system's state
  - → Definition depends on measured metric
    - → E.g. memory reference, processor interrupts, application processing, disk access, network activity
- **Profile**
  - → Aggregated picture of an application program
  - → E.g. accumulated runtime spent in each function
- **Trace**
  - → A log/sequence of individual events
  - → Includes event type and important system parameters
- **Overhead**
  - → Perturbation introduced by the monitoring technique

Monitor classifications

- **Level of monitor implementation**
  - → Hardware monitor
  - → Software monitor
  - → Hybrid
- **Trigger mechanisms**
  - → Event-driven
  - → Sample-driven
- **Recording**
  - → Profiling
  - → Tracing
- **Displaying ability**
  - → On-line
  - → Batch/Post mortem

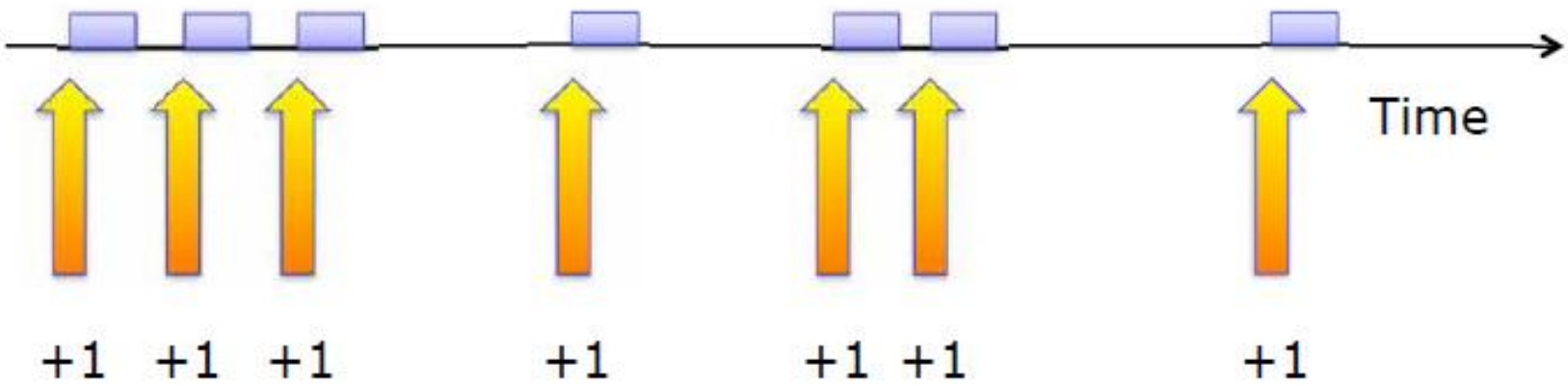
Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → **Event- & sample-driven triggers**
       - → **Overview**
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Event-driven triggers

- **Measures performance *only* when the pre-selected event occurs**
- **Modify system to record event**
- **Infrequent events → small overhead**
- **Frequent events → large overhead**
- **Can significantly alter program behavior**
- **Overhead assessment not easy**
- **Good for tools with low-frequency events**

Event-driven triggers – Example

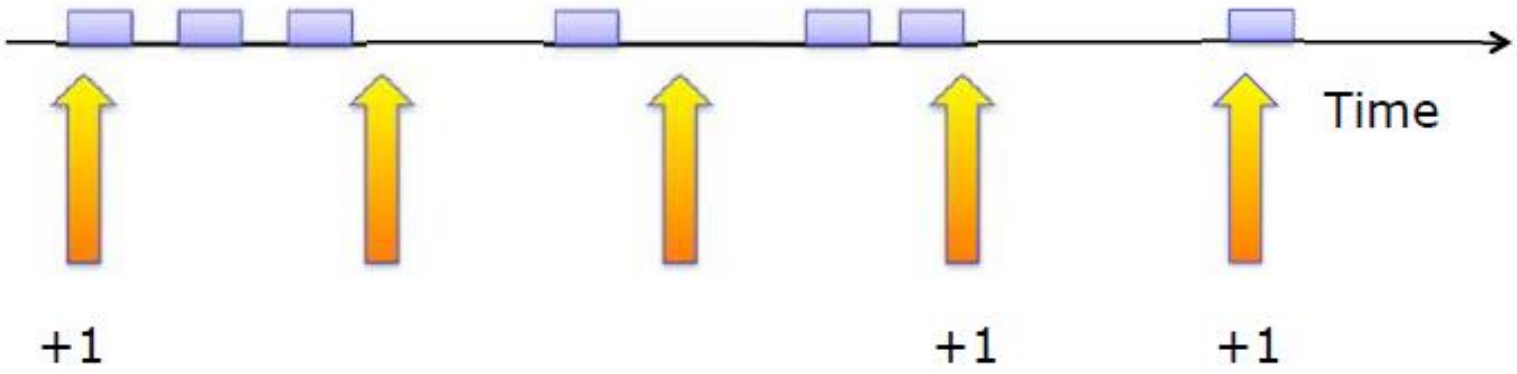


- **7 of 7 events are observed**

Sample-driven triggers

- **Performance is measured over snapshots at fixed time intervals**
- **Overhead of this technique is independent of the number of specific events**
  - → It depends on the frequency of snapshots taken (sampling frequency)
- **Will not measure every occurrence of a specific event**
- **Produces statistical view on the overall behavior of a system**
  - → Infrequent events may not be covered
- **Only very long runs are likely to produce comparable results**

Sample-driven triggers – Example



- **Only 3 of 7 events are observed in 5 samples**

Trigger comparison

	Event Trigger	Sample Trigger
Precision	Exact	Probabilistic
Perturbation	O(f(N_events))	Fixed
Overhead	O(f(N_events)) Depends on: ▪ Event types instrumented ▪ Program behavior ▪ Overhead per event	Constant Depends on: ▪ Sampling rate ▪ Overhead per sample

Interval timers

- **Measure execution time**
  - → Based on the concept of measuring actual clock cycles
- **Types**
  - → Hardware counters (see later)
    - → Counter of N-bit precision
    - → Value is number of clock cycles since initialization
  - → Software timers
    - → Based on interrupts initiated by hardware
    - → Value is count of interrupts
- **Some problems**
  - → Timer resolution determines quantization error
    - → Very short events might be missed
  - → Counters can overflow

The lecture *Performance and correctness analysis of parallel programs* will go more into different interval timers and their problems.

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → **Event- & sample-driven triggers**
       - → Overview
       - → **Basic block counting**
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Basic block counting

- **A basic block is a sequence of instructions that has no branches in or out of the sequence**
- **Add instructions to the block to count the number of times the block is executed**
- **After termination: Measurements form an execution histogram**

- **Difference to sampling**

  → Block counting gives the exact number of times a block was executed

- **Due to significant runtime overhead, block counting can have a severe impact on a program's behavior/performance**

- **Overhead: Added instructions, changed memory access**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → **Event- & sample-driven triggers**
       - → Overview
       - → Basic block counting
       - → **Instrumentation**
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Instrumentation

- **Instrumenting a program means**
  - → Putting additional instructions into the program for monitoring certain components (usually at least runtime)
- **Instrumented code triggers events (during runtime)**
  - → E.g. on entering and exiting a certain function/ basic block
- **Implemented by**
  - → Source code modification (manual instrumentation)
  - → Software exceptions
  - → Emulation
  - → Library
  - → Modification by the compiler

Source code modifications by manual instrumentation

- **Requires the programmer to add instrumentation statements to his source code manually**
  - → Adds overhead due to additional executed instructions
- **Advantage**
  - → Programmer can choose which part of the program is worth measuring
    - → Enables a pre-selection of possible hotspots
- **Disadvantage**
  - → Not automatic → time consuming
  - → Error prone
  - → Non-experienced programmers may think that they have a clear understanding of the program's workflow and miss non-obvious hotspots

Software exceptions

- **Certain types of processors support software exceptions just before the execution of each instruction**
- **Exception handler can be installed to interpret the instruction and its operands**
- **Advantages**
  - → Very accurate
  - → Very detailed
- **Disadvantages**
  - → By far too detailed in most cases
  - → Too low-level to enable an easy interpretation of workflow

Emulation

- **An emulator is a program that makes the system it is running on look like something completely different to the outside**

- **JVM is an application that**
  - → interprets Java byte-code
  - → translates it into machine instructions
  - → emulates a processor that can interpret java byte-code instruction format

- **Tracing is easily implemented then but**
  - → Emulation is slow compared to native code execution

Libraries

- **Parallel programs often use communication or runtime libraries**
  - → e.g. OpenMP, runtime library
  - → e.g. MPI implementation (differs depending on the system)
- **These libraries can either be instrumented or covered by instrumentation-wrapper replacements**
- **Gives quite a good overview on the program's behavior**

Modification by the compiler

- **Let the compiler add instrumentation instructions to the executable code it compiled automatically**
- **Similar to basic block profiling**
- **Two versions**
  - → As a compilation option
  - → Post-compilation software tool
- **Advantage**
  - → Fully automatic
- **Overhead**
  - → Depends on how often functions are called
  - → Usually different compiler options to control the overhead

