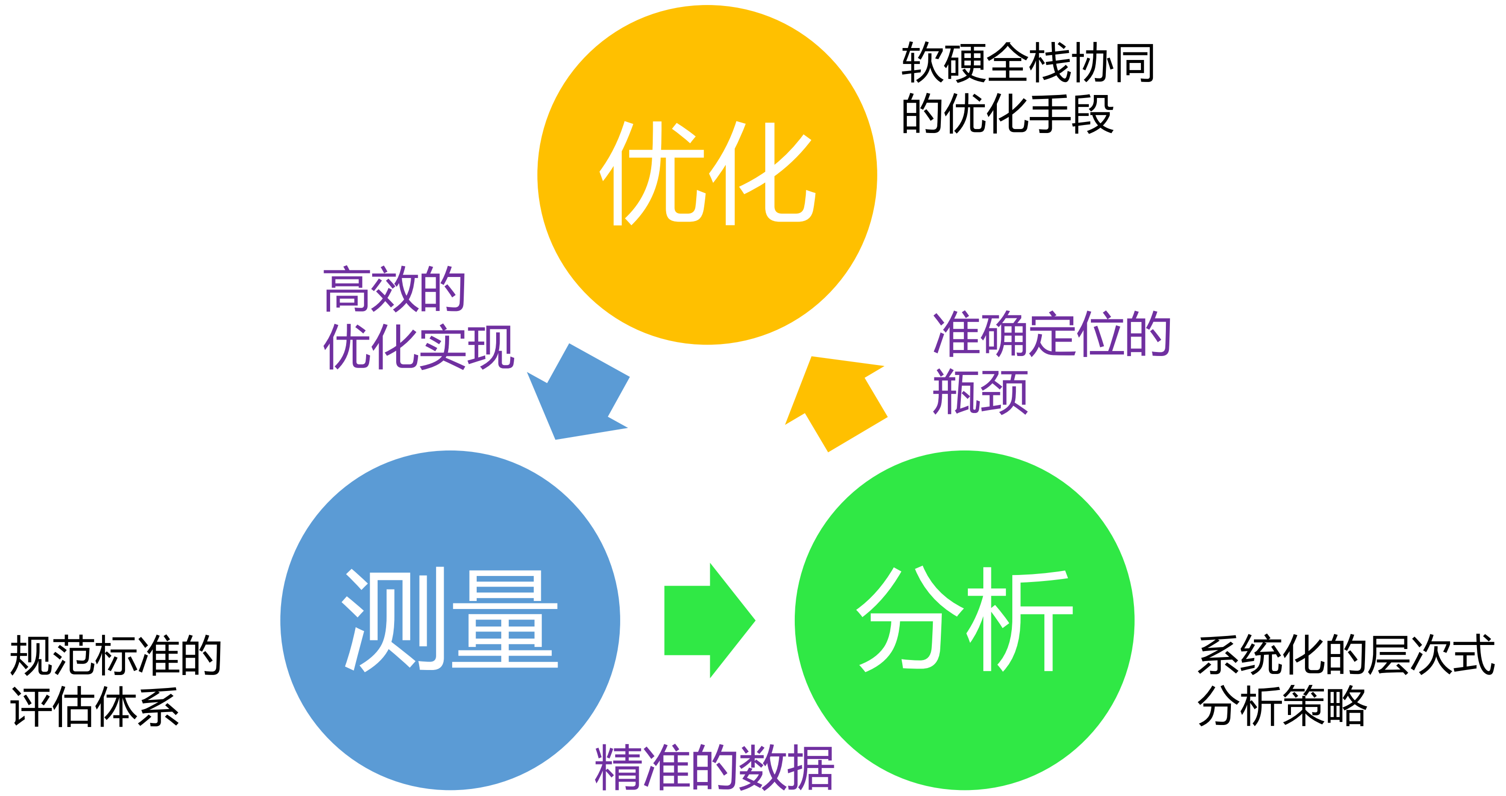
微架构性能分析方法

郭健美

2023年秋

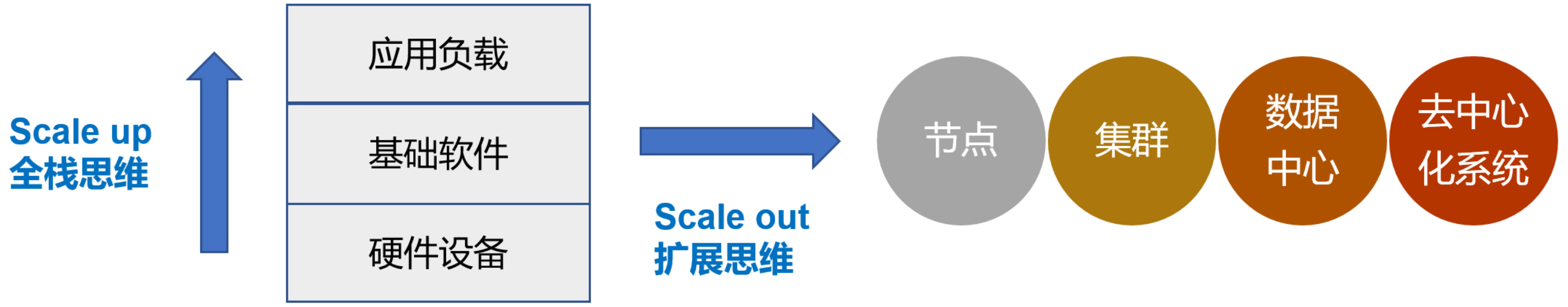
SOLE 系统优化实验室
华东师范大学

性能工程的基本流程


优化
软硬全栈协同
的优化手段
高效的
优化实现
准确定位的
瓶颈
测量
规范标准的
评估体系
精准的数据
分析
系统化的层次式
分析策略

性能分析方法论

- Scale up
  - Hennessy & Patterson, “Computer Architecture: A Quantitative Approach”: **CPU time’s Iron Law & CPI breakdown method**, 1990
  - Ahmad Yasin, “A Top-Down Method for Performance Analysis and Counters Architecture”: **TMAM**, ISPASS 2014
- Scale out
  - Google: “**Google-Wide Profiling (GWP)**, a continuous profiling infrastructure for data centers”, IEEE Micro, 2010
  - Google: “**WSMeter**: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers”, ASPLOS 2018
  - From SPEC Benchmarking to Online Performance Evaluation in Data Centers, LTB 2022



本课纲要：微架构性能分析

- **Iron law of processor performance**
- CPI breakdown method
- Top-down Microarchitecture Analysis Method (TMAM)

Iron Law of Processor Performance

Unfortunately, time is not always the metric quoted in comparing the performance of computers. Our position is that the only consistent and reliable measure of performance is the execution time of real programs, and that all proposed alternatives to time as the metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design.

