==========

**Disclaimer**

The following document represents our best effort, as of February 2026, to summarize and interpret the Top-Down memory-related metrics based on the information and understanding available at this time. While we have aimed for technical accuracy and clarity, interpretations may evolve as documentation, microarchitectural disclosures, or tooling support change.

We do not claim that the interpretations presented here are definitive or guaranteed to be fully accurate. They reflect our current best technical understanding of the metrics and their behavior given the publicly available information at the time of writing.

This work has been prepared by Victor Xirau (victor.xirau@bsc.es) and Guillem Guerrero (gguerrer@bsc.es). For technical questions or clarifications, please contact either author.

==========

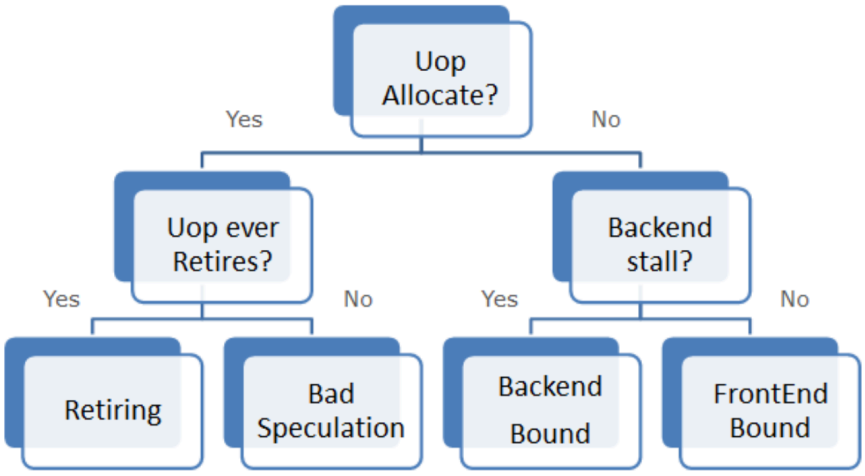
# Introduction

The goal of this document is to explain the Top-Down method in a detailed yet understandable way. This document is based in the original method proposed by Eyerman et al. (https://ieeexplore.ieee.org/document/4205127), and also Intel's specific approach done by Ahmad Yasin (https://rcs.uwaterloo.ca/~ali/cs854-f23/papers/topdown.pdf).

## The Top-Down Method

The Top-Down method aims to give useful hardware-related performance information to the software developer. To do this, it decomposes processor performance into individual cycles or slots that are then categorized depending on their resource utilization characteristics, building a CPI Stack.

Intel's particular approach, Top-Down Microarchitecture Analysis or TMA, starts with four categories: **Retiring**, **Front-End Bound**, **Back-End Bound**, or **Bad Speculation**. Every pipeline slot, which can be considered as a potentially dispatched micro-operation (uOp) in a cycle, is assigned to strictly **one** of these four categories. To decide which category is the most adequate, the following diagram is used:



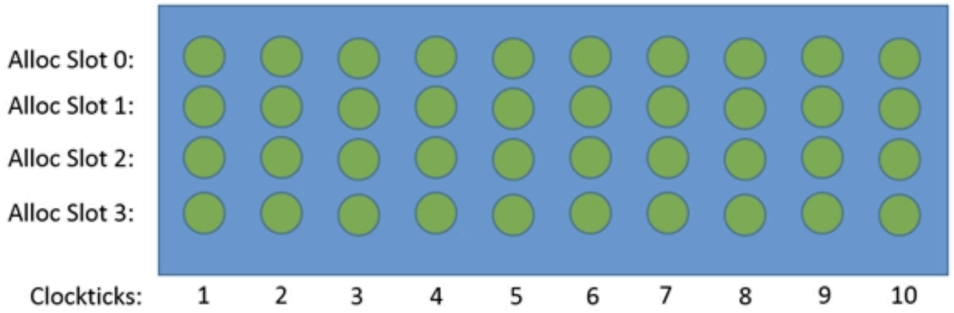

This first level intends to be a broad classification, that can be achieved by answering just two binary questions. At level 1, the goal is to see which broad category represents the biggest bottleneck for performance, so that deeper research can be properly guided.