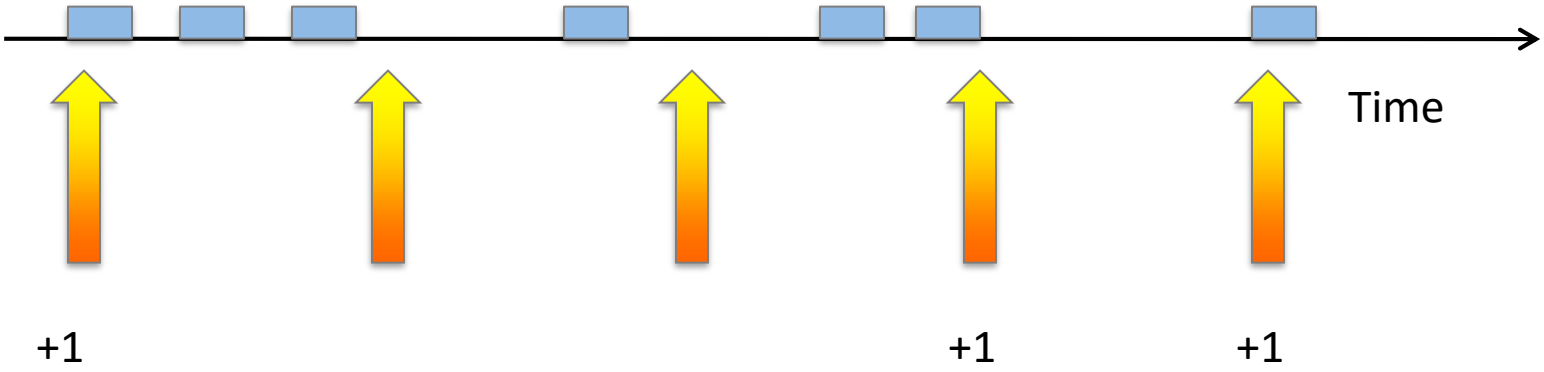
Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → **Event- & sample-driven triggers**
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → **PC sampling**
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

PC sampling

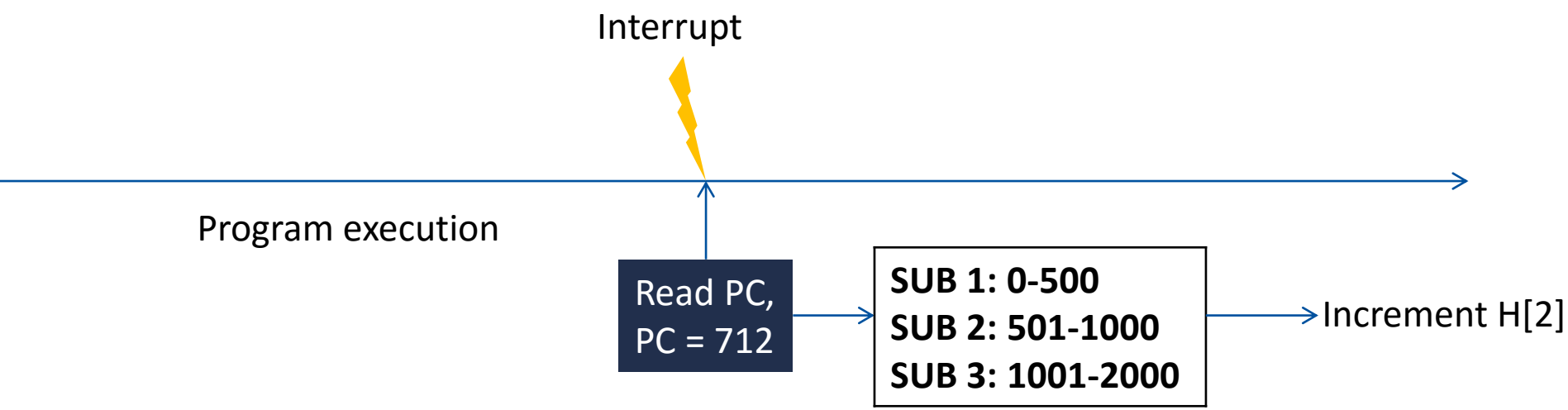
- **Program counter (PC) sampling is a statistical measurement**
  - → Only a subset of the actual program flow is measured at fixed intervals
  - → Assumption: The overall behavior will follow the characteristics of the subset measured
  - → Profile: samples taken at fixed times
  - → Record appropriate state information at each step
  - → Post-process to obtain overall profile

PC sampling – Example



- **Step**
  - → Examine PC on return address stack
  - → Use address map to translate PC into subroutine i
  - → Increment counter array element H[i]

PC sampling – Example



Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → **Profiling & tracing**
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Profiling

- **Recording technique**
- **Retrieving information about a programs runtime behavior**
  - → Most important detail is actual runtime
- **Summary information!**
  - → Does not provide information about the logical order in which the events occurred
- **Applies instrumentation or sampling for triggering**

Function profiling

- **Profile information per function**
  - → Exclusive (not counting any callees of the function) or inclusive (including callees of function) runtimes
  - → Flat profile or callgraph profile
- **Outcome can depend crucially on the compiler's ability to perform function inlining**
  - → Output may be distorted when some hot spot function gets inlined (runtime is attributed to its caller)
  - → Maybe disallow inlining when profiling (but may effect performance)
- **Tools, e.g.**
  - → gprof (uses instrumen-ation + sampling)
  - → Intel Vtune Amplfier XE

gprof example

```
%     cumulative    self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 77.0      1.24     1.24       15    82.67    82.67  dgemm_ [5]
  8.7      1.38     0.14        2    70.00    70.00  matgen_ [6]
  5.6      1.47     0.09       17     5.29     5.29  dtrsm_ [7]
```

% of overall program runtime used exclu-sively by this function

#seconds used by this function (exclusive)

#calls of this function

Average number of ms per call that were spent in this function (exclusive)

Average number of ms per call that were spent in this function (inclusive)

Line-based profiling

- **Function profiling becomes useless when functions with hot spots are large (in terms of code lines)**
  - → Code line-based profiling
- **Debug symbols must be included into the binary**
  - → Machine instruction's address in memory must be properly matched to the correct source line
  - → Usually by compiling with –g
- **Outcome may be distorted**
  - → Loop fusion, line arrangement, variable optimization, pipeline architecture
- **Profiling data on a loop-by-loop basis is usually safe (samples integrated across the loop body) check with inlining disabled**
- **Tools, e.g.**
  - → gprof
  - → Intel Vtune Amplifier XE

gprof example

```
%     cumulative    self
 time   seconds   seconds    name
 42.98     39.65    39.65    main (jacobi.c:82 @ 402653)
 42.79     79.12    39.47    __c_mcopy8
 13.60     91.67    12.55    _mp_barrier_tw
  0.56     92.19     0.52    main (jacobi.c:40 @ 401ded)
  0.04     92.23     0.04    main (jacobi.c:90 @ 4027c8)
```

Tracing

- **Recording technique**
- **Time ordered list of all the events that were recorded during program flow (*event trace*)**

  Contains

  → Information about the program's state, e.g. all instructions executed

  → Sequences of memory accesses, disk blocks referenced and messages sent over the network interface
- **In contrast to profiling**

  → Records more than the simple fact that a certain event has occurred

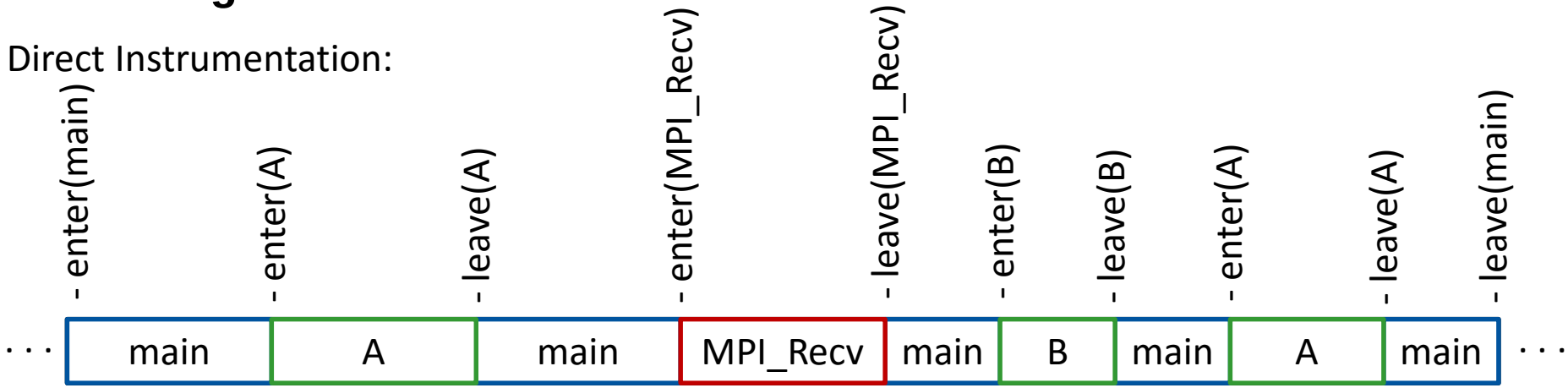
  → e.g. instead of keeping just the number of page faults, a tracing record strategy may store the addresses that caused the page fault