Even execution time can be defined in different ways depending on what we count. The most straightforward definition of time is called *wall-clock time*, *response time*, or *elapsed time*, which is the latency to complete a task, including storage accesses, memory accesses, input/output activities, operating system overhead—everything. With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program. Thus we need a term to consider this activity. *CPU time* recognizes this distinction and means the time the processor is computing, *not* including the time waiting for I/O or running other programs. (Clearly, the response time seen by the user is the elapsed time of the program, not the CPU time.)

$$\textit{CPU time} = \text{Instruction count} \times \text{Clock cycles per instruction} \times \text{Clock cycle time}$$

$$\text{X is } n \text{ times faster than Y: } n = \text{Execution time}_Y \,/\, \text{Execution time}_X \;=\; \text{Performance}_X \,/\, \text{Performance}_Y$$

$$\textit{Amdahl's Law:}\ \text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

AWARDS & RECOGNITION

John Hennessy and David Patterson Receive 2017 ACM A.M. Turing Award

ACM has named John L. Hennessy, former President of Stanford University, and David A. Patterson, retired Professor of the University of California, Berkeley, recipients of the 2017 ACM A.M. Turing Award for pioneering a systematic, quantitative approach to the design and evaluation of computer architectures with enduring impact on the microprocessor industry.

[John L. Hennessy, David A. Patterson: Computer Architecture - A Quantitative Approach, 6th Edition. Morgan Kaufmann 2017.]
https://en.wikipedia.org/wiki/Iron_law_of_processor_performance

Iron Law of Processor Performance

$$\frac{CPU\ Time}{Program} = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instructions} \times \frac{CPU\ Time}{Clock\ Cycles}$$

[John L. Hennessy, David A. Patterson: Computer Architecture - A Quantitative Approach, 6th Edition. Morgan Kaufmann 2017.]
https://en.wikipedia.org/wiki/Iron_law_of_processor_performance

Iron Law of Processor Performance

$$\frac{CPU\ Time}{Program} = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instructions} \times \frac{CPU\ Time}{Clock\ Cycles}$$

- Increase clock cycles (Hardware)
  - Increase CPU frequency (but Moore’s Law & Dennard Scaling are ending)
  - Increase cores (but Amdahl’s Law still works)
  - Increase sockets (but expensive & addressing NUMA is not trivial)
  - Increase hardware threads (Intel HT or SMT, but counting clock cycles or computing CPU utilization is not trivial)

Iron Law of Processor Performance

$$\frac{CPU\ Time}{Program} = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instructions} \times \frac{CPU\ Time}{Clock\ Cycles}$$

- Reduce instruction path length (Software)
  - Better compilers
  - Profile-Guided Optimization (PGO)
  - Runtime optimization (recompilation and hot methods' inlining)
  - Reduce garbage collections
  - Rewrite source code (hopefully more efficient)

https://en.wikipedia.org/wiki/Instruction_path_length

Iron Law of Processor Performance

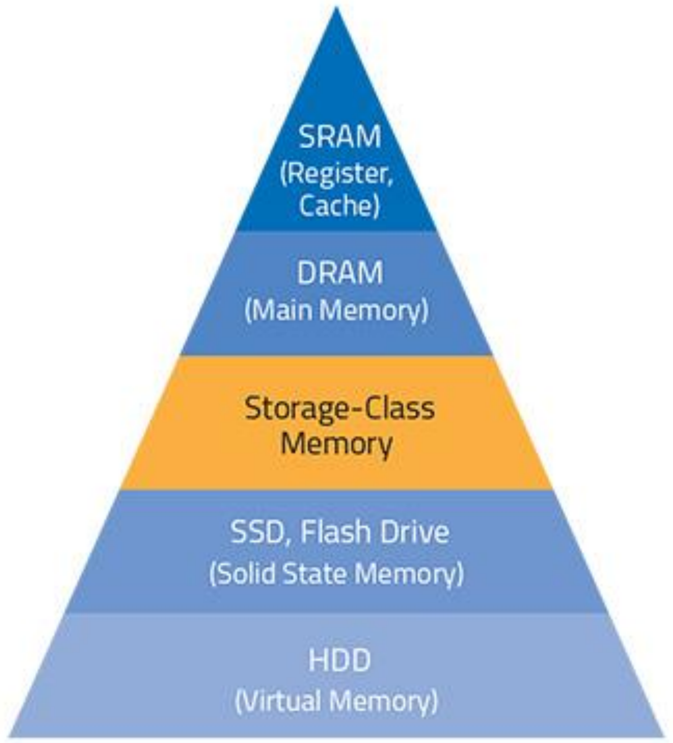
$$\frac{CPU\ Time}{Program} = \frac{Instructions}{Program} \times \frac{Clock\ Cycles}{Instructions} \times \frac{CPU\ Time}{Clock\ Cycles}$$

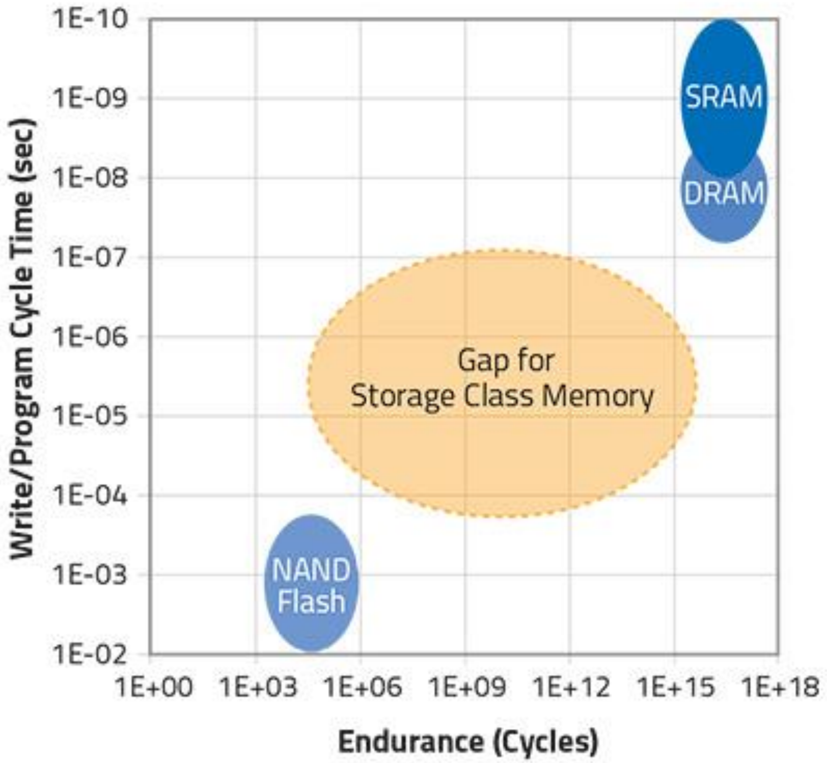
- Reduce CPI (Software+Hardware)
  - Advanced CPU designs
    - Bigger caches (but power increases) or better cache placements/alignment
    - Less or faster memory accesses (NVM/AEP, NUMA)
    - Better branch predictors
    - Better ITLB or DTLB (large pages 4KB, 2MB or 1GB)
    - Better prefetching
  - Better inlining (new compilers and JIT compilation in runtime)