There are a total of 4 levels of metrics. This document contains a complete list of the metrics and their categories in Appendix Metrics. Since this document focuses on memory related metrics, level 3 and level 4 will only cover those metrics that belong to this area.

## Pipeline Slots & Clockticks

Depending on the level of the metric, this can be measured using pipeline slots or clock ticks (also called core clock cycles). This is not minor, since it does not just affect how the metric is measured, but also how the result should be interpreted.

The Top-Down Characterization assumes that for each CPU core, on each clock cycle, there are four pipeline **slots** available which have to be classified in one of Top-Down's categories.




CPU representation retiring 40 execution slots. Source: https://portal.nacad.ufrj.br/online/intel/vtune2017/help/GUID-B7751BB0-7E69-11E6-AAEF-28D24465DA3C.html

In the image, a total of 40 slots can be seen. This number results from the 4 core slots multiplied by the number of clock cycles. If we consider that green slots are retiring uOps, in this case all 40 slots would fall under the Retiring Level 1 category.

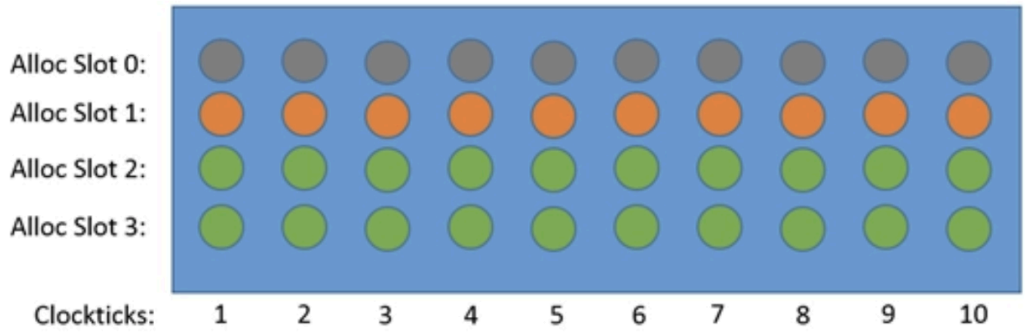
However, if we now consider the same set of slots, but now containing some stalls (greyed-out circles), the result changes:

CPU representation retiring 20 execution slots plus 20 stalled slots. Source: https://portal.nacad.ufrj.br/online/intel/vtune2017/help/GUID-B7751BB0-7E69-11E6-AAEF-28D24465DA3C.html

Now, 50% of the slots would still fall under the Retiring category, but the other 50% would be distributed between Bad Speculation, Backend Bound, and Frontend bound.

This slot-based approach is used for Level 1 and Level 2. However, Level 3 and Level 4 use Clock Ticks, which are measured following a different logic.

**Clock Ticks** are equivalent to core clock cycles. This means that any metric that is calculated using them, will consider all slots for a same cycle as the same clock tick. Continuing with the same diagram as a base:




CPU representation with 100% retiring clocktics, and 100% for the other two categories. Source: https://portal.nacad.ufrj.br/online/intel/vtune2017/help/GUID-FE3360D3-42E7-4955-BCB9-E00FA1839FBC.html

Considering slots, we still have 50% of them retiring (green) and 50% falling in other categories (orange and grey). However, from the Clock Tick perspective, since we have **at least** one stall on every cycle, 100% of the Clock Ticks would be stalled. In this case, if we imagine that green circles mean Retiring, orange mean L1 Bound, and grey mean L2 Bound, we could say that 50% of slots are retiring, 100% of clock ticks are L1 Bound, and 100% of clock ticks are L2 Bound.

For Level 1 and Level 2 metrics such as Retiring or Bad Speculation, Top-Down uses slots. However, metrics are based on clock ticks for Level 3 and Level 4.