  → Requires significantly more storage

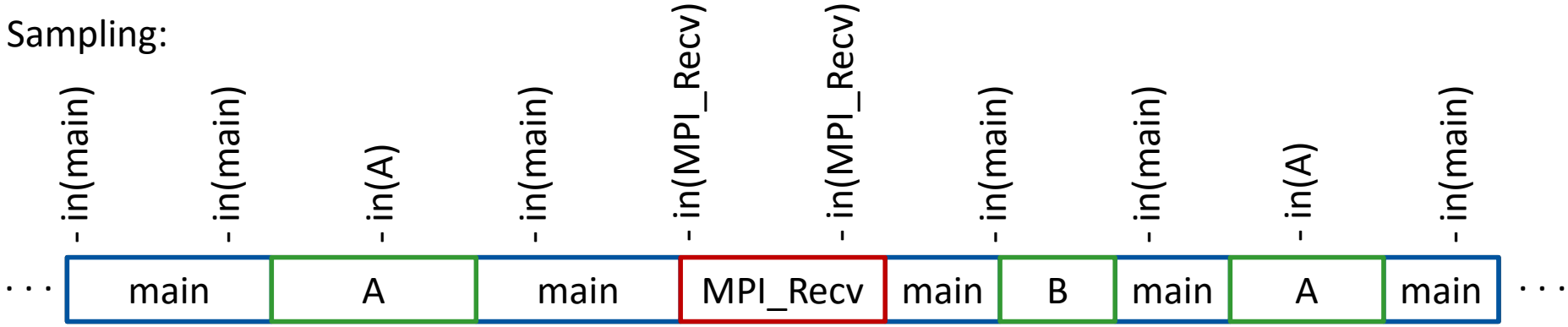
Tracing

- **Tracing stores all information in an event trace.**

Direct Instrumentation:



Sampling:

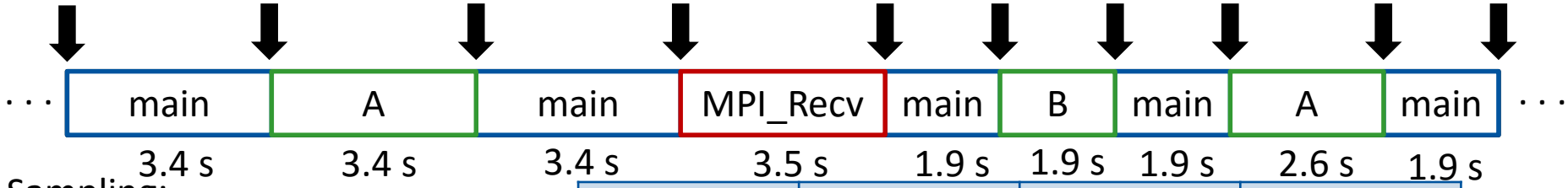


Tracing

- **Tracing stores all information in an event trace.**

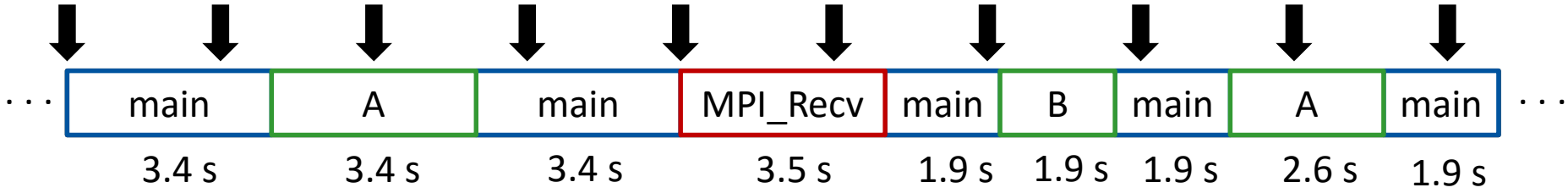
Direct Instrumentation:

main	A	MPI_Recv	B
12.5 s	6.0 s	3.5 s	1.9 s



Sampling:

main	A	MPI_Recv	B
12.5 s	5.0 s	5.0 s	0 s



Tracing advantages & disadvantages

- **Difficulties**
  - → Performance slow-down
  - → Perturbations induced by additional tracing instructions
  - → Amount of data collected (huge size)
  - → Additional slowdown due to writing tracing information to disk
  - → THUS, time required to save the program state may significantly alter the program's performance characteristics / behavior
- **Advantages**
  - → Detailed results
  - → Summary information can be computed for any subset of time space
  - → Useful for both Performance tuning and debugging
  - → Identification of synchronization issues

Recording comparison

	Tracing	Profiling
Precision	exact information	accumulated information
Overhead	higher overhead Depends on ▪ the number of events	lower runtime overhead
Space requirements	easily hundreds of MB or GB for larger applications Depends on ▪ the number of events	smaller amount of space needed normally some MB

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → **Hardware performance counters**
     - → Overview of tools
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Hardware performance counters

- **Help to identify why code is slow**
  - → What is the limiting resource?

- **Definition**

  In computers, hardware performance counters, or hardware counters are a set of special-purpose registers built into modern microprocessors to store the counts of hardware-related activities within computer systems. Advanced users often rely on those counters to conduct low-level performance analysis or tuning.

  (from: http://en.wikipedia.org)

- **Only small number of counters per processor**
  - → Often far less than ten
- **Absolute counter values are often hard to interpret**
  - → How to interpret 10,000 cache misses? Good? Bad?