本课纲要：微架构性能分析

- Iron law of processor performance
- **CPI breakdown method**
- Top-down Microarchitecture Analysis Method (TMAM)

CPI Breakdown Method





https://blog.lamresearch.com/tech-brief-abcs-of-new-memory/

CPI Breakdown Method

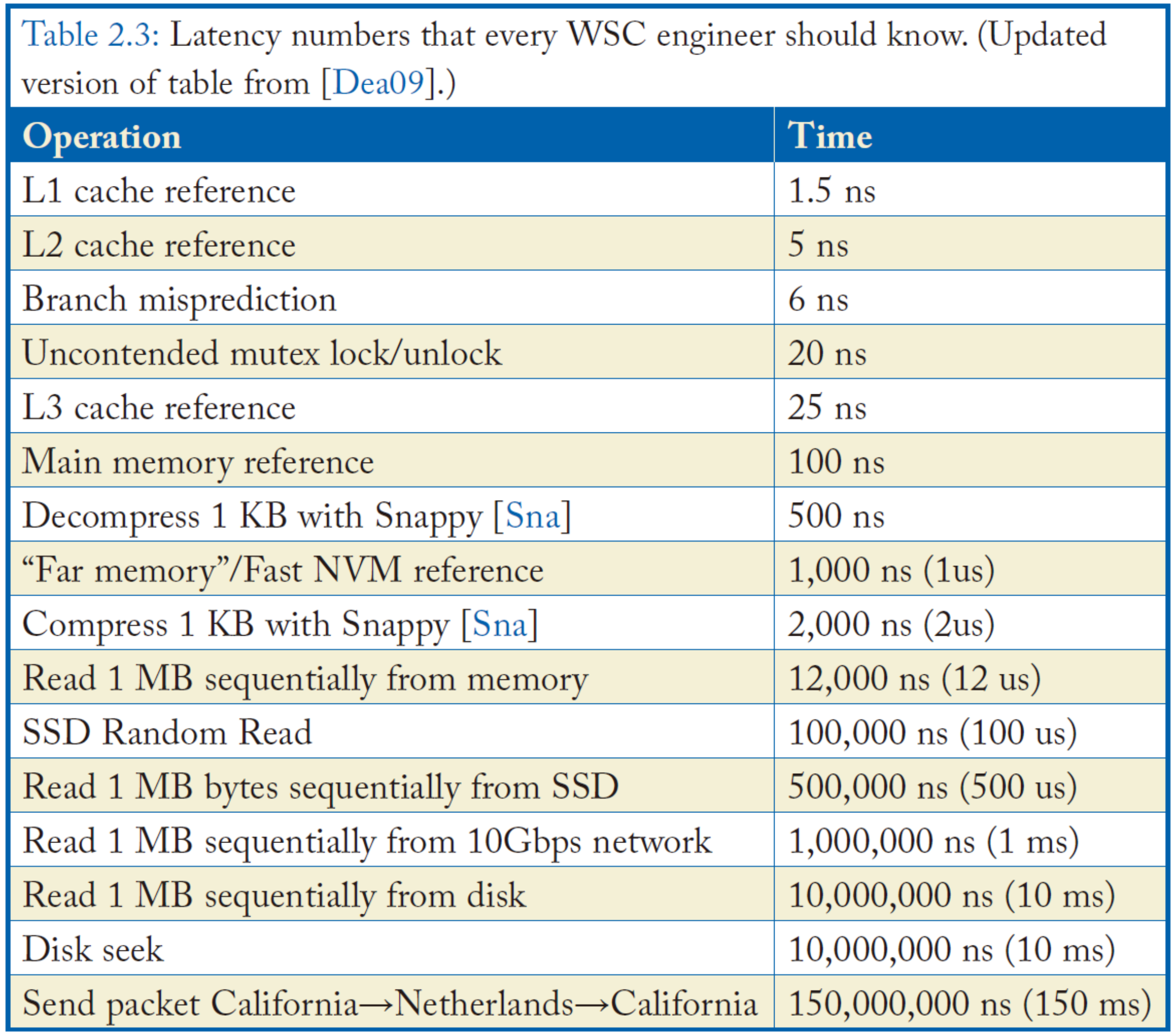


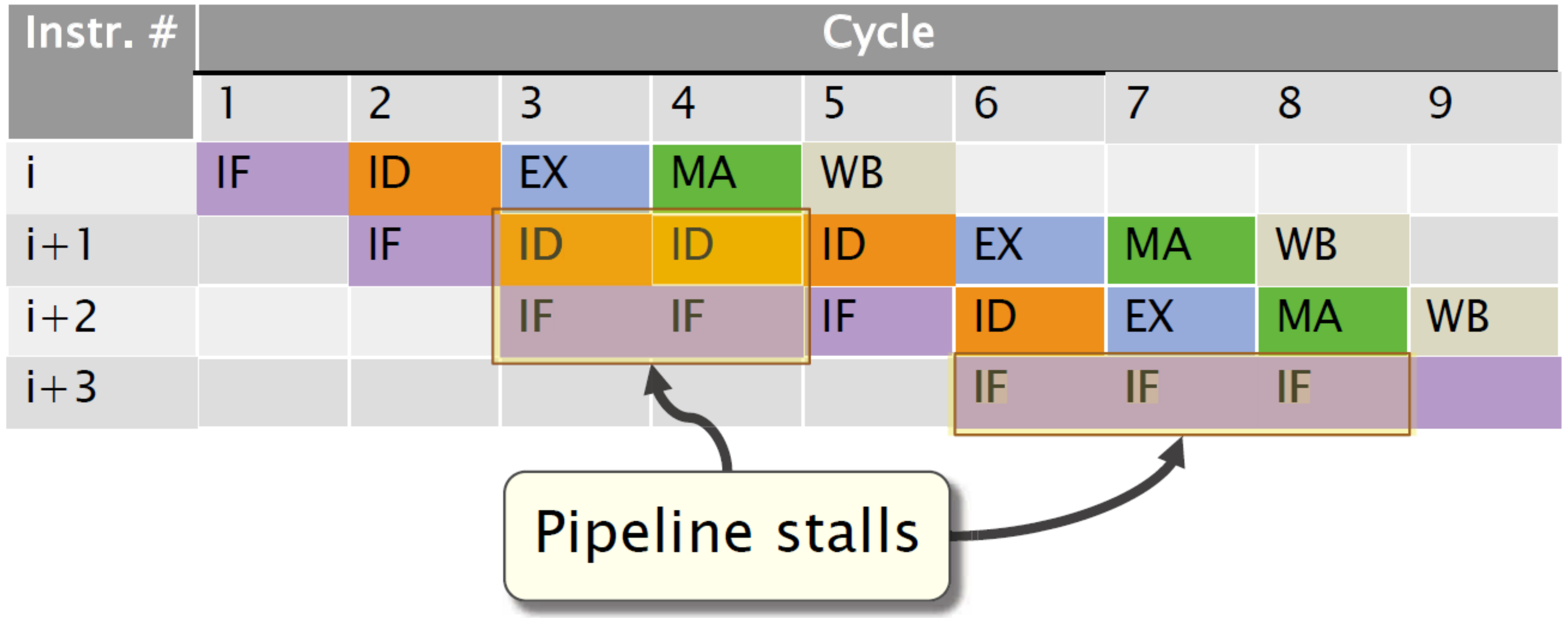
Table 2.3: Latency numbers that every WSC engineer should know. (Updated version of table from [Dea09].)

