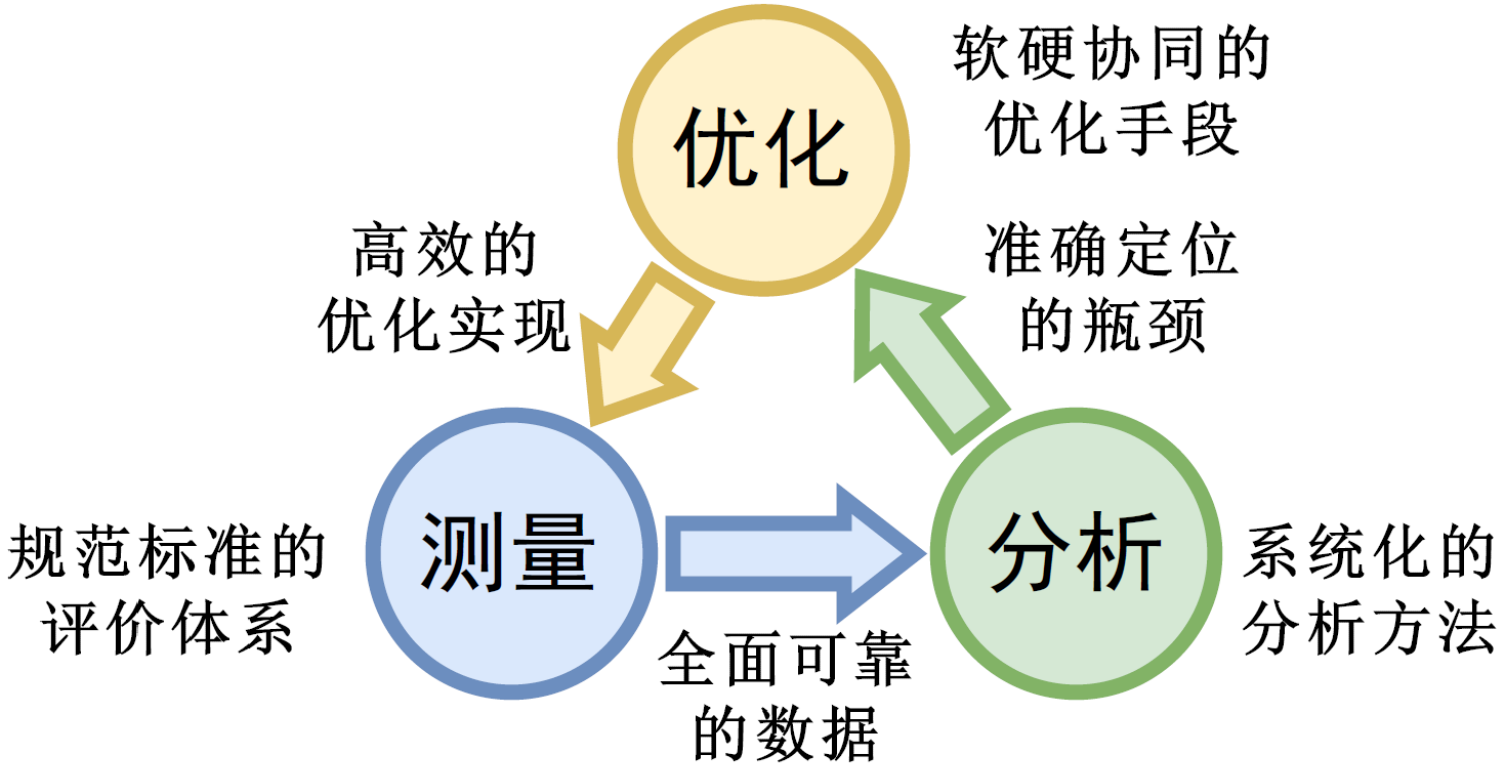
微体系结构性能分析

郭健美

2024年秋

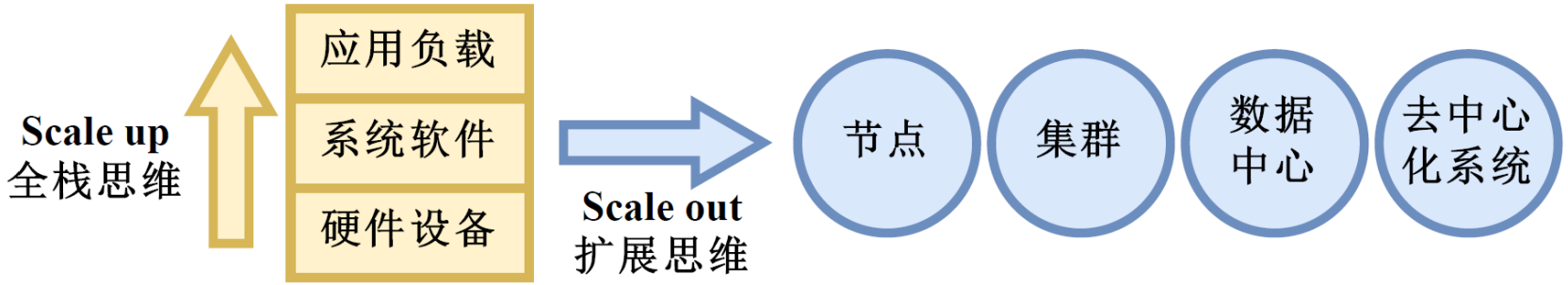
SOLE 系统优化实验室
华东师范大学

数据驱动的系统优化方法

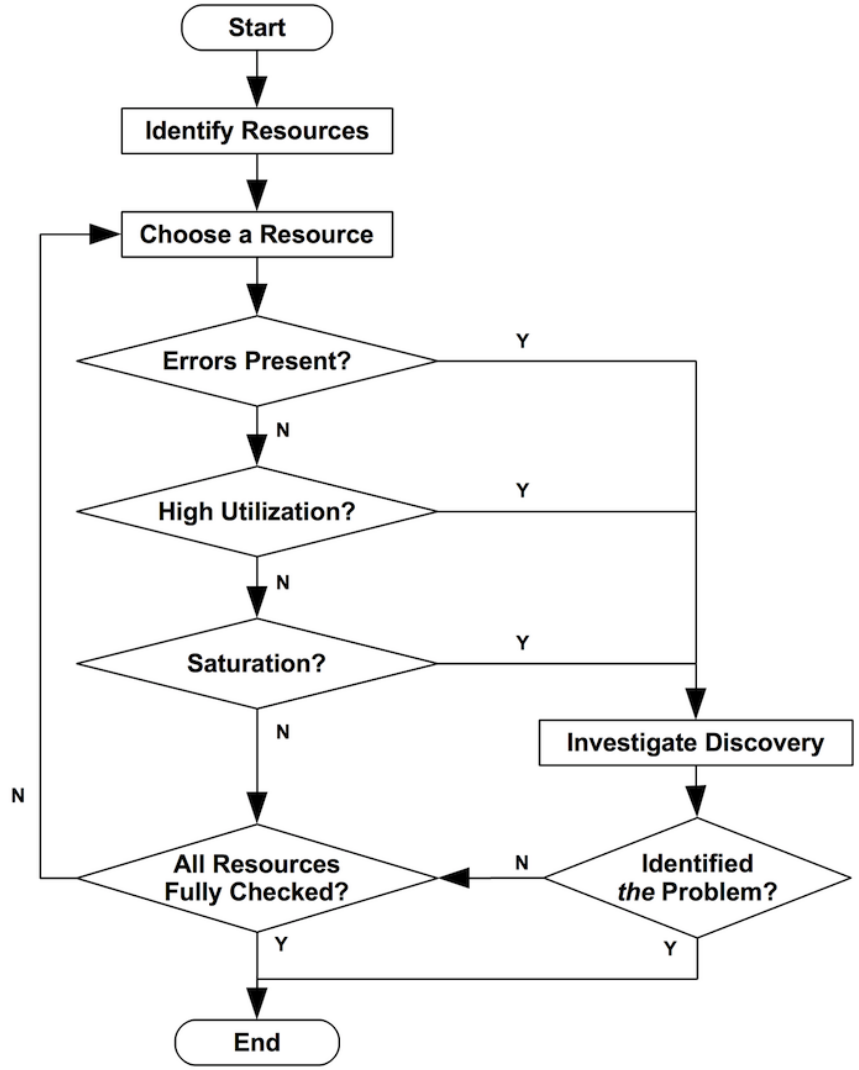


性能分析方法论

- Scale up
  - **Iron Law of processor performance & CPI breakdown method** : J. L. Hennessy & D. A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann, 1990
  - **TMAM** : A. Yasin, “A Top-Down Method for Performance Analysis and Counters Architecture”, ISPASS 2014
  - **USE** : B. Gregg. “Thinking methodically about performance”. Commun. ACM 56(2): 45-51 (2013)
- Scale out
  - **GWP**: G. Ren, et al., “Google-Wide Profiling: a continuous profiling infrastructure for data centers”, IEEE Micro, 2010
  - **WSMeter** : J. Lee, et al., “WSMeter: A Performance Evaluation Methodology for Google's Production Warehouse-Scale Computers”, ASPLOS 2018
  - **RUE / RCU**: J. Guo. “From SPEC Benchmarking to Online Performance Evaluation in Data Centers”, LTB 2022 Keynote.