## General Considerations

- This document does not cover all the metrics. It covers just a set of them; mainly memory-related metrics. Check the appendix to see the complete tree of metrics.
- Cmask can be added to a counter, to set a threshold. For example, if we do `EXE_ACTIVITY.BOUND_ON_LOADS:c5`, it means that This means that we will add 1 “count” to the counter only if in that cycle, there are 5 or more occurrences of the event.The syntax can be: `EXE_ACTIVITY.BOUND_ON_LOADS:c5` `EXE_ACTIVITY.BOUND_ON_LOADS:c=5` `EXE_ACTIVITY.BOUND_ON_LOADS.c=5` or using the raw event code like `cpu/event=0xa6,umask=0x21,cmask=5/`. Intel’s Top-Down paper notes that such thresholds are *implementation-specific* – chosen based on the core’s issue width and buffering – and are meant to represent when a workload is genuinely memory-bound rather than just experiencing occasional latency that doesn’t bottleneck execution. Some of the masks are there “by default” in the definition of the event. For example, the event `EXE_ACTIVITY.BOUND_ON_LOADS` is defined with cmask 5, there is no defined event with the same event code and umask but with no cmask.

- Since the same clock tick can belong to different Level 3 and Level 4 metrics, one should never understand these metrics as mutually exclusive. Results where Level 3 or Level 4 metrics don't add up to 100% are common and are not necessarily wrong.
- Level 3 and Level 4 categories just add more insight, but they are not breaking down the slots like Level 2 did for Level 1.
- Metrics have been explained using Intel's latest TMA counters which enabled `PERF_METRICS` on those that had it available. It would also be worth exploring alternative formulations, starting with older Intel machines that already could compute topdown without `PERF_METRICS`.

# Top-Down Levels Explained

The main memory-related Top-Down Microarchitecture Analysis metrics are explained here. All Level 1 and Level 2 metrics are explained, but for Level 3 and Level 4 only those related to memory are discussed.

For each metric, the Intel VTune explanation, formula, used counters, and details will be provided.

**!!!!** All the first level metrics, as well as those in L2, are finally divided by PERF_METRICS_SUM. This metric is basically computed by adding up all the L1 metrics together, and is used to normalize the percentages to make sure they all add up to 100%.

$$\text{PERF\_METRICS\_SUM} = \frac{\begin{array}{l}\text{PERF\_METRICS.FRONTEND\_BOUND} + \text{PERF\_METRICS.BAD\_SPECULATION} \\ + \text{PERF\_METRICS.RETIRING} + \text{PERF\_METRICS.BACKEND\_BOUND}\end{array}}{\text{TOPDOWN.SLOTS}}$$

## Level 1

Level 1 metrics represent the 4 basic categories for the Top-Down Microarchitecture Analysis. These are: Retiring, Bad Speculation, Frontend Bound, and Backend Bound. All these metrics are slot-based.

### Retiring

- VTune Explanation: Retiring metric represents a Pipeline Slots fraction utilized by useful work, meaning the issued uOps that eventually get retired. Ideally, all Pipeline Slots would be attributed to the Retiring category. Retiring of 100% would indicate the maximum possible number of uOps retired per cycle has been achieved. Maximizing Retiring typically increases the Instruction-Per-Cycle metric. Note that a high Retiring value does not necessary mean no more room for performance improvement. For

example, Microcode assists are categorized under Retiring. They hurt performance and can often be avoided.

- Formula:

$$Retiring = \frac{(PERF\_METRICS.RETIRING \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: This metric tells us how much of the CPU's issue bandwidth is doing productive work. A low Retiring percentage means many slots are going idle or getting wasted (due to frontend, backend, or speculation issues). Ideally, you want to maximize Retiring (to increase IPC). However, if Retiring is high but performance is still sub-par, you'd check the Level 2 breakdown (Heavy vs. Light Operations) to see if those retired uOps were inefficient (e.g. many slots used for one high-latency instruction).

**Bad Speculation**

- VTune Explanation: Bad Speculation represents a Pipeline Slots fraction wasted due to incorrect speculations. This includes slots used to issue uOps that do not eventually get retired and slots for which the issue-pipeline was blocked due to recovery from an earlier incorrect speculation. For example, wasted work due to mispredicted branches is categorized as a Bad Speculation category. Incorrect data speculation followed by Memory Ordering Nukes is another example.