Example

- **Hardware Counters of an Intel Nehalem Processor:**

SB_DRAIN.ANY, STORE_BLOCKS.AT_RET, STOF… DAD_MISSES.PDE_MIS, DTLB_LOAD_MISSES.LARGE_W,
MEM_INST_RETIRED.LOADS, MEM_INST_RET… MEM_UNCORE_RETIRED.L3_D,
MEM_UNCORE_RETIRED.OTHE, MEM_UNCOF… MP_OPS_EXE.MMX, FP_COMP_OPS_EXE.SSE_FP,
FP_COMP_OPS_EXE.SSE2_INT, FP_COMP_OP… NT_128.PACKED_SHIFT, SIMD_INT_128.PACK,
SIMD_INT_128.UNPACK, SIMD_INT_128.PACK… B, LOAD_DISPATCH.ANY, ARITH.CYCLES_DIV_BUSY,
INST_QUEUE_WRITES, INST_DECODED.DEC0, … D_HIT, L2_RQSTS.RFO_MISS, L2_RQSTS.RFOS,
L2_RQSTS.IFETCH_HIT, L2_RQSTS.IFETCH_MIS… S.DEMAND.S_S, L2_DATA_RQSTS.DEMAND.E_S,
L2_DATA_RQSTS.DEMAND.M_, L2_DATA_RQS… A_RQSTS.PREFETCH.M, L2_WRITE.RFO.I_STATE,
L2_WRITE.RFO.S_STATE, L2_WRITE.RFO.M_ST… E, L2_WRITE.LOCK.HIT, L2_WRITE.LOCK.MESI,
L1D_WB_L2.I_STATE, L1D_WB_L2.S_STATE, L1D_WB_L2.E_STATE, L1D_WB_L2.M_STATE, L1D_WB_L2.MESI, CPU_CLK_UNHALTED.THREAD, CPU_CLK_UNHALTED.REF_P, L1D_CACHE_LD.I_STATE, L1D_CACHE_LD.S_STATE, L1D_CACHE_LD.E_STATE,
L1D_CACHE_LD.M_STATE, L1D_CACHE_LD.MESI, L1D_CACHE_ST.S_STATE, L1D_CACHE_ST.E_STATE, L1D_CACHE_ST.M_STATE, L1D_CACHE_LOCK.HIT, L1D_CACHE_LOCK.S_STATE, L1D_CACHE_… E_STATE, L1D_CACHE_LOCK.M_STATE, L1D_ALL_REF.ANY,
L1D_ALL_REF.CACHEABLE, DTLB_MISSES.ANY, DTLB_MISSES.WALK_COMPLET, DTLB_MISSES.STLB_HIT, DTLB_MISSES.PDE_MISS, DTLB_MISSES.LARGE_WALK_C, LOAD_HIT_PRE, L1D_PREFETCH.… L1D_PREFETCH.MISS, L1D_PREFETCH.TRIGGERS,
L1D.M_REPL, L1D.M_EVICT, L1D.M_SNOOP_EVICT, L1D_CACHE_PREFETCH_LOCK, L1D_CACHE_LOCK_FB_HIT, CACHE_LOCK_CYCLES.L1D_L2, CACHE_LOCK_CYCLES.L1D, IO_TRANSACTIONS, L1I.CY… LARGE, ITLB.HIT, ITLB_MISSES.ANY,
ITLB_MISSES.WALK_COMPLET, ILD_STALL.LCP, ILD_STALL.MRU, ILD_STALL.IQ_FULL, ILD_STALL.REGEN, ILD_STALL.ANY, BR_INST_EXEC.COND, BR_INST_EXEC.DIRECT, BR_INST_EXEC.IND… EA,
BR_INST_EXEC.DIRECT_NEAR, BR_INST_EXEC.INDIRECT_NEA, BR_INST_EXEC.NEAR_CALLS, BR_INST_EXEC.TAKEN, BR_MISP_EXEC.COND, BR_MISP_EXEC.DIRECT, BR_MISP_EXEC.INDIREC…
BR_MISP_EXEC.DIRECT_NEAR, BR_MISP_EXEC.INDIRECT_NEA, BR_MISP_EXEC.NEAR_CALLS, BR_MISP_EXEC.TAKEN, RESOURCE_STALLS.ANY, RESOURCE_STALLS.LOAD, RESOURCE_STAL… LL,
RESOURCE_STALLS.FPCW, RESOURCE_STALLS.MXCSR, RESOURCE_STALLS.OTHER, MACRO_INSTS.FUSIONS_DECO, BACLEAR_FORCE_IQ, ITLB_FLUSH, OFFCORE_REQUESTS.L1D_WR, UOP…
UOPS_EXECUTED.PORT3_COR, UOPS_EXECUTED.PORT3_COR, UOPS_EXECUTED.PORT4_COR, UOPS_EXECUTED.PORT5, UOPS_EXECUTED.PORT015, UOPS_EXECUTED.PORT234, OFFCOR… E.HIT,
SNOOP_RE… D.ANY_P, INST_RETIRED.X87, INST_RETIRED.MMX, UOPS_RETIRED.ANY, UOPS_RETIRED.RETIRE_SLOTS, UOPS_RETIRED.MACRO_FUSE, MACHINE_CLEARS.CYCLES,
MACHINE_… INST_RETIRED.CONDITION, BR_INST_RETIRED.NEAR_CAL, BR_MISP_RETIRED.ALL_BRAN, BR_MISP_RETIRED.NEAR_CAL, SSEX_UOPS_RETIRED.PACKED,
SSEX_UOP… R, SSEX_UOPS_RETIRED.VECTOR, ITLB_MISS_RETIRED, MEM_LOAD_RETIRED.L1D_HIT, MEM_LOAD_RETIRED.L2_HIT, MEM_LOAD_RETIRED.L3_UNS,
MEM_LOA… LFB, MEM_LOAD_RETIRED.DTLB_MI, FP_MMX_TRANS.TO_FP, FP_MMX_TRANS.TO_MMX, FP_MMX_TRANS.ANY, MACRO_INSTS.DECODED, UOPS_DECODED.MS,
UOPS_DEC… LLS.REGISTERS, RAT_STALLS.ROB_READ_POR, RAT_STALLS.SCOREBOARD, RAT_STALLS.ANY, SEG_RENAME_STALLS, ES_REG_RENAMES, UOP_UNFUSION,
BR_INST_D… BPU_CLEARS.LATE, L2_TRANSACTIONS.LOAD, L2_TRANSACTIONS.RFO, L2_TRANSACTIONS.IFETCH, L2_TRANSACTIONS.PREFETCH, L2_TRANSACTIONS.L1D_WB,
L2_TRANSA… TATE, L2_LINES_IN.E_STATE, L2_LINES_IN.ANY, L2_LINES_OUT.DEMAND_CLEA, L2_LINES_OUT.DEMAND_DIRT, L2_LINES_OUT.PREFETCH_CLE,
L2_LINES_O… _CYCLES, FP_ASSIST.ALL, FP_ASSIST.OUTPUT, FP_ASSIST.INPUT, SIMD_INT_64.PACKED_MPY, SIMD_INT_64.PACKED_SHIFT, SIMD_INT_64.PACK,
SIMD_INT_64.UNPACK, SIMD_INT_64.PACKED_LOGICA, CP… PACKED_ARITH, SIMD_INT_64.SHUFFLE_MOVE, UNC_GQ_CYCLES_FULL.READ_, UNC_GQ_CYCLES_FULL.WRITE, UNC_GQ_CYCLES_FULL.PEER_, UNC_GQ_CYCLES_NOT_EMPTY,
UNC_GQ_CYCLES_NOT_EMPTY, UNC_GQ_CYCLES_NOT_EMPTY, UNC_… READ_TRACK, UNC_GQ_ALLOC.RT_L3_MISS, UNC_GQ_ALLOC.RT_TO_L3_RE, UNC_GQ_ALLOC.RT_TO_RTID_, UNC_GQ_ALLOC.WT_TO_RTID, UNC_GQ_ALLOC.WRITE_TRAC,
UNC_GQ_ALLOC.PEER_PROBE, UNC_GQ_DATA.FROM_QPI, UNC_GQ_DATA.FRO… MC, UNC_GQ_DATA.FROM_L3, UNC_GQ_DATA.FROM_CORES_, UNC_GQ_DATA.FROM_CORES_, UNC_GQ_DATA.TO_QPI_QMC, UNC_GQ_DATA.TO_L3,
UNC_GQ_DATA.TO_CORES, UNC_SNP_RESP_… P_RESP_TO_LOCAL_H, UNC_SNP_RESP_TO_REMOTE,
UNC_SNP_RESP_TO_REMOTE, UNC_SNP_RES… HITS.READ, UNC_L3_HITS.WRITE, UNC_L3_HITS.PROBE,
UNC_L3_HITS.ANY, UNC_L3_MISS.READ, UNC… NC_L3_LINES_IN.F_STATE, UNC_L3_LINES_IN.ANY,
UNC_L3_LINES_OUT.M_STATE, UNC_L3_LINES… QUESTS.IOH_RE, UNC_QHL_REQUESTS.IOH_WR,
UNC_QHL_REQUESTS.REMOTE, UNC_QHL_RE… L_CYCLES_FULL.LOCA, UNC_QHL_CYCLES_NOT_EMPT,
UNC_QHL_CYCLES_NOT_EMPT, UNC_QHL_CY… _QHL_ADDRESS_CONFLIC, UNC_QHL_CONFLICT_CYCLES.I,
UNC_QHL_CONFLICT_CYCLES., UNC_QHL_CO… MC_NORMAL_FULL.WRI, UNC_QMC_NORMAL_FULL.WRI,
UNC_QMC_NORMAL_FULL.WRI, UNC_QMC_… UNC_QMC_ISOC_FULL.WRITE.C, UNC_QMC_BUSY.READ.CH0,
UNC_QMC_BUSY.READ.CH1, UNC_QMC_BUS… CUPANCY.CH1, UNC_QMC_OCCUPANCY.CH2,
UNC_QMC_ISSOC_OCCUPANCY., UNC_QMC_… C, UNC_QMC_NORMAL_READS.C,
UNC_QMC_NORMAL_READS.A, UNC_QMC_H… NC_QMC_CRITICAL_PRIORIT, UNC_QMC_CRITICAL_PRIORIT,
UNC_QMC_CRITICAL_PRIORIT, UNC_QMC_W… MC_WRITES.PARTIAL.C, UNC_QMC_WRITES.PARTIAL.C,
UNC_QMC_CANCEL.CH0, UNC_QMC_CANCEL… TE, UNC_QMC_PRIORITY_UPDATE,
UNC_QHL_FRC_ACK_CNFLTS.L, UNC_QPI_TX… I_TX_STALLED_SINGL, UNC_QPI_TX_STALLED_SINGL,
UNC_QPI_TX_STALLED_SINGL, UNC_QPI_TX… QPI_TX_STALLED_MULTI, UNC_QPI_TX_STALLED_MULTI,
UNC_QPI_TX_HEADER.BUSY.LI, UNC_QPI_TX_HEADER.BUSY.LI, UNC_QPI_RX_NO_PPT_CREDI, UNC_QPI_RX_NO_PPT_CREDI, UNC_DRAM_OPEN.CH0, UNC_DRAM_OPEN.CH1, UNC_DRAM_OPEN.CH2, UNC_DRAM_PAGE_CLOSE.CH0,
UNC_DRAM_PAGE_CLOSE.CH1, UNC_DRAM_PAGE_CLOSE.CH2, UNC_DRAM_PAGE_MISS.CH0, UNC_DRAM_PAGE_MISS.CH1, UNC_DRAM_PAGE_MISS.CH2, UNC_DRAM_READ_CAS.CH0, UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_READ_CAS.CH1,
UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_READ_CAS.CH2, UNC_DRAM_READ_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH0, UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH1, UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_WRITE_CAS.CH2,
UNC_DRAM_WRITE_CAS.AUTO, UNC_DRAM_REFRESH.CH0

L1I.HITS:
Counts all instruction fetches that hit the L1 instruction cache.

Some are easy to understand

Many are hard to grasp.
→ Abstraction layers are important.

BR_MISP_EXEC.COND:
Counts the number of mispredicted conditional near branch instructions executed, but not necessarily retired.

Overview of useful events

- **Number of bus transactions (= cache line transfers)**
  - → Often *cache misses* are rather monitored
    - → Prefetching mechanisms can interfere with #cache misses counted
  - → Counting bus transactions is safer way to account for the actual data volume transferred over the memory bus
  - → If close to maximum bandwidth (given by the Stream benchmark): bus utilization must not be optimized

- **Number of loads and stores**
  - → Indication as to how efficiently cache lines are used for computation
  - → E.g. if #DP loads/store per cache line < its length in DP words → may be strided memory access

Overview of useful events

- **Number of floating-point operations**
  - → Different counts for single and double precision; packed and scalar
  - → Derived metric (together with runtime): Floating-point operations per second (Flop/s)
    - → If close to peak Flop/s of architecture: standard code optimization must not be done

- **Mispredicted branches**
  - → Counter is incremented when CPU has predicted the outcome of a conditional branch and prediction has proved to be wrong
  - → Penalty for a mispredicted branch can be tens of cycles (depending on architecture)

Overview of useful events

- **Pipeline stalls**
  - → Cannot be avoided if performance is limited by memory bandwidth
  - → Difficult to identify the point where there are “too many” bubbles
  - → Especially important on in-order architectures (e.g. GPUs)

- **Number of instructions executed**
  - → Together with clock cycles: how effectively the superscalar hardware is utilized
  - → Derived metric: instructions per cycle
    - → Experiences for compiler-generated code: not over 2-3 instructions per cycle
  - → Derived metric: clock cycles per instruction (CPI)
    - → CPI < 1 is desired

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → **Monitoring**
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → **Overview of tools**
   - → Evolution of performance aspects
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

gprof

- **Event-based instrumentation + sampling**
- **Compile programs with the option –pg**
- **Available on the RWTH compute cluster**
- **During execution a profile file named gmon.out is created**