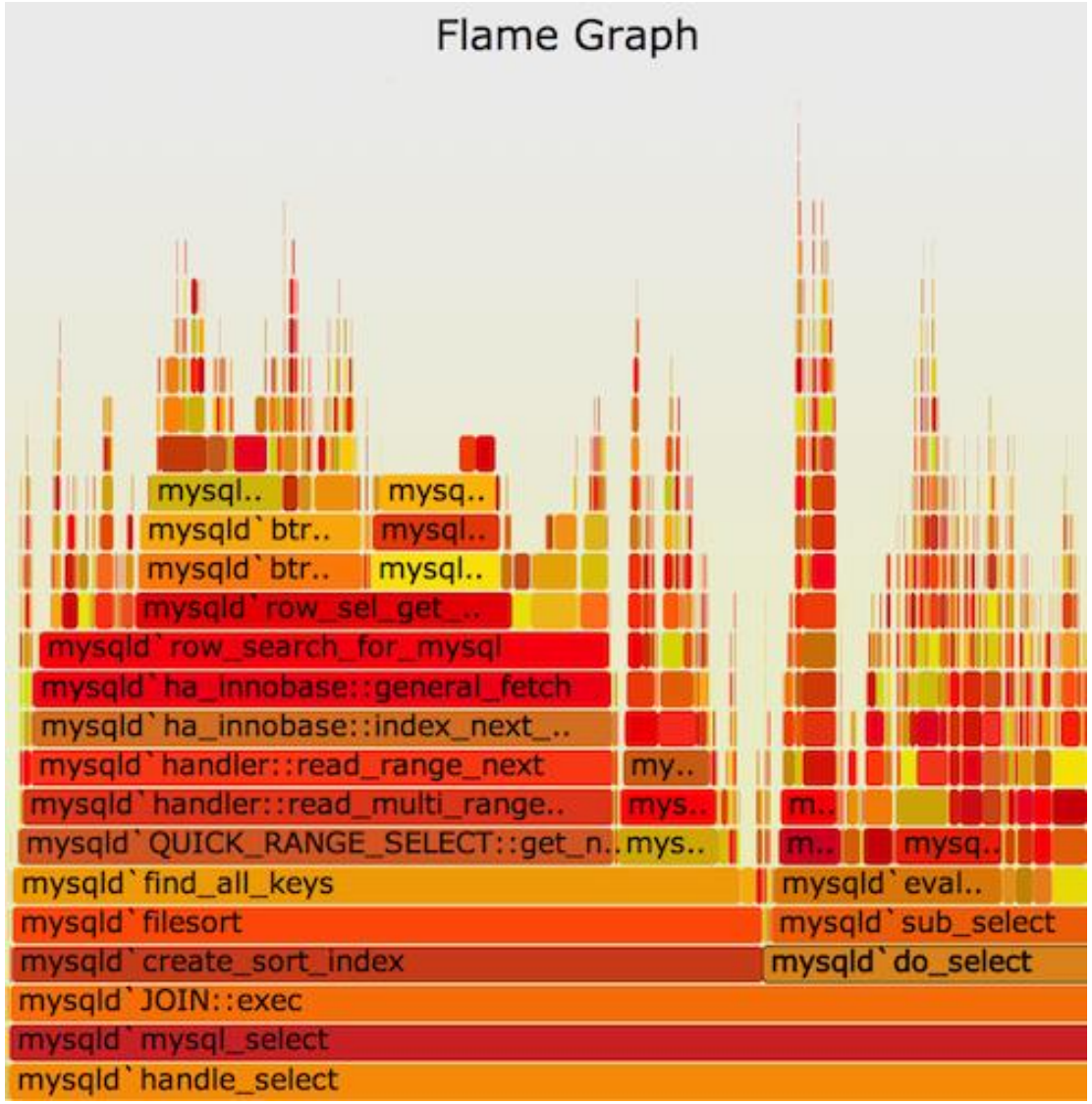


USE (Utilization Saturation and Errors) 方法



https://www.brendangregg.com/usemethod.html

调用栈的火焰图



https://www.brendangregg.com/flamegraphs.html

本课内容

- 处理器性能的铁律
- CPI 分解方法
- 自顶向下的微体系结构分析方法 TMAM

处理器性能的铁律

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.)

[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

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.

处理器性能的铁律

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 处理器性能取决于上述三个部分，任一部分的改进都会带来处理器执行程序的性能提升
- 然而，很难完全独立地改进一个部分而不影响其他部分，因为改进这三部分所涉及的技术方法是相互依赖的

[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

处理器性能的铁律

- 本质上，所有计算机都是用以一定速率运行的时钟构建的
- 时钟周期是计算机执行运算的最基本的、最小的时间单位，所有的运算操作都是由时钟周期这样的离散时间事件完成的

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

$$\text{A program's CPU time} = \frac{\text{Clock cycles}}{\text{Program}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

$$\text{A program's CPU time} = \frac{\text{Clock cycles}}{\text{Program}} \times \frac{1}{\text{Frequency}}$$

优化每时钟周期的时长

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 提高频率
  - 受限于摩尔定律和登纳德缩放定律逐渐失效

- 单位时间内提供更多时钟周期
  - 假设完成一个程序所需要的总时钟周期数是固定的，那么如果能在单位时间内提供更多时钟周期，则可以减少程序的CPU 时间，从而更快地完成程序

优化每时钟周期的时长

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 单位时间内提供更多时钟周期
  - 更多处理器核心
    - 受限于阿姆达尔定律、核间通信延迟
  - 更多处理器
    - 受限于多路（socket）实现的成本、跨路访问延迟、NUMA架构
  - 更多服务器
    - 受限于服务器间通讯延迟、集群调度和运维、故障恢复
  - 更多硬件线程（hardware threads）
    - 受限于硬件线程对物理核的资源竞争

