Analysis and modeling of

Computational Performance

Lecture 10

Single node performance optimization

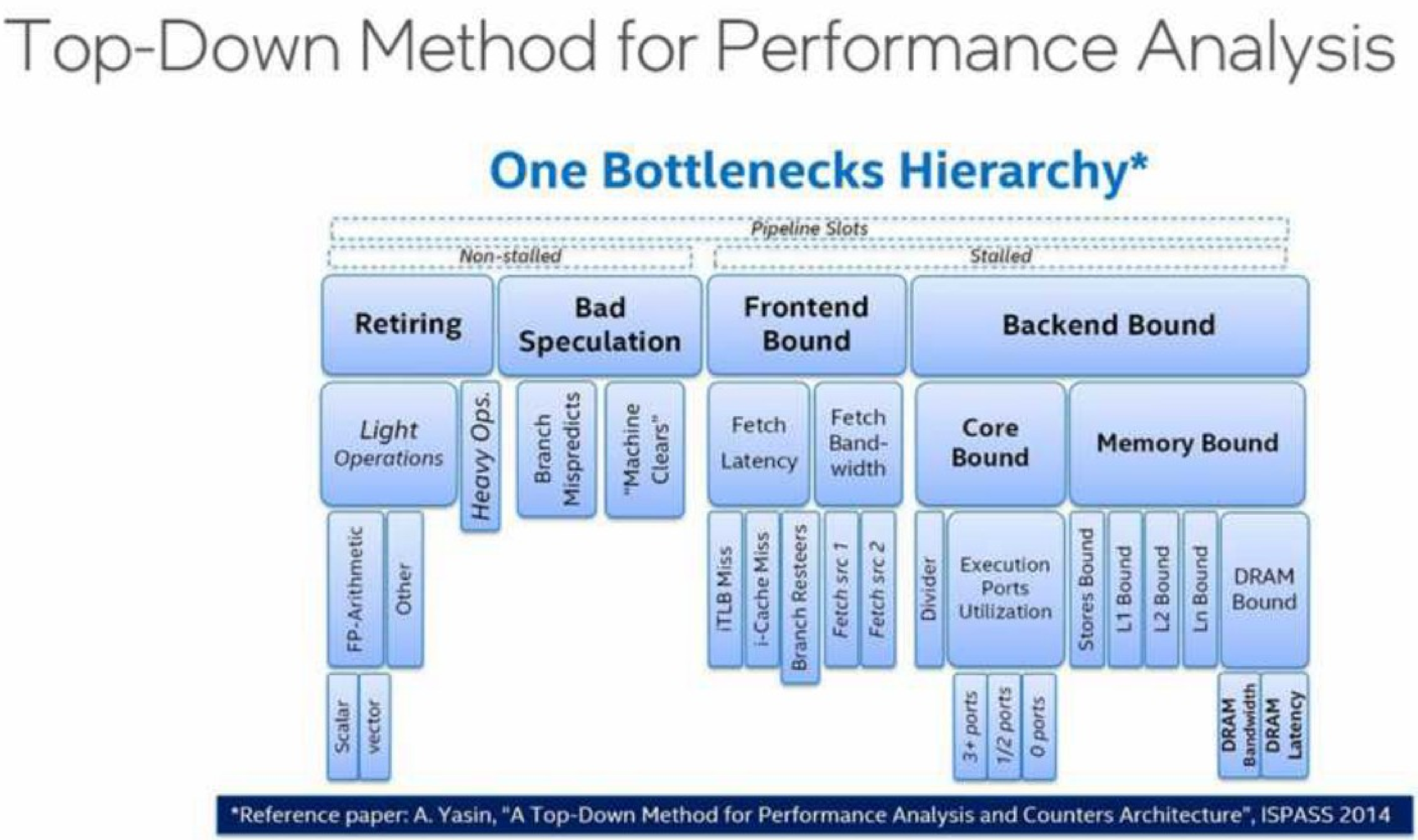
Software optimization

- Software optimization can have several goals:
  - minimization of execution time
    - the only one we are interested in, further called just optimization
  - minimization of memory footprint
  - other requirements, often depending on the particular software type or domain of application
- Optimization can be performed by different means at different stages of software development
  - by properly choosing algorithms and data structures while designing codes
    - depends on the domain of application
  - by proper implementation at the stage of source code creation
    - the main concern today is exploitation of parallel capabilities
    - even scalable software should have high single node performance
  - by using optimizing compiler
  - by the use of hardware designed for performance