→ Can be analyzed with:

gprof <program_name>

```
                                     called/total       parents
index  %time    self descendents  called+self    name       index
                                     called/total       children
-----------------------------------------------
                1.24        0.00      15/15          dgetrf_ [4]
[5]     77.0    1.24        0.00      15         dgemm_ [5]
                0.00        0.00      30/150         lsame_ [16]
-----------------------------------------------
...
 %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 77.0      1.24     1.24       15    82.67    82.67  dgemm_ [5]
  8.7      1.38     0.14        2    70.00    70.00  matgen_ [6]
  5.6      1.47     0.09       17     5.29     5.29  dtrsm_ [7]
  3.7      1.53     0.06      984     0.06     0.06  dger_ [9]
  3.1      1.58     0.05       32     1.56     1.56  dlaswp_ [10]
  0.6      1.59     0.01      990     0.01     0.01  dswap_ [12]
  0.6      1.60     0.01       16     0.62     5.00  dgetf2_ [8]
  0.6      1.61     0.01        1    10.00    10.00  dmxpy_ [13]
  0.0      1.61     0.00     1000     0.00     0.00  idamax_ [14]
  0.0      1.61     0.00      999     0.00     0.00  dscal_ [15]
  0.0      1.61     0.00      150     0.00     0.00  lsame_ [16]
```

Intel VTune Amplifier

- Sampling tool
- Comes with analyzer tool to visualize results
- Stack tracing method
- Option for analysis using hardware counters
- Available on the RWTH compute cluster

- Usage on RWTH compute cluster:

```
>module load intelvtune
>amplxe-gui
```

GUI based, analysis relatively intuitive

Score-P

- **Supports tracing and profiling**
- **Uses direct instrumentation**
- **Supports C/C++ and Fortran with MPI, OpenMP and hybrid codes**
- **Useful for large scale applications**

- **Available on the RWTH compute cluster:**

```
>module load UNITE
>module load scorep
```

- **Usage**
  1. Precede your compiler command with *scorep*
  2. (a) Run your application as usual to generate a profile

     (b) Set SCOREP_ENABLE_TRACING=true, SCOREP_ENABLE_PROFILING=false and run the application for a trace.
  3. Analyze the data in scorep-XXXXXX

```
#C:
>scorep icc test.c –openmp –o a.out
#MPI:
>scorep mpicc test.c –openmp –o a.out
```

Performance API (PAPI)

- **Standard API to access hardware counters**
  - → provides access to a set of hardware counters with standardized names and over a standardized interface
  - → used in many tools for hardware counter access (also in Score-P)
  - → papi_avail provides a list of available counters

```
    Name        Code     Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   No   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  Yes   Yes  Level 1 cache misses
PAPI_L2_TCM  0x80000007  Yes   No   Level 2 cache misses
```

developed at:
THE UNIVERSITY of TENNESSEE

What you have learnt

- **What is a hot spot?**
- **What kind of trigger mechanisms do exist?**
  - → What is the difference between event-driven and sample-driven triggers?
  - → What is instrumentation?
- **What kind of recording mechanisms do exist?**
  - → What is the difference between profiling and tracing?
- **What can hardware performance counter measure?**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → **Evolution of performance aspects**
   - → Common sense optimizations
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

What we have learnt from the history

- **Every optimization method is only valid in a (historical) context**

- **Many optimization methods have a limited shelf life**
  - → Architectures change over time
  - → Technologies/ compilers/ software change over time
  - → Corresponding tuning methods must change

- **What we teach today might be outdated in a couple of years**

- **Nevertheless, a general sensibility of the architectural design stays important**

Arithmetic optimization

```
*
*     SNRM2 := sqrt( x'*x ).
*
*  Further Details
*  ===============
*
*  -- This version written on 25-October-1982.
*     Modified on 14-October-1993 to inline the call to SLASSQ.
*     Sven Hammarling, Nag Ltd.
*
*  =====================================================================
*
*     .. Parameters ..
      REAL ONE,ZERO
      PARAMETER (ONE=1.0E+0,ZERO=0.0E+0)
*     ..
*     .. Local Scalars ..
      REAL ABSXI,NORM,SCALE,SSQ
      INTEGER IX
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC ABS,SQRT
*     ..
      IF (N.LT.1 .OR. INCX.LT.1) THEN
          NORM = ZERO
      ELSE IF (N.EQ.1) THEN
          NORM = ABS(X(1))
      ELSE
          SCALE = ZERO
          SSQ = ONE
*         The following loop is equivalent to this call to the LAPACK
*         auxiliary routine:
```

...

from netlib.org
(old) BLAS reference implementation

Once upon a time…
Optimization of arithmetic operations must be done by hand
If (s == 1): result = value
Else: result = s*value

Today: compiler may do it/ arithmetic operations are less expensive

Note: most processors vendors offer optimized versions for their microprocessor architectures

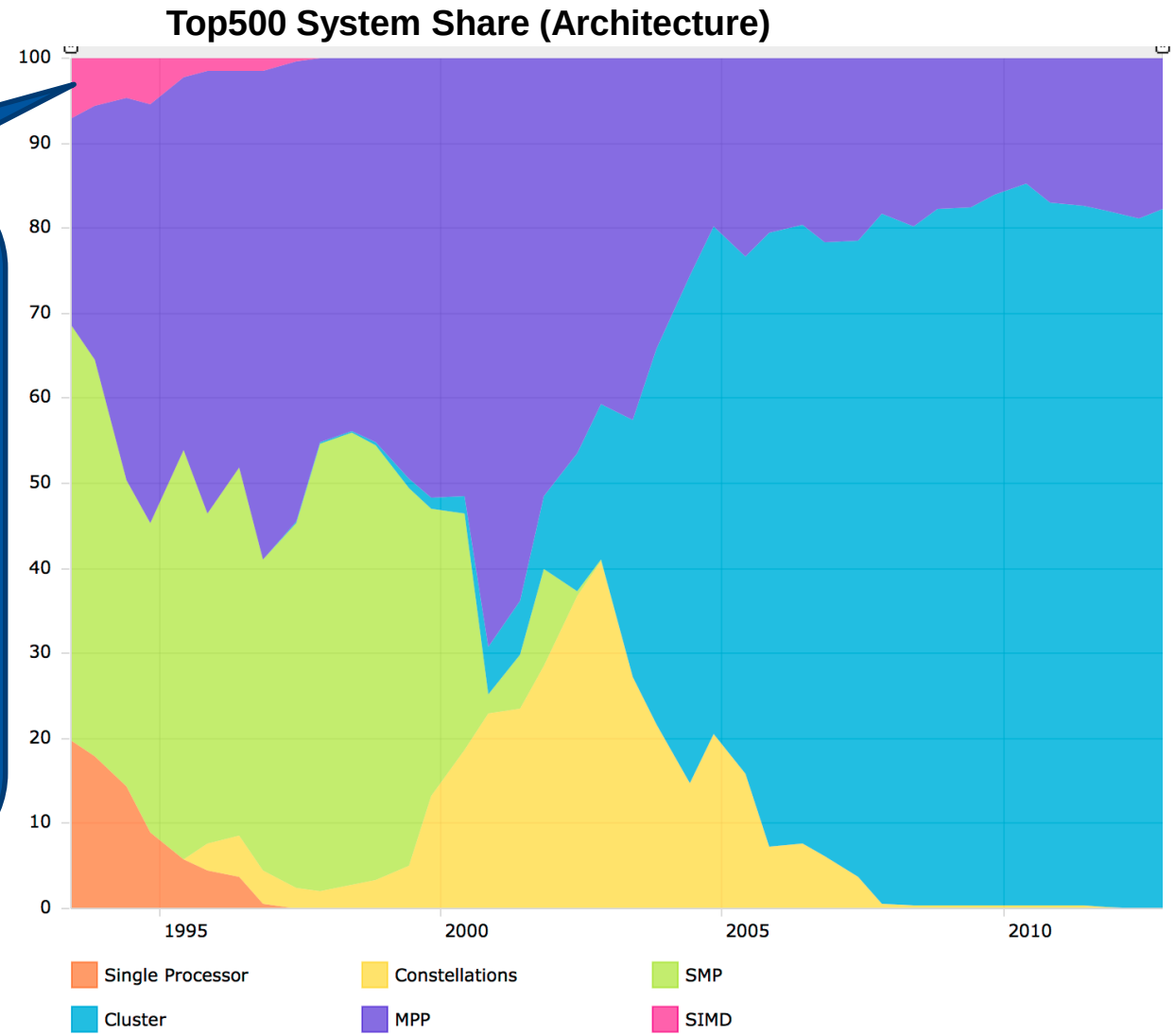
SIMD optimization

Once upon a time…

- Vector processors were common
- SIMD optimization was the most important tuning activity

Today

- Commodity processors did not support SIMD for a long time
- Now it gets more important again (SSE, AVX,…)



Cache optimization

CPU	Year	Bit Width	#Transistors	Clock	L1 / L2 /L3
4004	1971	4	2300	740 kHz	
8008	1972	8	3500	500 kHz	
8086	1978	16	29.000	10 Mhz	
80286	1982	16	134.000	25 MHz	
80386	1985	32	275.000	33 Mhz	
80486	1989	32	1.200.000	50 MHz	8K
Pentium I	1994	32	3.100.000	66 MHz	8K
Pentium II	1997	32	7.500.000	300 MHz	16K/512K*
Pentium III	1999	32	9.500.000	600 MHz	16K/512K*
Pentium IV	2000	32	42.000.000	1.5 GHz	8K/256K
Pentium D	2005	64	115.000.000	3.2 GHz	16K/2MB
Core i7	2008	64	731.000.000	3.2 GHz	32K/256K/8MB
Westmere-EP	2010	64	1.170.000.000	3.46 GHz	32K/256K/12MB
Haswell-EP	2014	64	5.560.000.000	2.3 GHz	32K/256K/45MB
Broadwell-EP	2016	64	7.200.000.000	2.2 GHZ	32K/256K/55MB