优化指令路径长度

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 减少程序的**指令路径长度（Instruction Path Length, IPL）**
  - 采用更好的编译器和静态编译优化技术
    - 例如，常量折叠、死代码消除、循环不变量外提、控制流优化、数据流优化等
  - 采用**性能画像引导的优化（Profile-Guided Optimization, PGO）**等动态编译优化技术
    - PGO本身会引入一定的运行时开销、所采集现场的代表性可能有限
  - 减少垃圾回收（Garbage Collection, GC）
    - GC本身会引入额外的指令来标记和回收内存中的对象
  - 重写源代码
    - 慎用！高德纳（Donald Knuth）“Premature optimization is the root of all evil.”强调在优化前应该首先进行性能分析，了解程序的真正瓶颈所在，集中精力解决关键瓶颈问题，而不是过早地去优化代码

优化指令路径长度

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 注意：指令路径长度 IPL 与 每条指令的平均时钟周期数 CPI 之间存在一种**权衡**，减少 IPL 可能增加 CPI，因而仅仅考虑 IPL 的减少不一定会带来程序总体性能的提升

- 处理器性能的铁律将这种权衡明确定义了出来，即：
  程序性能优化应该考虑程序执行时间的整体优化，而不仅仅是考虑影响性能的单一部分的优化

优化每条指令的平均时钟周期数 CPI

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- CPI / IPC 是衡量处理器性能的重要指标
- 较低的CPI 或较高的IPC 通常表示处理器执行程序的效率较高，即可以用更少的时钟周期来完成更多的指令
- CPI 的优化既涉及时钟周期、又涉及指令路径长度。因此，CPI 的优化是软硬件一体的优化，不仅要考虑底层硬件和计算机组成，还要考虑指令集架构和编译技术等

优化每条指令的平均时钟周期数 CPI

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

- 流水线优化：更细分、更深的流水线
- 超标量执行：同一时钟周期内执行更多指令
- 分支预测优化：更高效的分支预测算法
- 高速缓存优化：增加缓存大小或更优的缓存置换/预取算法
- 内存访问优化：减少工作集的大小或采用更快的硬件（如，高带宽存储器 HBM ）
- 采用更高效的指令集：如，SIMD 指令或加密指令
- 使用硬件加速：如，GPU 或 FPGA

CPI 分解方法

CPI 之所以成为最流行的处理器性能指标之一

- 一个原因在于它容易量化处理器性能，既可以在流片前通过仿真来估计，也可以在流片后进行实际测量
- 另一个原因在于，CPI可以进一步分解，从更细粒度或从处理器各相关部件提供更多瓶颈分析和优化建议
  - **可分解**
  - **累加性**

CPI 分解方法

- 根据不同指令的 CPI 分解
- 根据不同停顿的 CPI 分解

根据不同指令的 CPI 分解

指令分类			x86-64 平台	AArch64 平台
分支	间接分支	跳转	jmp<mem>, <reg>)	br
		函数调用	call(<mem>, <reg>)	blr
		返回	ret	ret
	条件跳转		jle / jne / je / ja /jna ...	b.<cond> / cbz / cbnz ...
	无条件直接跳转		jmp / call <imm>	b / bl
运算	算术		cmp / sub / mul ...	cmp / sub / mul / add ...
	逻辑		and / or / xor / test ...	and / or ...
	移位		shr / sha / sar / sal	asrv / rorv / lslv / bfm ...
数据传送			mov / movzx ...	movi / movz / fmov ...
栈操作			push / pop	–
内存读写			–	str / stp / ldr / ldp
向量化指令			SSE / AVX / AVX2 / AVX512	Neon / SVE
其它			lea / setb / nop ...	adrp / adr / nop / csel ...

根据不同指令的 CPI 分解

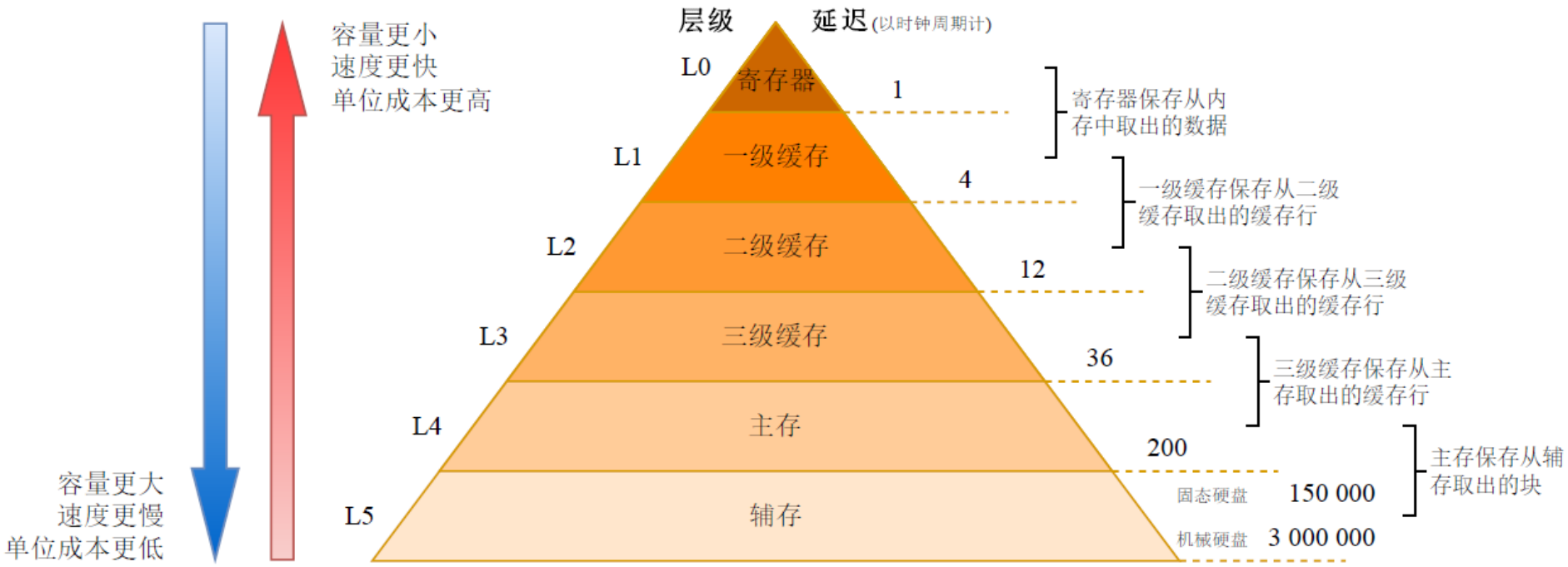
- 一个程序的执行是由不同类型指令的执行构成的，同时，不同类型的指令在特定处理器上执行时所需要的时钟周期数也是不同的