Software optimization

- Software optimization is often blamed for being an obstacle for proper code development
  - Donald Knuth: "“Premature optimization is the root of all evil”
    - but the full quote includes “The real problem is that programmers have spent far too much time worrying about efficiency **in the wrong places and at the wrong times**; ..."
  - Performance optimization have to be done for the code that works
    - however, in order to give optimization a chance to improve the performance, the code has to be designed from the beginning with the future performance optimization in mind
  - Often employed strategy
    - predict the places most important from the performance point of view
    - separate the related code, create working version of the program
    - perform optimization, by **removing "bottlenecks"**
      - bottleneck is a place that cause performance degradation for a particular code or even particular case of input data

Software optimization



Software optimization

- The prediction of places most important from the performance point of view can be based on the analysis of the number of instructions and memory accesses done in a given part of the code
  - the parts of the code with the highest percentage of expected execution time are called "hot spots"
  - optimizing "hot spots" may be the most effective way for performance improvement
  - "hot spots" often become performance "bottlenecks"
    - it is also possible that a bottleneck appears in a place where relatively few operations are performed but these operations are (or become in certain circumstances) extremely slow
      - e.g. swapping or other secondary storage (hard disk or SSD) access, slow network connection, etc.
  - we will be mainly concerned with "hot spot" optimization, but will keep in mind that code profiling and bottleneck discovery should be the first step in optimization for a particular code

Software optimization

- The optimization should concern parts of the code most important from the performance point of view
  - "hot spots" can be identified through algorithm and source code analysis
  - "bottlenecks" can be found by profiling
- After separating the code related to the performance, different actions can be performed:
  - a proper high performance library can be found that provides functions necessary for code implementation
    - e.g. many linear algebra packages, with LAPACK being a prominent example, are successfully used in numerous programs
    - using libraries creates dependencies that may become problematic during code evolution
  - optimization can be performed for the code
    - the optimization usually depends on target execution environment and hardware, creating less portable code

Software optimization

- How to optimize a part of the code:
  - use optimizing compiler
  - perform manual optimization
    - contemporary optimizing compilers are doing their job very well
    - it is difficult to obtain by changing the source the same effect as by the use of an optimizing compiler
      - without optimization options compilers often produce unnecessarily slow code (e.g. for debugging purposes)
    - the best way for manual optimization is to apply specific techniques that help compilers to produce more effective code
      - allow for reducing the number of operations, effectively using different instruction pipelines, removing dependencies, choosing proper functions and instructions, vectorizing code, optimally use memory hierarchy
  - use a different programming language, designed for performance
    - eventually employ assembler language

Single node performance optimization

- Summary of techniques, important points, pitfalls to avoid
  - increase data locality and optimize memory access patterns
    - for reducing the number of memory accesses and better cache utilization
      - use e.g. cache blocking, register blocking
      - minimize the number of TLB misses
    - for better use of NUMA memories
      - use proper data placement together with thread affinity control
  - avoid memory contention (mapping different data to the same cache line, cache block, memory bank, etc.)
    - array sizes being the power of two
      - use padding
    - avoid false sharing
  - reduce pipeline stalls, caused e.g. by
    - data dependencies
    - indirect addressing
    - function calls, conditional statements (especially inside loops)

Single node performance optimization

- Summary of techniques, important points, pitfalls to avoid
  - allow optimizing compilers to work efficiently
    - remove aliases
      - inform compilers using suitable options or directives
    - allow for vectorization (remove dependencies)
  - use special memory allocation with proper alignment
  - perform classical optimizations that are not done by the compiler
    - reduce the number of operations in the algorithm, increase locality, etc.
    - especially when the task is too complex for the compiler
  - reduce system overhead when possible
    - do not allow for major page faults
  - allow hardware to effectively employ branch prediction, hardware prefetching, hardware multithreading, out-of-order execution, etc.
    - when necessary use software prefetching
  - use compiler *intrinsics* or assembly code

Software optimization

- Steps in practical software optimization process
  - choose proper algorithm (with future performance in mind)
  - implement for functional requirements
    - and possibly some, other than performance, non-functional requirements (safety, security, reliability etc.)
  - compile with proper optimization options switched on and create execution profile and identify performance bottlenecks
  - use high performance library routines for bottlenecks
    - if exist and their use does not interfere with other code development goals and limitations (e.g. required independence of external libraries)
  - manually optimize parts of the code related to performance
    - always check the effects of modifications using different compilers and compiler options – inspect assembler and test execution time
  - use different language for kernel implementation
    - eventually employ assembler intrinsics or write assembly code
  - test the final performance – use profilers, hardware counters, etc.
    - compare with the peak performance of the executing hardware