Operation	Time
L1 cache reference	1.5 ns
L2 cache reference	5 ns
Branch misprediction	6 ns
Uncontended mutex lock/unlock	20 ns
L3 cache reference	25 ns
Main memory reference	100 ns
Decompress 1 KB with Snappy [Sna]	500 ns
"Far memory"/Fast NVM reference	1,000 ns (1us)
Compress 1 KB with Snappy [Sna]	2,000 ns (2us)
Read 1 MB sequentially from memory	12,000 ns (12 us)
SSD Random Read	100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD	500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network	1,000,000 ns (1 ms)
Read 1 MB sequentially from disk	10,000,000 ns (10 ms)
Disk seek	10,000,000 ns (10 ms)
Send packet California→Netherlands→California	150,000,000 ns (150 ms)

[Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition. 2019.]

Pipelined Execution in Practice

In practice, various issues can prevent an instruction from executing during its designated cycle, causing the processor pipeline to stall.

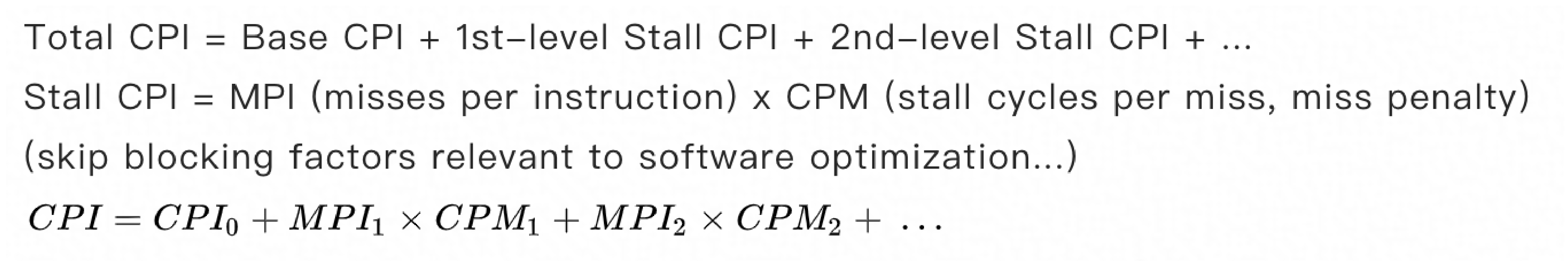


Instr. #	Cycle								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MA	WB				
i+1		IF	ID	ID	ID	EX	MA	WB	
i+2			IF	IF	IF	ID	EX	MA	WB
i+3						IF	IF	IF	

Pipeline stalls

CPI

CPI Breakdown Method



Total CPI = Base CPI + 1st–level Stall CPI + 2nd–level Stall CPI + ...

Stall CPI = MPI (misses per instruction) x CPM (stall cycles per miss, miss penalty)

(skip blocking factors relevant to software optimization...)

$$CPI = CPI_0 + MPI_1 \times CPM_1 + MPI_2 \times CPM_2 + \ \ldots$$

[David A. Patterson, John L. Hennessy. Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition). 2012.]

CPI Breakdown Method

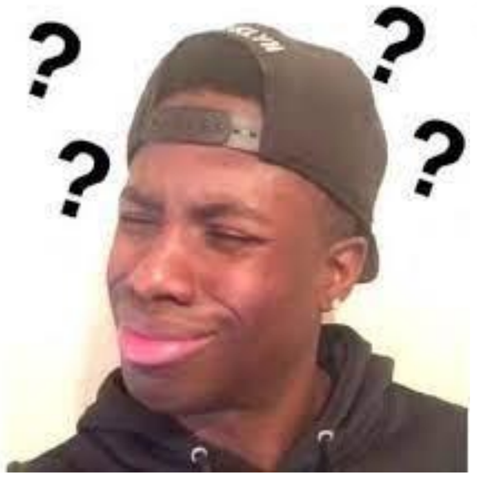
Total CPI = Base CPI + 1st–level Stall CPI + 2nd–level Stall CPI + ...

Stall CPI = MPI (misses per instruction) x CPM (stall cycles per miss, miss penalty)

(skip blocking factors relevant to software optimization...)

$$CPI = CPI_0 + MPI_1 \times CPM_1 + MPI_2 \times CPM_2 + \ \ldots$$

Limitation?



[David A. Patterson, John L. Hennessy. Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition). 2012.]