$$\text{A program's CPU time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Clock cycle}}$$

$$\text{A program's CPU time} = \left(\sum_i \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

$$\text{Total CPI} = \frac{\text{Total clock cycles}}{\text{Total instructions}} = \frac{\sum_i \text{IC}_i \times \text{CPI}_i}{\text{IC}} = \sum_i \frac{\text{IC}_i}{\text{IC}} \times \text{CPI}_i$$

根据不同停顿的 CPI 分解



根据不同停顿的 CPI 分解

【例1】假设某处理器的基本CPI (Base CPI) 是0.5，即理想情况下所有访问都在一级缓存命中时的CPI 为0.5，且该处理器频率为2.5 GHz。每次内存访问时间为100 纳秒，且包含所有未命中处理时间。如果某程序在该处理器执行时，一级缓存上的每条指令未命中率（Misses Per Instruction, MPI）达到了0.01，则该程序的总CPI 为多少?

根据不同停顿的 CPI 分解

$$
\begin{aligned}
\text{Total CPI}_1 &= \text{Base CPI} + \text{Memory stall CPI} \\
&= \text{Base CPI} + \frac{\text{Memory accesses}}{\text{Instruction}} \times \frac{\text{Memory stall cycles}}{\text{Memory access}} \\
&= \text{Base CPI} + \text{L1 cache MPI} \times \text{L1 cache miss penalty}
\end{aligned}
$$

$$
\text{Total CPI}_1 = 0.5 + 0.01 \times 250 = 3
$$

根据不同停顿的 CPI 分解

【例2】在例1 的基础上，假设在处理器中再加个二级缓存，其每次访问时间为5 纳秒，并且，与例1 相同的程序在二级缓存上的MPI 为0.005，则此时该程序的总CPI 为多少？处理器性能提升了多少?

根据不同停顿的 CPI 分解

$$\begin{aligned}\text{Total CPI}_2 &= \text{Base CPI} + \text{L2 stall CPI} + \text{Memory stall CPI} \\ &= \text{Base CPI} + \text{L1 cache MPI} \times \text{L1 cache miss penalty} \\ &\quad + \text{L2 cache MPI} \times \text{L2 cache miss penalty}\end{aligned}$$

$$\text{Total CPI}_2 = 0.5 + 0.01 \times 12.5 + 0.005 \times 250 = 0.5 + 0.125 + 1.25 = 1.875$$

$$\frac{\text{Total CPI}_1}{\text{Total CPI}_2} = \frac{3}{1.875} = 1.6$$

加入二级缓存后的处理器性能
是加入前的 1.6 倍!

根据不同停顿的 CPI 分解

$$\begin{aligned}\text{Total CPI} &= \text{Base CPI} + \text{2nd-level stall CPI} + \text{3rd-level stall CPI} + \cdots \\ &= \text{Base CPI} + \text{1st-level MPI} \times \text{1st-level miss penalty} \\ &\quad + \text{2nd-level MPI} \times \text{2nd-level miss penalty} + \cdots\end{aligned}$$

根据不同停顿的 CPI 分解

- 基于各层停顿可加性，定位处理器的性能瓶颈
- 在例2中，主要的性能瓶颈是什么？如何优化?

$$\text{Total CPI}_2 = 0.5 + 0.01 \times 12.5 + 0.005 \times 250 = 0.5 + 0.125 + 1.25 = 1.875$$

Top-down Microarchitecture Analysis Method (TMAM)