Once upon a time…

- CPUs had no internal caches or only small ones

Today

- CPUs comprise different levels of caches
- Cache optimization important to tackle the CPU-memory gap

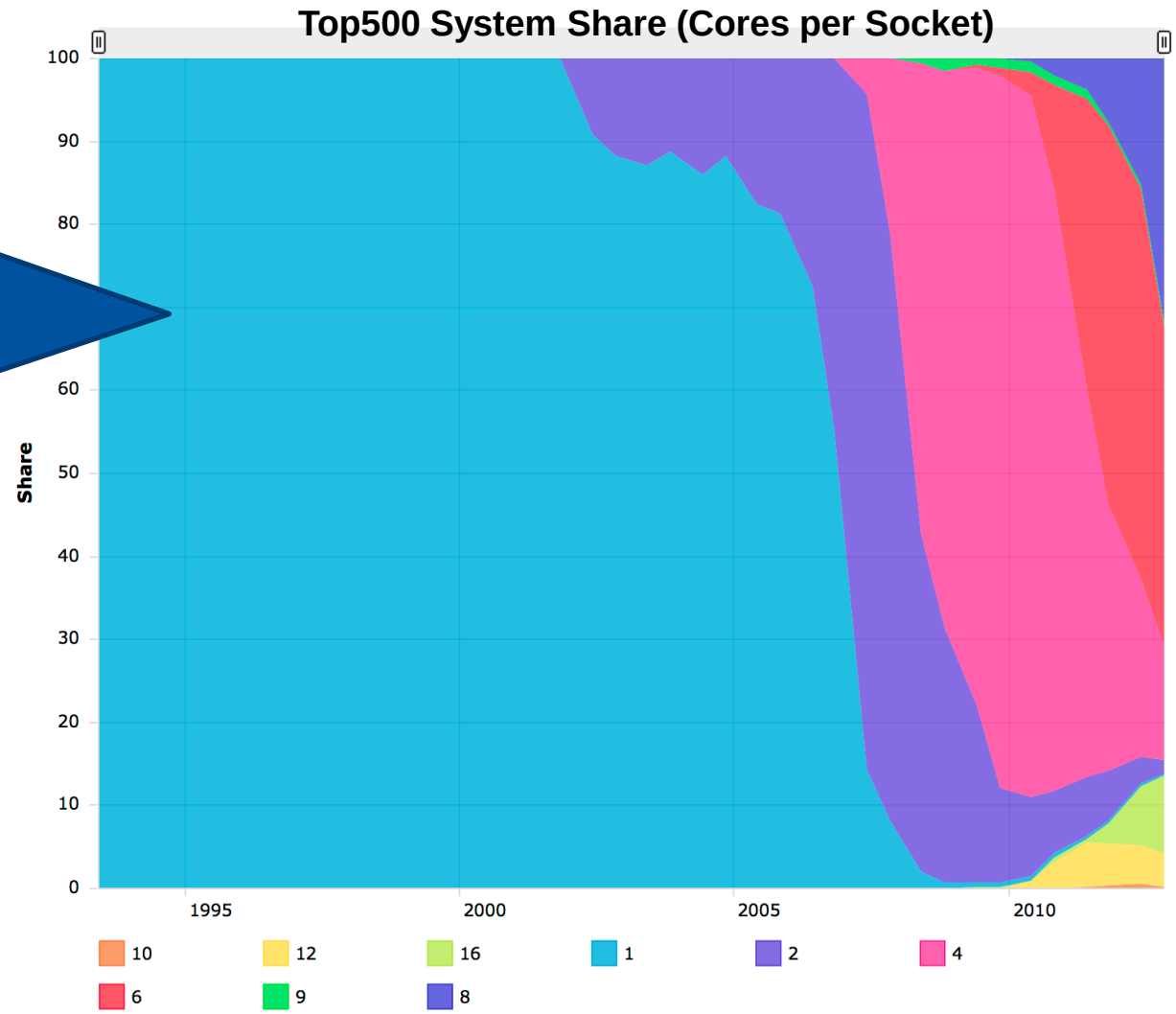
Multi-core optimization

Once upon a time…

- Single-core processors were common

Today

- Multi-core processors
- Multi-core optimization is important (parallelism, data affinity,…)



Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → **Do less work**
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Do less work – Simple loops

- **Do less work**
  - → Rearrange code such that less work than before is being done
- **Many programs can benefit from small code changes that despite their trivial complexity can significantly increase performance**

- **Loop example**
  - → Often, programs do more work than required

C/C++

```
for(i=0; i<N; ++i)
{
  if( data[i] % 10 == 0 )
  {
    flag=true;
  }
}
```

Fortran

```
do i=1,N
  if(mod(data(i), 10)==0)
   flag=.true.
  end if
end do
```

Do less work – Simple loops

- **If the condition induces no side effects, the loop may break after the flag got set to true the first time:**

C/C++

```
for(i=0; i<N; ++i)
{
  if( data[i] % 10 == 0 )
  {
    flag=true;
    break;
  }
}
```

Fortran

```
do i=1,N
  if(mod(data(i), 10)==0)
   flag=.true.
   exit
  end if
end do
```

- **If one other element in the dataset fits to the condition, it has no further effect since flag is already set to true. Therefore processing further elements is redundant and waste of computational resources**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → **Avoid expensive operations**
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Avoid expensive operations

- **Some implementations just translate the formulas into code without respect to performance issues**
  - → Good, but often “expensive” operations (e.g. sin(x))
- **Performance optimization by replacing expensive operations by cheaper alternatives**
  - → Keep in mind that performance optimization bears the slight danger of changing numerics or even results
- **A common example:**

```
while(condition)
{
  […]
  int x = (someval % 10);
  double s = sin(x);
}
```

Tabulating

- **It can be profitable to consider e.g. the input range of expensive functions (such as trigonometric, e.g. sin, cos, tan, exp,…)**
- **Optimization technique is called tabulating**

Table setup (executed once):

```
for(x = 0; x < 10; ++x)
{
  sin_table[x] = sin(x);
}
```

```
while(condition)
{
  […]
  int x = (someval % 10);
  double s = sin_table[x];
}
```

Tabulating

- **Table lookup is done at virtually no costs compared to the execution of the sine-function**
- **Lookup-tables can, depending on their size, fit into the L1 Cache and have very few CPU cycles of access time**
- **Tabulating can not be applied when the input range to function can not be isolated**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → **Shrink the working set**
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Shrink the working set

- **Working set of code: amount of memory it uses or rather touches**
- **Shrinking the working set is always an optimization**
  - → Higher probability for cache hits
- **Consider the types of variables you are using**
  - → Doubles, floats, integers,…
  - → E.g. a byte can serve the same purpose as an integer
    - → And thus more data can fit into the L2/L3 Cache
- **Consider: Not all microprocessors handle "small" datatypes as efficient as word-size datatypes**
  - → If the algorithm operates on a byte basis, loading a byte may result into load + masking + shift operation instead of a simple load
- **If SIMD operations can be applied to multiple data elements at once, shrinking the working set may become quite effective**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → **Eliminate common subexpressions**
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Eliminate common subexpressions

- **If parts of complex expressions can be precalculated, they should not be explicitly calculated in e.g. a loop construct**
- **In case of loops this optimization is called loop invariant code motion:**

```
for(i = 0; i < N; ++i)
{
  a[i] = a[i]+s+r*sin(x);
}
```

```
tmp = s+r*sin(x);
for(i = 0; i < N; ++i)
{
  a[i] = a[i]+tmp;
}
```

- **Compiler can detect this situation in principle**
  - → If the compiler needs to apply associativity rules it may refrain from doing so
  - → You may need to help the compiler

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → **Avoid branches**
     - → Use SIMD instruction sets
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Avoid branches

- **Branches may prevent the compiler from applying loop unrolling or SIMD vectorization (especially in loops)**
  - → Avoid branches whenever possible
- **Processor does speculative execution**
  - → But mispredicted branches are costly
- **An example:**

```
for(i = 0; i < N; ++i)
{
  for(j = 0; j < N; ++j)
  {
    if(i < j)
      a[i][j] = a[i][j]+1;
    else if(i > j)
      a[i][j] = a[i][j]-1;
  }
}
```

j →

i ↓

–	+1	+1	+1	+1
-1	–	+1	+1	+1
-1	-1	–	+1	+1
-1	-1	-1	–	+1
-1	-1	-1	-1	–

Avoid branches

- **In certain situations loop nests may be transformed so that all conditional statements vanish:**

```
for(i = 0; i < N; ++i)
{
  for(j = 0; j < N; ++j)
  {
    if(i < j)
      a[i][j] = a[i][j]+1;
    else if(i > j)
      a[i][j] = a[i][j]-1;
  }
}
```

→

```
for(i = 0; i < N; ++i)
{
  for(j = i+1; j < N; ++j)
    a[i][j] = a[i][j]+1;
  for(j = 0; j < i; ++j)
    a[i][j] = a[i][j]-1;
}
```