- Formula:

$$BadSpeculation = \frac{(PERF\_METRICS.BAD\_SPEC \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: High Bad Speculation means you're losing performance because the processor's guesses (branch prediction or other speculative execution) are frequently wrong. For example, a branch misprediction causes the CPU to flush uOps and restart on the correct path, wasting slots.

**Frontend Bound**

- VTune Explanation: Front-End Bound metric represents a slots fraction where the processor's Front-End undersupplies its Back-End. Front-End denotes the first part of the processor core responsible for fetching operations that are executed later on by the Back-End part. Within the Front-End, a branch predictor predicts the next address to fetch, cache-lines are fetched from the memory subsystem, parsed into instructions, and lastly decoded into micro-ops (uOps). Front-End Bound metric denotes unutilized issue-slots when there is no Back-End stall (bubbles where Front-End delivered no uOps while Back-End could have accepted them). For example, stalls due to instruction-cache misses would be categorized as Front-End Bound.
- Formula:

$$FrontendBound = \frac{(PERF\_METRICS.FRONTEND\_BOUND \,/\, TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM} - \frac{INT\_MISC.UOP\_DROPPING}{TOPDOWN.SLOTS}$$

- Details: If Front-End Bound is high, it implies the program's performance is limited by instruction supply issues. Essentially, the core wasn't doing work because it had no uOps to execute, due to front-end inefficiencies.

**Backend Bound**

- VTune Explanation: Back-End Bound metric represents a Pipeline Slots fraction where no uOps are being delivered due to a lack of required resources for accepting new uOps in the Back-End. Back-End is the portion of the processor core where an out-of-order scheduler dispatches ready uOps into their respective execution units, and, once completed, these uOps get retired according to the program order. For example, stalls due to data-cache misses or stalls due to the divider unit being overloaded are both categorized as Back-End Bound. Back-End Bound is further divided into two main categories: Memory Bound and Core Bound.
- Formula:

$$BackendBound = \frac{(PERF\_METRICS.BACKEND\_BOUND \,/\, TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: High Backend Bound values mean that the pipeline often had instructions ready, but these instructions couldn't issue/complete due to some capacity constraint or latency in the backend (execution units or memory subsystem).

## Level 2

Level 2 metrics split each Level 1 category into two subcategories, providing more detail about the cause of stalls or inefficiencies. The formulas given use the Level 1 metrics (or underlying counters) to compute these. They are structured so that each pair of sub-metrics sums (approximately) to the parent metric. Level 1 metrics are split into Level 2 metrics in the following way:

- **Retiring** splits into **Heavy Operations** and **Light Operations**.
- **Bad Speculation** splits into **Branch Mispredict** and **Machine Clears**.
- **Frontend Bound** splits into **Front-End Latency** and **Front-End Bandwidth**.
- **Backend Bound** splits into **Memory Bound** and **Core Bound**.

**Heavy Operations**

- VTune Explanation: This metric represents the fraction of slots where the CPU retired heavy-weight operations (instructions that require 2+ uops).
- Formula:

$$HeavyOperations = \frac{(PERF\_METRICS.HEAVY\_OPERATIONS \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: A high Heavy Operations fraction means many pipeline slots are being “consumed” by relatively few instructions (each instruction taking multiple μops worth of pipeline resources). This can be a sign of suboptimal code because ideally, more instructions would be simple (1 μop each) to fully utilize decode/issue bandwidth.

**Light Operations**

- VTune Explanation: This metric represents a fraction of slots where the CPU retired light-weight operations - instructions that require no more than one uop (micro-operation). This correlates with total number of instructions used by the program. A uops-per-instruction ratio of 1 should be expected. While this is the most desirable metric, high values can also provide opportunities for performance optimizations.
- Formula:

$$LightOperations = max(0,\ Retiring - HeavyOperations)$$

- Counter Information: This metric is calculated based on previously calculated metrics.
- Details: Using the max() just guards against any rounding issue causing a tiny negative. Logically, HeavyOperations is a subset of Retiring so this difference should be greater or equal to 0. Higher LightOperations fraction implies the program is executing mostly

simple instructions (1 uOp each), which often correlates with efficient software (doing more with fewer uOps).