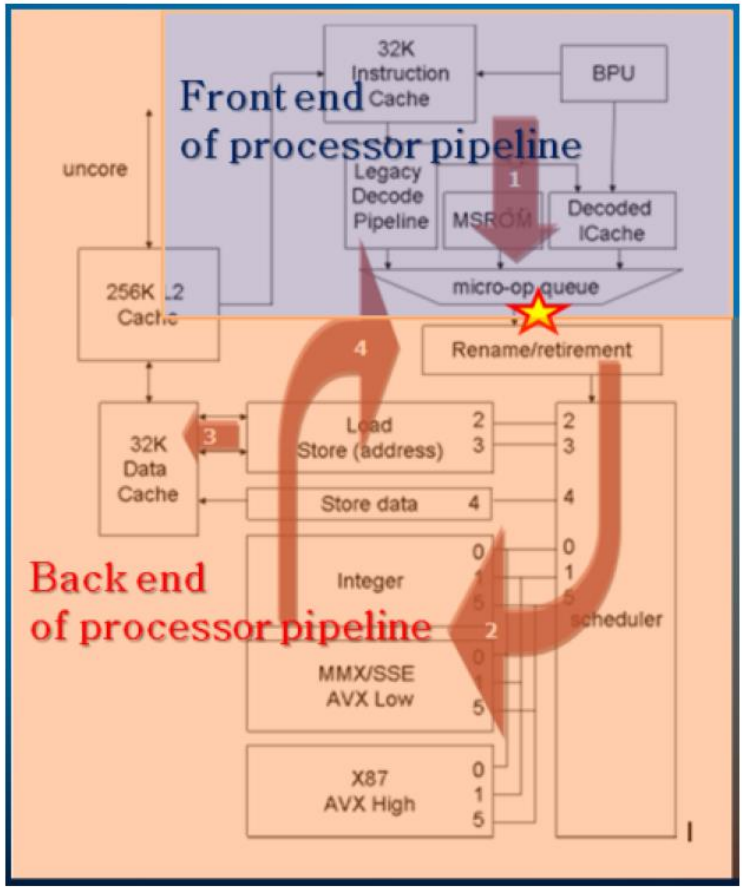


Figure 1: Out-of-order CPU block diagram - Intel Core™

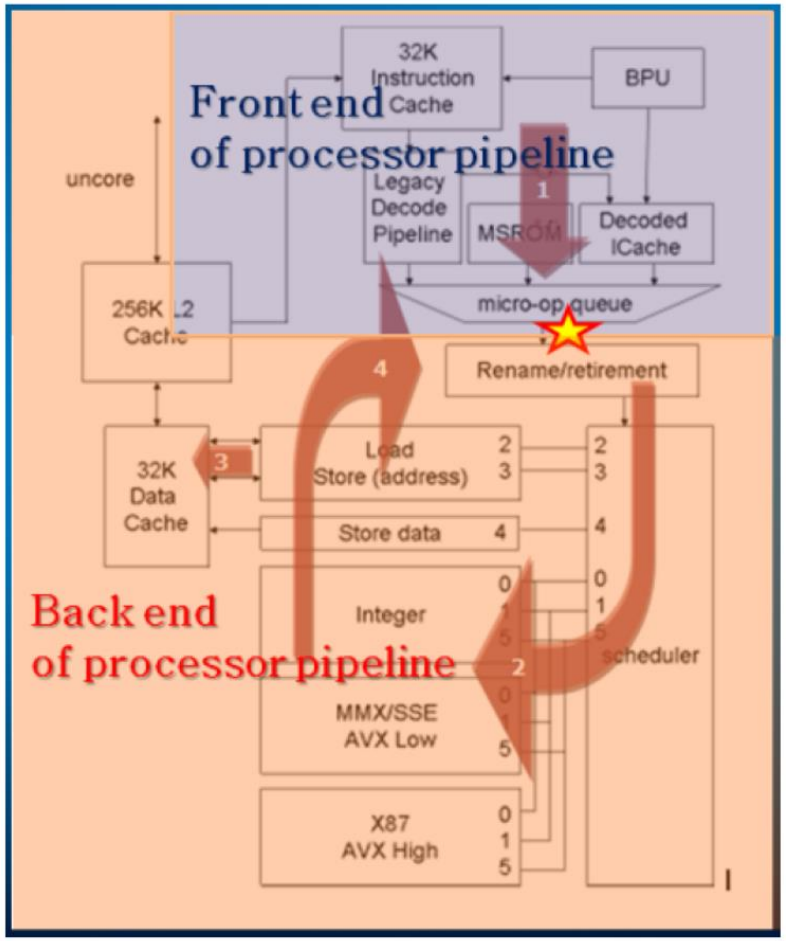
[Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. ISPASS 2014]