本课纲要：微架构性能分析

- Iron law of processor performance
- CPI breakdown method
- **Top-down Microarchitecture Analysis Method (TMAM)**

Top-down Microarchitecture Analysis Method (TMAM)

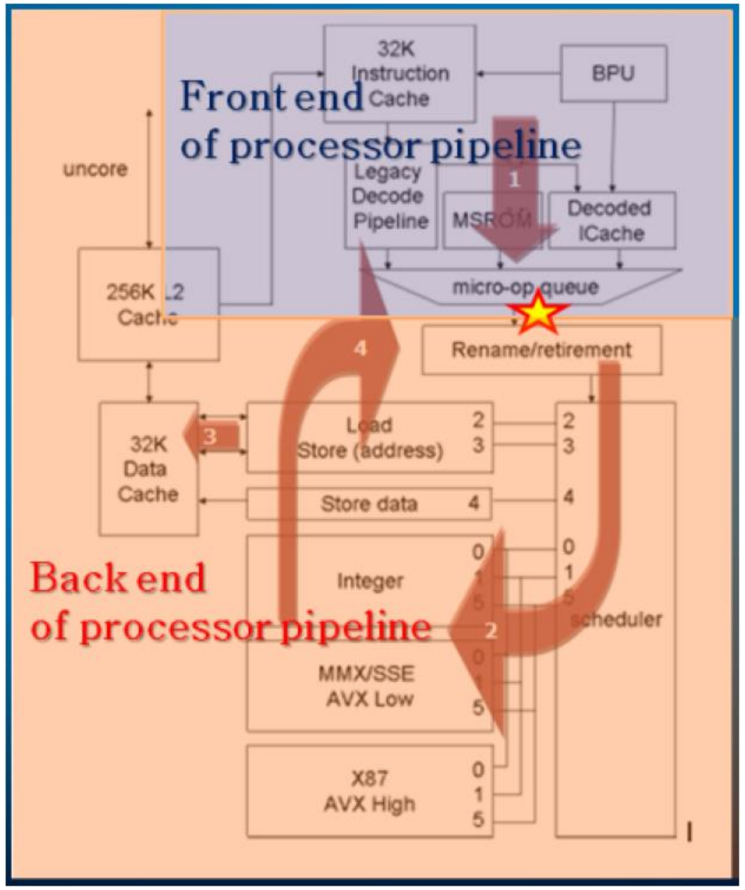


Figure 1: Out-of-order CPU block diagram - Intel Core™

[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]

Traditional methods [4][5] do simple estimations of stalls. E.g. the numbers of misses of some cache are multiplied by a pre-defined latency:

$$\texttt{Stall\_Cycles} = \Sigma\ \texttt{Penalty}_i * \texttt{MissEvent}_i$$

While this "naïve-approach" might work for an in-order CPU, surely it is not suitable for modern out-of-order CPUs due to numerous reasons: (1) *Stalls overlap*, where many units work in parallel. E.g. a data cache miss can be handled, while some future instruction is missing the instruction cache. (2) *Speculative execution*, when CPU follows an incorrect control-path. Events from incorrect path are less critical than those from correct-path. (3) *Penalties are workload-dependent*, while naïve-approach assumes a fixed penalty for all workloads. E.g. the distance between branches may add to a misprediction cost. (4) *Restriction to a pre-defined set of miss-events*, these sophisticated microarchitectures have so many possible hiccups and only the most common subset is covered by dedicated events. (5) *Superscalar inaccuracy*, a CPU can issue, execute and retire multiple operations in a cycle. Some (e.g. client) applications become limited by the pipeline's bandwidth as latency is mitigated with more and more techniques.

Top-down Microarchitecture Analysis Method (TMAM)

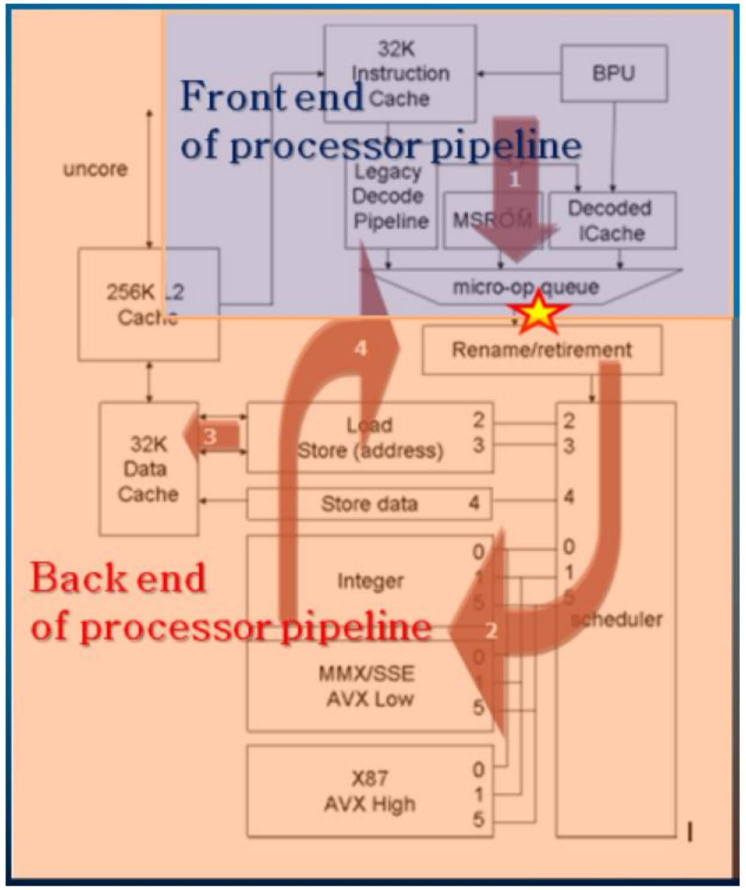


Figure 1: Out-of-order CPU block diagram - Intel Core™

For each pipeline-slot

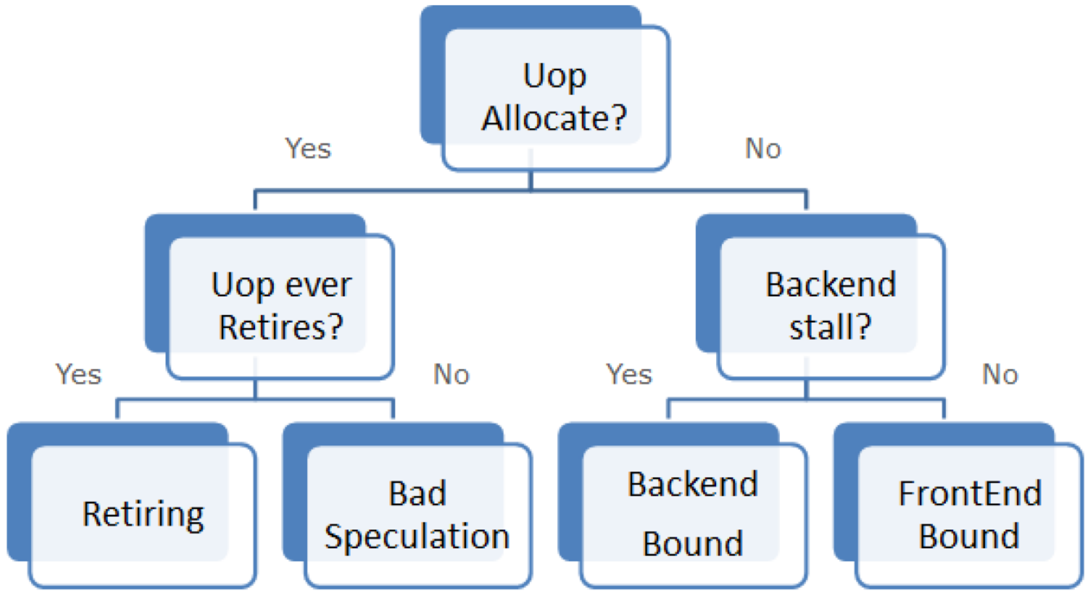


Figure 3: Top Level breakdown flowchart

[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]