- **Clearly the second variant has a bigger optimization potential**

Outline

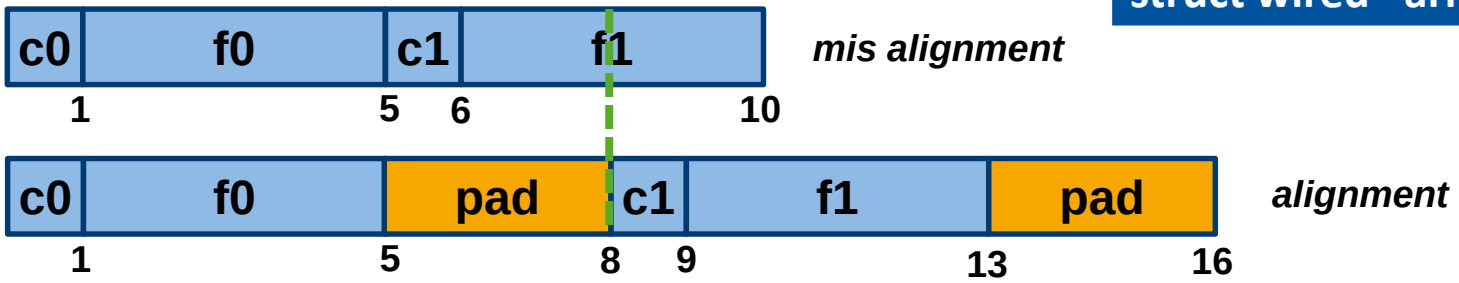
1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → **Use SIMD instruction sets**
     - → Compiler impact
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Use SIMD instruction sets

- **Vectorization enables several operations with a single instruction**
  - → Register-to-register operations have increased performance
  - → If data type is smaller, more results can be gained
  - → Data must be aligned
    - → Optimize data access by padding

```
struct wired {
  char c;
  float f;
};
struct wired *array;
```

Assume: data access per 8 Byte



- **But, using SIMD over scalar instructions is not a guarantee for performance improvement**
  - → Memory-bound code will not accelerate much

    (registers just wait longer on data)

Use SIMD instruction sets

- **Check on your compiler for vectorization**
  - → It may generate non-optimal code
  - → Manual inspection of vectorized code (assembly level) is the only option in such a scenario

- **Use compiler intrinsics if the compiler cannot vectorize your code**

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → **Compiler impact**
     - → C++ optimizations
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Compiler impact

- **Compiler option can have a great impact on program performance**
  - → Compiler based optimization has varying influence on code, but are in general beneficial
- **Every modern compiler has command line switches to enable or disable certain optimization patterns**
- **Check different compilers for more performance potential**
- **Do not rely on the compiler to identify every optimization potential**
  - → Design your code in the most economical way in terms of performance
    - → e.g. not: pow(x,2) or x**2 but: x*x
- **Compilers can be surprisingly intelligent and foolish at the same time**

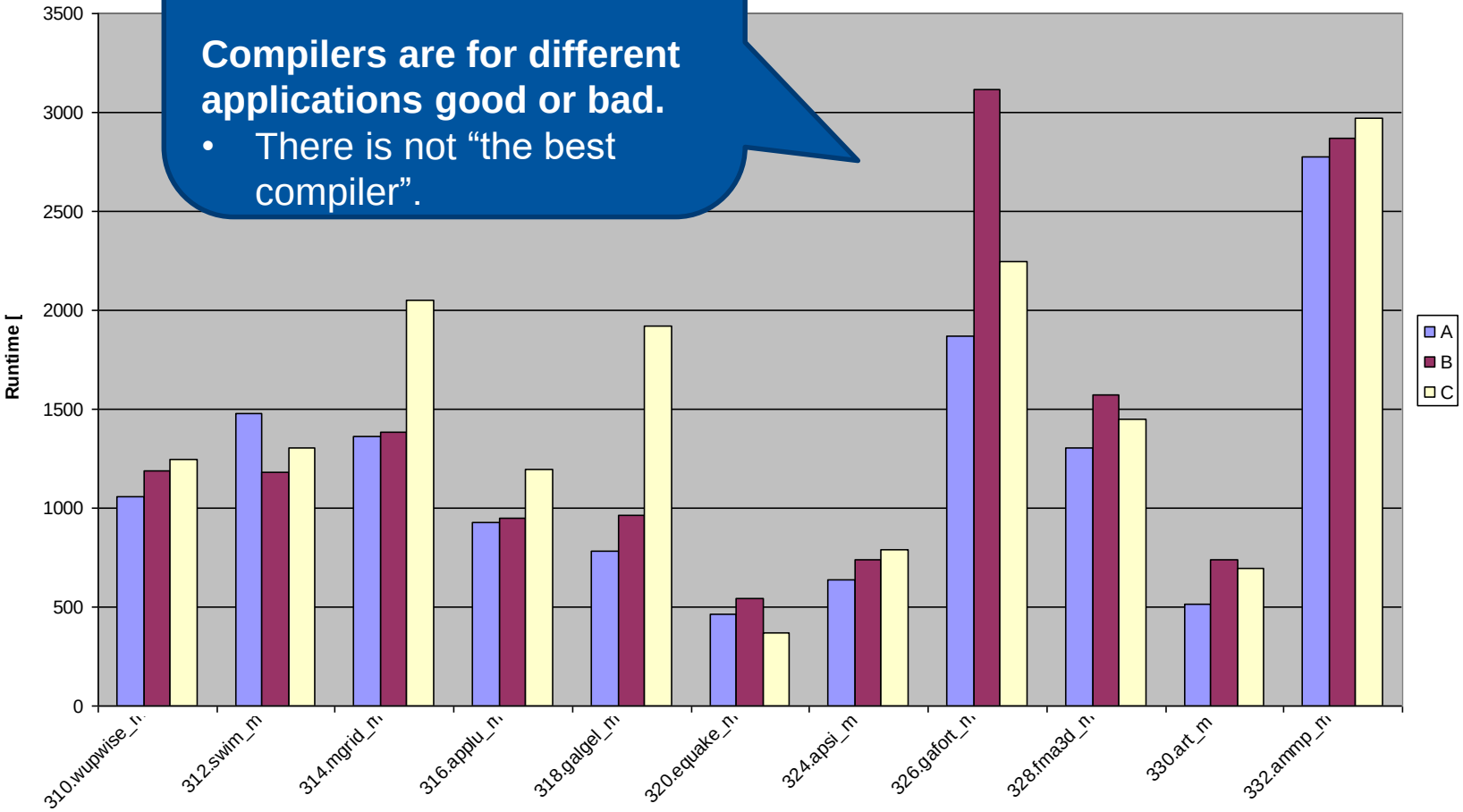
Compiler comparison with SPEC OMPM2001

Different compilers can deliver different performances.

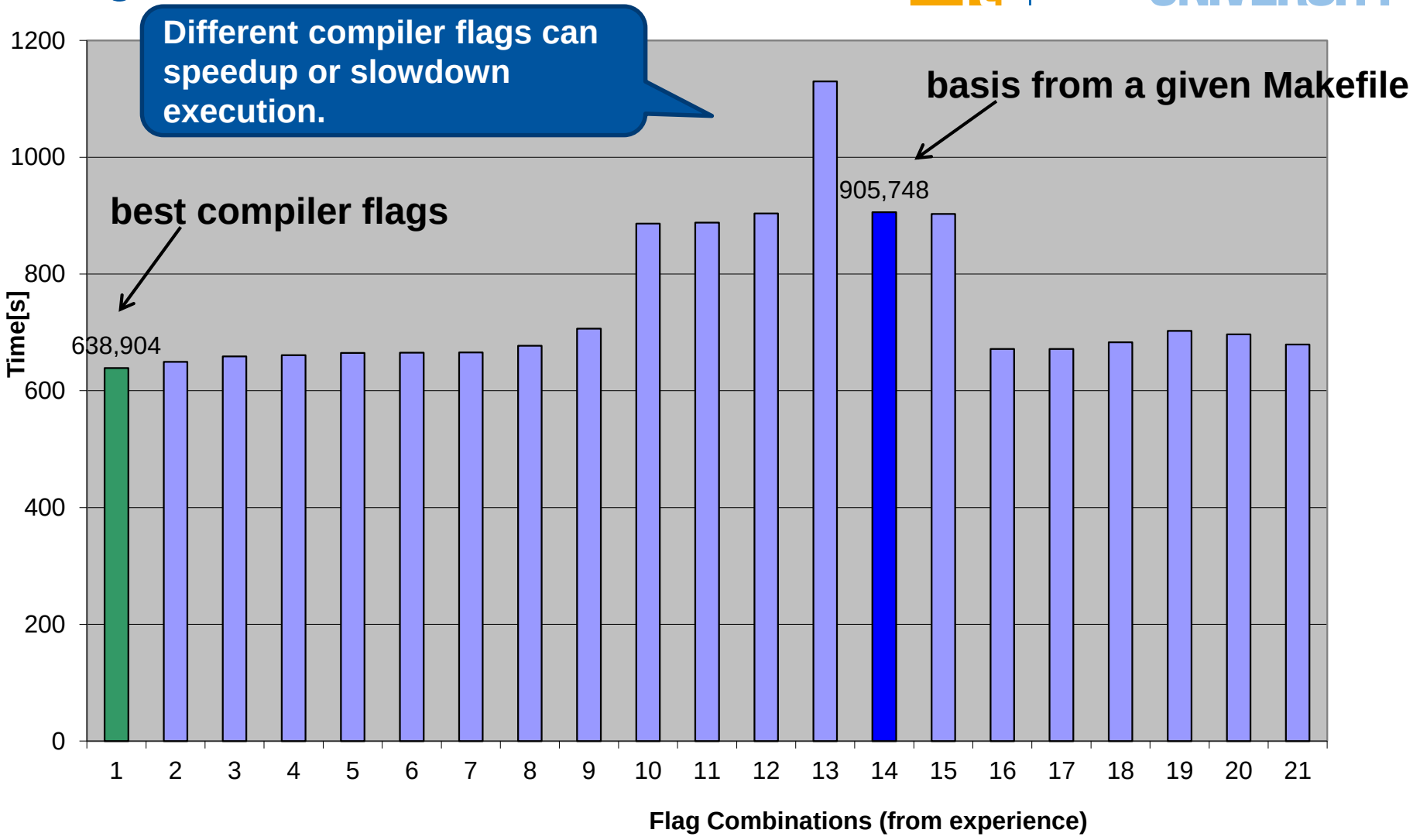
- Compiler switch may help.