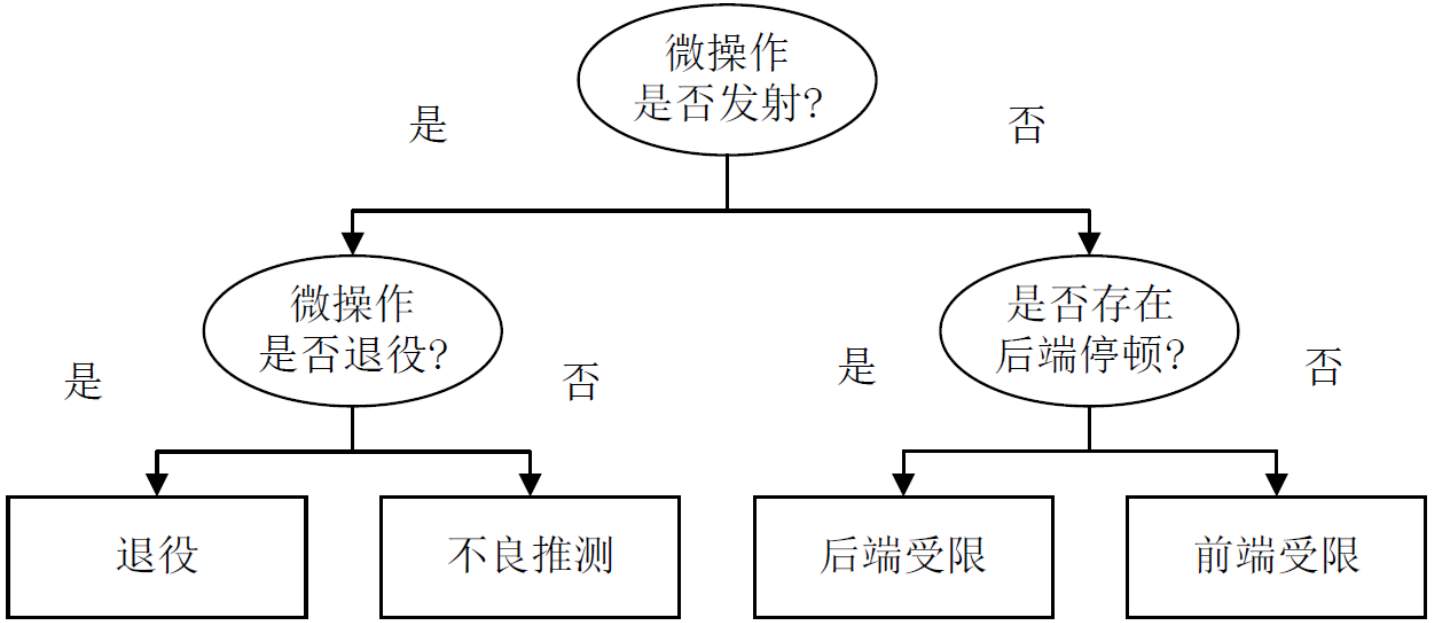
Traditional methods [4][5] do simple estimations of stalls. E.g. the numbers of misses of some cache are multiplied by a pre-defined latency:

$$Stall\_Cycles = \Sigma\ Penalty_i * MissEvent_i$$

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.