Top-down Microarchitecture Analysis Method (TMAM)

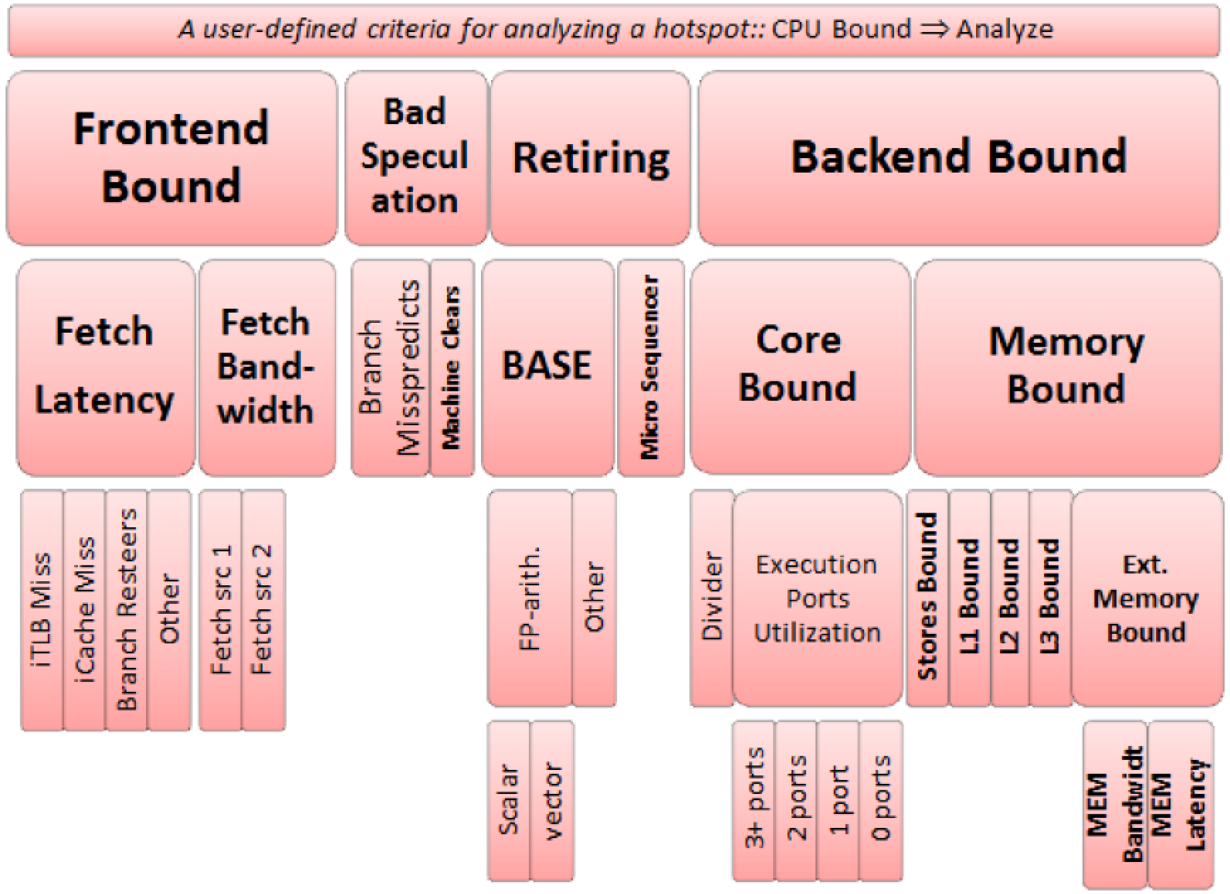
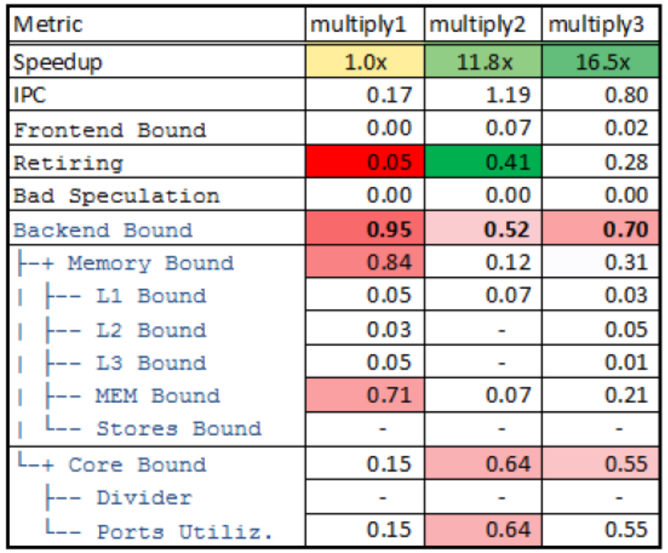


Figure 2: The Top-Down Analysis Hierarchy

[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]

Table 4: Results of tuning Matrix-Multiply case



Metric	multiply1	multiply2	multiply3
Speedup	1.0x	11.8x	16.5x
IPC	0.17	1.19	0.80
Frontend Bound	0.00	0.07	0.02
Retiring	0.05	0.41	0.28
Bad Speculation	0.00	0.00	0.00
Backend Bound	0.95	0.52	0.70
├-+ Memory Bound	0.84	0.12	0.31
| ├-- L1 Bound	0.05	0.07	0.03
| ├-- L2 Bound	0.03	-	0.05
| ├-- L3 Bound	0.05	-	0.01
| ├-- MEM Bound	0.71	0.07	0.21
| └-- Stores Bound	-	-	-
└-+ Core Bound	0.15	0.64	0.55
├-- Divider	-	-	-
└-- Ports Utiliz.	0.15	0.64	0.55

The initial code in `multiply1()` is extremely MEM Bound as big matrices are traversed in cache-unfriendly manner.