Compilers are for different applications good or bad.

- There is not “the best compiler”.



Performance impact of compiler flags



Source: Ulf Markwardt, TU Dresden

General compiler optimization levels

- **Compilers offers a set of standard optimization options**
  - → -O0, -O1, …
- **Which optimizations are implied at which level is not standardized and may vary with different compilers**
  - → See the compiler manual for further information on that topic
- **Higher level optimizations may include mixing of source lines, elimination of redundant variables and operations, rearrangement of arithmetic expressions**
  - → Debugging is difficult if code is optimized
  - → For debugging purposes compile with –O0 (no optimizations at all)

Computational accuracy

- **Compiler may refrain from applying specific optimizations if this includes rearranging code in a way that may influence computational accuracy.**
  - → Level of rearrangements allowed can be specific by command-line option in modern compilers

- **Floating point denormals (numbers that fall below machine precision) handling may have a significant influence on performance too. If possible, apply flush-to-zero handling by hardware.**

Register optimizations

- **Register usage**
  - → One of the most vital tasks a compiler has to do
  - → Registers are fastest operands, but limited

- **Compiler puts operands that are used most often into registers**
  - → Keeps them there as long as possible
  - → If too few registers are available to hold all operands, some need to be written back to memory (spilling)

- **Inlining can help with register optimizations**
  - → Operands that reside in registers may not have to be written back to memory on behalf of the function call

Compiler logs

- **Compiler logs**
  - → Compilers offer options to generate annotated source code listings or logs that describe the optimizations that could be applied in more detail
  - → Supplies additional information about register usage and spilling, superscalar operations, pipeline utilization and speculative executions.

- **For more detailed analysis the compiler can generate assembly listings**
  - → These may not always be easy to compare against the sourcecode they originated from

Outline

1. Why supercomputers?
2. Modern processors
3. **Basic optimization techniques for serial code**
   - → Motivation for serial code tuning
   - → Monitoring
     - → Event- & sample-driven triggers
       - → Overview
       - → Basic block counting
       - → Instrumentation
       - → PC sampling
     - → Profiling & tracing
     - → Hardware performance counters
     - → Overview of tools
   - → Evolution of performance aspects
   - → **Common sense optimizations**
     - → Do less work
     - → Avoid expensive operations
     - → Shrink the working set
     - → Eliminate common subexpressions
     - → Avoid branches
     - → Use SIMD instruction sets
     - → Compiler impact
     - → **C++ optimizations**
4. Data access optimization
5. Parallel computers
6. Parallelization and optimization strategies
7. Parallel algorithms
8. Distributed-memory programming with MPI
9. Shared-memory programming with OpenMP
10. Hybrid programming (MPI + OpenMP)
11. Heterogeneous architectures (GPUs, Xeon Phis)
12. Energy efficiency

Allocation/ Deallocation

- **Avoid frequent allocation/deallocation whenever possible**
  - → Delay construction of objects until the moment they are actually needed:

```
void f( double epsilon, int N )
{
  std::vector v(N);
  if( rand() > epsilon*RAND_MAX )
  {
    v = obtain_data(N);
    process_data(v);
  }
}
```

Better: →

```
void f( double epsilon, int N )
{
  if( rand() > epsilon*RAND_MAX )
  {
    std::vector v(obtain_data(N));
    process_data(v);
  }
}
```

Allocation/ Deallocation

- **Use static construction if**
  - → working set size is constant
  - → allocation is only done once

```
const  int N = 10000;

void f( double epsilon )
{
  std::vector v(N);
  if( rand() > epsilon*RAND_MAX )
  {
    v = obtain_data(N);
    process_data(v);
  }
}
```

→

```
const  int N = 10000;

void f( double epsilon )
{
  static std::vector v(N);
  if( rand() > epsilon*RAND_MAX )
  {
    v = obtain_data(N);
    process_data(v);
  }
}
```

→ In multithreaded applications static data has to be taken special care of

Allocation/ Deallocation

- **Preallocation may be beneficial for performance:**
- **This may be combined with static variables…**

```
void f(int max)
{
    for(int N=0; N < max; ++N)
    {
      char *data = new char[N];
      obtain_data(data, N);
      process_data(data, N);
      delete []data;
    }
}
```

```
void f(int max)
{
  int cursz = 1000;
  char *data = new char[cursz];
  for(int N=0; N < max; ++N)
  {
      if( N > cursz )
      {
        cursz = N*2;
        delete []data;
        data = new char[cursz];
      }
      obtain_data(data, N);
      process_data(data, N);
  }
  delete []data;
}
```

- **Take care of cleaning up the allocated memory when you are using static vars or preallocation**

Abstractions

- **High level abstractions may prevent the compiler from applying the appropriate optimization schemes:**

```
double scalar_product( std::vector<double> a, std::vector<double> b)
{
   double result = 0;
   assert( a.size() == b.size() );
   int size = a.size();
   for( int i=0; i < size; ++i )
      result += a[i] + b[i];
}
```

- **Example**
  - → Overloaded index operator of std::vector is used to refer to individual elements
  - → Compiler may refuse applying SIMD vectorization to the above loop because of the single layer of abstraction
- **Instead fall back to using raw pointers to the working set.**

Abstractions

- **Example using pointers:**

```
double scalar_product( std::vector<double> a, std::vector<double> b)
{
  double result = 0;
  assert( a.size() == b.size() );
  int size = a.size();
  double *da = a.data();
  double *db = b.data();
  for( int i=0; i < size; ++i )
     result += *(da++) + *(db++);
}
```

- **Optimization is much easier from the compilers point of view**
- **If you decide on using high-level abstractions, check for the compiler to inline operator overloading, etc. correctly**

Quiz: What you have learnt

Question 1: What does Moore's law state?

The

a) CPU performance
b) memory size
c) CPU clock speed
d) number of transistors per chip

doubles every 12-24 months.

Quiz: What you have learnt

Question 1: What does Moore's law state?

d) The number of transistors per chip doubles every 12-24 months.

Quiz: What you have learnt

Question 2: Given are the following caches with their latency T_{lat}, bandwidth BW and length of their cache lines C_L. Which of the following caches delivers the best overall transfer time *T* for transferring 8 bytes?

a) $T_{lat} = 100\ ns$, BW = 4 GB/s, $C_L = 64$ byte

b) $T_{lat} = 150\ ns$, BW = 100 GB/s, $C_L = 64$ byte

c) $T_{lat} = 100\ ns$, BW = 4 GB/s, not organized in cache lines

d) $T_{lat} = 70\ ns$, BW = 2 GB/s, $C_L = 64$ byte

Quiz: What you have learnt

Question 2: Given are the following caches with their latency T_{lat}, bandwidth BW and length of their cache lines C_L. Which of the following caches delivers the best overall transfer time *T* for transferring 8 bytes?

d) $\boldsymbol{T} = \boldsymbol{T_{lat}} + \frac{\boldsymbol{N}}{\boldsymbol{B}} = 70\text{ ns} + \frac{\mathbf{64\ byte}}{\mathbf{2\ GB/s}} = \mathbf{99.80\ ns}$

Quiz: What you have learnt

Question 3: What is the cache hit rate β of the following code?

```
double a[64];
double tmp = 0;
for (int i=0; i<64; ++i) {
  tmp += a[i];
}
```

a) $\beta = 0$

b) $\beta = 1$

c) $\beta = \frac{7}{8} = 0.875$

d) $\beta = \frac{63}{64} = 0.984$

Assumptions:

- → size of the cache lines is 64 Bytes,
- → a double value has a size of 64-bit
- → cold cache (empty)
- → no compiler optimizations are applied
- → *tmp* is hold in a register
- → accesses to *i* are not evaluated

Quiz: What you have learnt

Question 3: What is the cache hit rate β of the following code?

```
double a[64];
double tmp = 0;
for (int i=0; i<64; ++i) {
  tmp += a[i];
}
```

$$c)\ \beta = \frac{7}{8} = 0.875$$

Assumptions:

- → size of the cache lines is 64 Bytes,
- → a double value has a size of 64-bit
- → cold cache (empty)
- → no compiler optimizations are applied
- → *tmp* is hold in a register
- → accesses to *i* are not evaluated

Quiz: What you have learnt

Question 4: Given is a direct-mapped cache with 4 frames. The following accesses to the main memory are observed:

#1, #3, #2, #1, #7, #3, #8, #2, #7, #4

What is the cache hit rate β of this sequence?

a) $\beta = 0$

b) $\beta = 1$

c) $\beta = \frac{2}{10}$

d) $\beta = \frac{4}{10}$

Assumptions:

→ no compiler optimizations are applied

→ cold cache (empty)

Quiz: What you have learnt

Question 4: Given is a direct-mapped cache with 4 frames. The following accesses to the main memory are observed:

#1, #3, #2, #1, #7, #3, #8, #2, #7, #4

What is the cache hit rate β of this sequence?

$$c)\ \beta = \frac{2}{10}$$

Frame 0	Frame 1	Frame 2	Frame 3
	#1		
			#3
		#2	
	#1		
			#7
			#3
#8			
		#2	
			#7
#4			

ons are