自顶向下的微体系结构分析方法 TMAM





自顶向下的微体系结构分析方法 TMAM

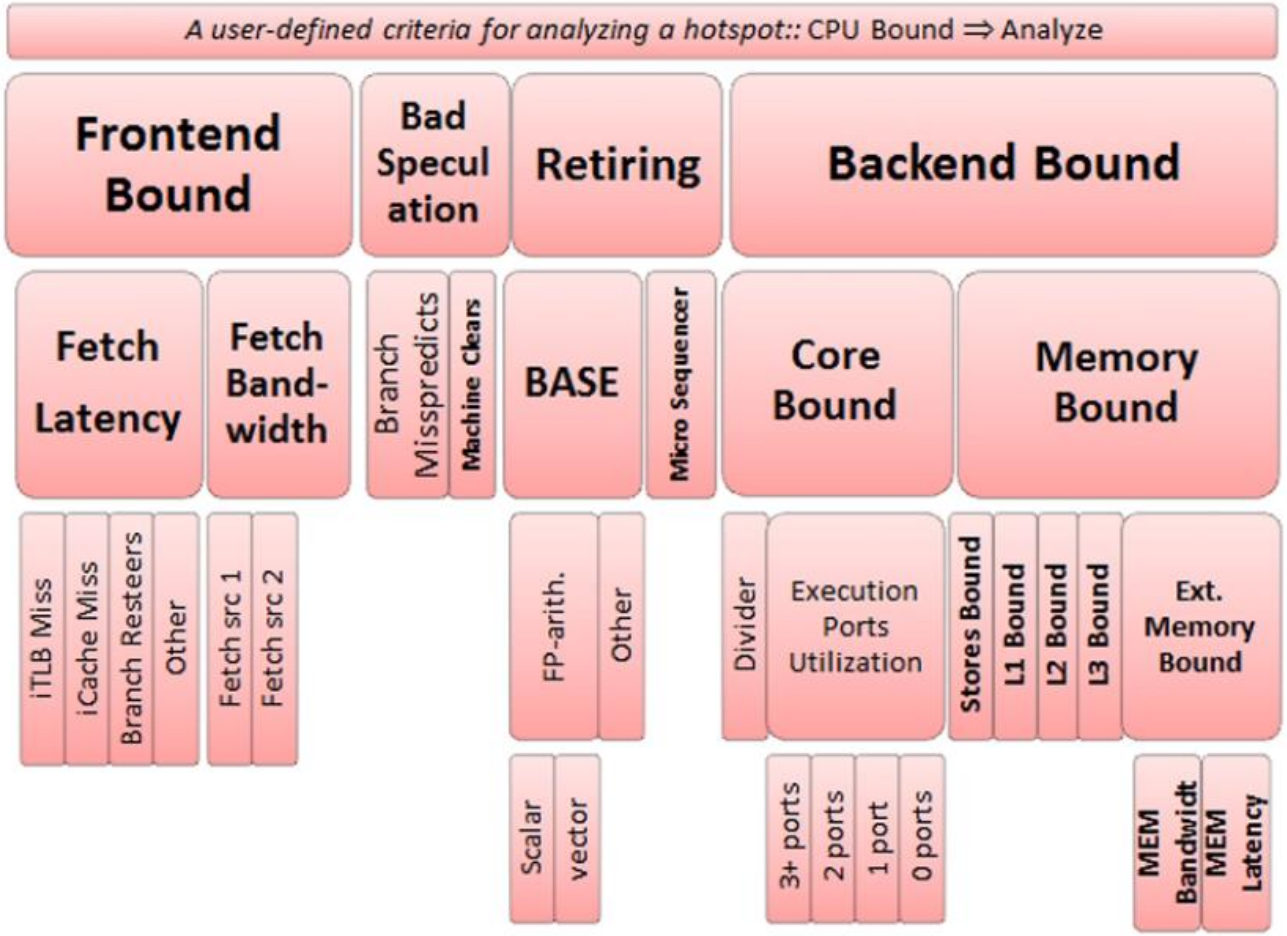


Table 7: Intel's implementation of Top-Down Metrics

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

一个矩阵乘法的例子

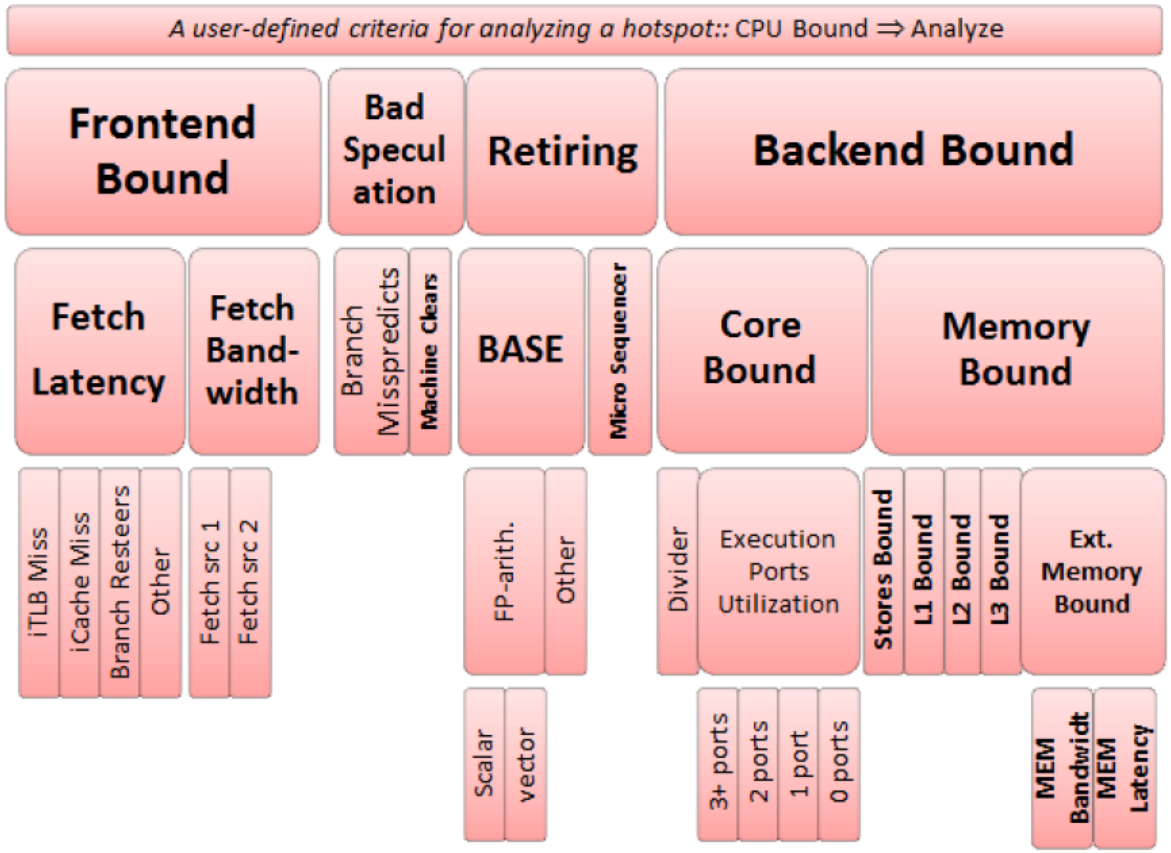


Figure 2: The Top-Down Analysis Hierarchy

[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.

采用 TMAM 的相关工具

- 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
- Arm Topdown Tool https://gitlab.arm.com/telemetry-solution/telemetry-solution/-/tree/main/tools/topdown_tool

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

小结

- 性能分析的主要目的是定位性能瓶颈
- 程序性能优化的一个重要原则是应该考虑程序执行时间的整体优化，而不仅仅是影响性能的任何单一部分的优化
- 唯一一致且可靠的性能测量指标是真实程序的执行时间
- 利用 CPI 分解方法及其累加性，可以帮助开发人员判断造成CPI偏高的主要原因，进而定位处理器的性能瓶颈，为处理器的优化提供建议
- 现代处理器各种复杂设计给微体系结构的准确分析带来了很多挑战，系统化的性能分析方法仍需不断探索和完善