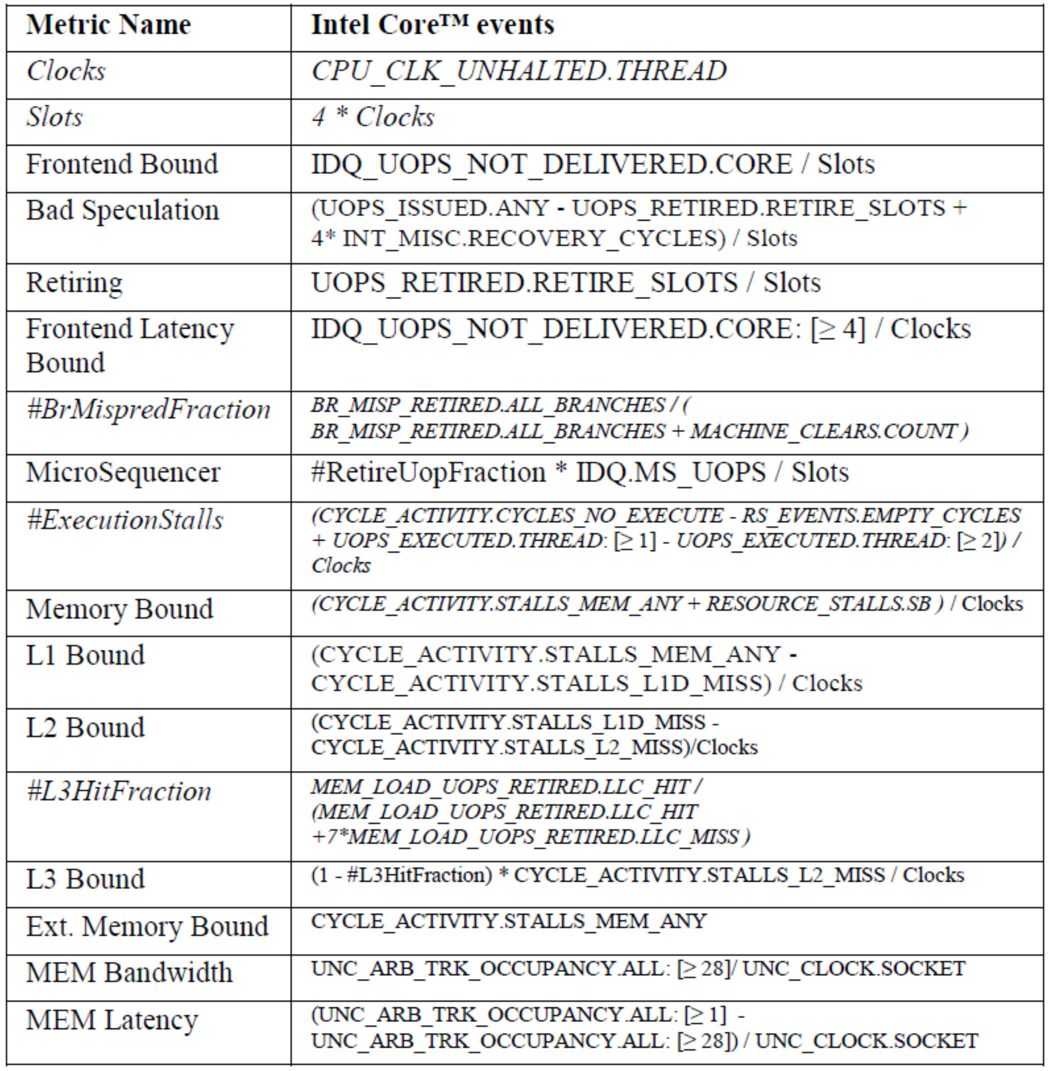
Loop Interchange optimization, applied in `multiply2()` gives big speedup. The optimized code continues to be Backend Bound though now it shifts from Memory Bound to become Core Bound.

Next in `multiply3()`, Vectorization is attempted as it reduces the port utilization with less net instructions. Another speedup is achieved.

Top-down Microarchitecture Analysis Method (TMAM)

- Tools
  - Intel VTune Profiler https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
  - PMU-tools https://github.com/andikleen/pmu-tools
  - 华为鲲鹏性能分析工具Hyper Tuner https://support.huawei.com/enterprise/zh/computing/hyper-tuner-pid-250633156

Table 7: Intel's implementation of Top-Down Metrics



Metric Name	**Intel Core™ events**
Clocks	*CPU_CLK_UNHALTED.THREAD*
Slots	*4 * Clocks*
Frontend Bound	IDQ_UOPS_NOT_DELIVERED.CORE / Slots
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4* INT_MISC.RECOVERY_CYCLES) / Slots
Retiring	UOPS_RETIRED.RETIRE_SLOTS / Slots
Frontend Latency Bound	IDQ_UOPS_NOT_DELIVERED.CORE: [≥ 4] / Clocks
#BrMispredFraction	*BR_MISP_RETIRED.ALL_BRANCHES / (BR_MISP_RETIRED.ALL_BRANCHES + MACHINE_CLEARS.COUNT)*
MicroSequencer	#RetireUopFraction * IDQ.MS_UOPS / Slots
#ExecutionStalls	*(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE - RS_EVENTS.EMPTY_CYCLES + UOPS_EXECUTED.THREAD*: [≥ 1] - *UOPS_EXECUTED.THREAD*: [≥ 2]) / *Clocks*
Memory Bound	*(CYCLE_ACTIVITY.STALLS_MEM_ANY + RESOURCE_STALLS.SB)* / Clocks
L1 Bound	(CYCLE_ACTIVITY.STALLS_MEM_ANY - CYCLE_ACTIVITY.STALLS_L1D_MISS) / Clocks
L2 Bound	(CYCLE_ACTIVITY.STALLS_L1D_MISS - CYCLE_ACTIVITY.STALLS_L2_MISS)/Clocks
#L3HitFraction	*MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT +7*MEM_LOAD_UOPS_RETIRED.LLC_MISS)*
L3 Bound	(1 - #L3HitFraction) * CYCLE_ACTIVITY.STALLS_L2_MISS / Clocks
Ext. Memory Bound	CYCLE_ACTIVITY.STALLS_MEM_ANY
MEM Bandwidth	UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 28]/ UNC_CLOCK.SOCKET
MEM Latency	(UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 1] - UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 28]) / UNC_CLOCK.SOCKET

[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]

Top-down Microarchitecture Analysis Method (TMAM)

Limitation?

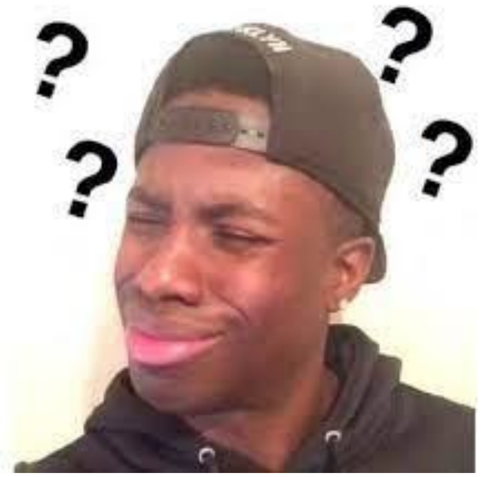
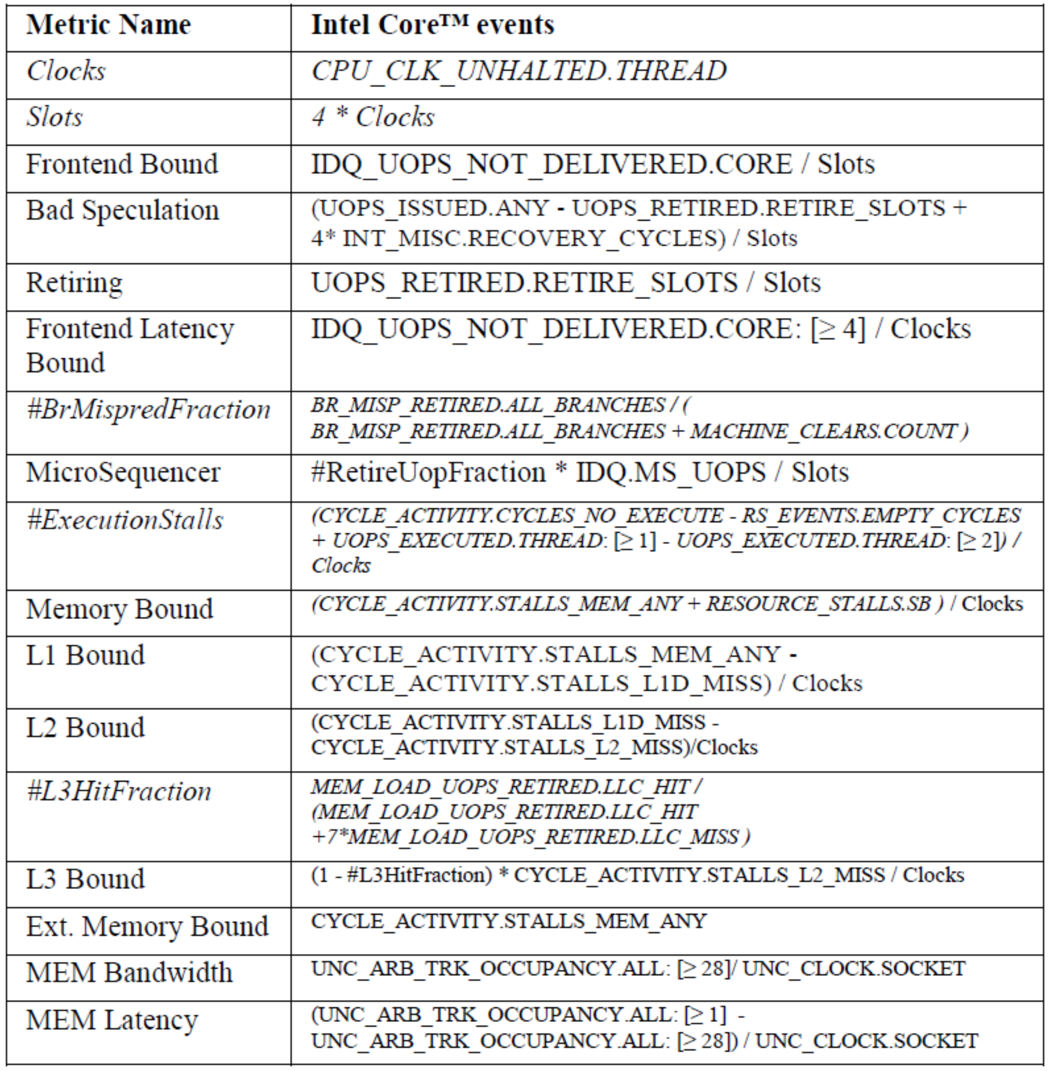


Table 7: Intel's implementation of Top-Down Metrics



Metric Name	**Intel Core™ events**
Clocks	*CPU_CLK_UNHALTED.THREAD*
Slots	*4 * Clocks*
Frontend Bound	IDQ_UOPS_NOT_DELIVERED.CORE / Slots
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + 4* INT_MISC.RECOVERY_CYCLES) / Slots
Retiring	UOPS_RETIRED.RETIRE_SLOTS / Slots
Frontend Latency Bound	IDQ_UOPS_NOT_DELIVERED.CORE: [≥ 4] / Clocks
#BrMispredFraction	*BR_MISP_RETIRED.ALL_BRANCHES / (BR_MISP_RETIRED.ALL_BRANCHES + MACHINE_CLEARS.COUNT)*
MicroSequencer	#RetireUopFraction * IDQ.MS_UOPS / Slots
#ExecutionStalls	*(CYCLE_ACTIVITY.CYCLES_NO_EXECUTE - RS_EVENTS.EMPTY_CYCLES + UOPS_EXECUTED.THREAD*: [≥ 1] - *UOPS_EXECUTED.THREAD*: [≥ 2]) / *Clocks*
Memory Bound	*(CYCLE_ACTIVITY.STALLS_MEM_ANY + RESOURCE_STALLS.SB)* / Clocks
L1 Bound	(CYCLE_ACTIVITY.STALLS_MEM_ANY - CYCLE_ACTIVITY.STALLS_L1D_MISS) / Clocks
L2 Bound	(CYCLE_ACTIVITY.STALLS_L1D_MISS - CYCLE_ACTIVITY.STALLS_L2_MISS)/Clocks
#L3HitFraction	*MEM_LOAD_UOPS_RETIRED.LLC_HIT / (MEM_LOAD_UOPS_RETIRED.LLC_HIT +7*MEM_LOAD_UOPS_RETIRED.LLC_MISS)*
L3 Bound	(1 - #L3HitFraction) * CYCLE_ACTIVITY.STALLS_L2_MISS / Clocks
Ext. Memory Bound	CYCLE_ACTIVITY.STALLS_MEM_ANY
MEM Bandwidth	UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 28]/ UNC_CLOCK.SOCKET
MEM Latency	(UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 1] - UNC_ARB_TRK_OCCUPANCY.ALL: [≥ 28]) / UNC_CLOCK.SOCKET

[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]