**Branch Mispredict**

- VTune Explanation: When a branch mispredicts, some instructions from the mispredicted path still move through the pipeline. All work performed on these instructions is wasted since they would not have been executed had the branch been correctly predicted. This metric represents slots fraction the CPU has wasted due to Branch Misprediction. These slots are either wasted by uOps fetched from an incorrectly speculated program path, or stalls when the out-of-order part of the machine needs to recover its state from a speculative path.
- Formula:

$$BranchMispredict = \frac{(PERF\_METRICS.BRANCH\_MISPREDICT \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: High Branch Mispredict percentage means a lot of work was wasted on wrong path execution or flushing. The structure directly reflects that branch mispredicts consume a share of the pipeline bandwidth that produces no retirement. This metric is important because branch mispredictions force the CPU to discard work and refetch instructions, incurring in a performance penalty roughly equal to the pipeline depth. The formula captures both aspects of that penalty: (1) slots used by incorrectly speculated µops, and (2) slots where the pipeline was idle waiting to fetch the correct path after the mispredict.

**Machine Clears**

- VTune Explanation: Certain events require the entire pipeline to be cleared and restarted from just after the last retired instruction. This metric measures three such events: memory ordering violations, self-modifying code, and certain loads to illegal address ranges. Machine Clears metric represents slots fraction the CPU has wasted due to Machine Clears. These slots are either wasted by uOps fetched prior to the clear, or stalls the out-of-order portion of the machine needs to recover its state after the clear.
- Formula:

$$MachineClears = max(0,\ BadSpeculation - BranchMispredict)$$

- Counter Information: This metric is calculated based on previously calculated metrics.

- Details: Machine clears are pipeline flushes caused by events other than branch mispredicts. During a machine clear, the pipeline is flushed just like a mispredict, so those cycles have no retiring uOps and are counted in Bad Speculation. Since branch-related ones are separately accounted, the remainder is attributed to Machine Clears.

**Frontend Latency**

- VTune Explanation: This metric represents a fraction of slots during which CPU was stalled due to front-end latency issues, such as instruction-cache misses, ITLB misses or fetch stalls after a branch misprediction. In such cases, the front-end delivers no uOps.
- Formula:

$$FrontendLatency = \frac{(PERF\_METRICS.FETCH\_LATENCY \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM} - \frac{INT\_MISC.UOP\_DROPPING}{TOPDOWN.SLOTS}$$

- Details: Essentially, this metric counts cases where the Instruction Fetch/Decode pipeline had bubbles because it was waiting on something (for example, a long-latency event like an i-cache miss or ITLB miss). nIn these metric, `INT_MISC.UOP_DROPPING` is also subtracted to exclude bubbles not caused by true frontend latency.

**Frontend Bandwidth**

- VTune Explanation: This metric represents a fraction of slots during which CPU was stalled due to front-end bandwidth issues, such as inefficiencies in the instruction decoders or code restrictions for caching in the DSB (decoded uOps cache). In such cases, the front-end typically delivers a non-optimal amount of uOps to the back-end.
- Formula:

$$FrontendBandwidth = max(0,\ FrontendBound - FrontendLatency)$$

- Counter Information: This metric is calculated based on previously calculated metrics.
- Details: In essence, Frontend Bandwidth issues occur when the frontend is active but not keeping up with the backend's demand. Causes for this include the decoder not being able to sustain the needed uOp rate, or limits in the throughput of the DSB (Decoded Stream Buffer) versus what the backend could accept.

**Memory Bound**

- VTune Explanation: This metric shows how memory subsystem issues affect the performance. Memory Bound measures a fraction of slots where pipeline could be stalled due to demand load or store instructions. This accounts mainly for incomplete

in-flight memory demand loads that coincide with execution starvation in addition to less common cases where stores could imply back-pressure on the pipeline.
- Formula:

$$MemoryBound = \frac{(PERF\_METRICS.MEMORY\_BOUND \ / \ TOPDOWN.SLOTS)}{PERF\_METRICS\_SUM}$$

- Details: This metric measures how often memory delays are keeping the execution units idle. Two main contributors are (1) pending load misses causing no ready uOps to execute, and (2) cases where many stores are buffered (store queue full) which can also stall the pipeline.

**Core Bound**

- VTune Explanation: This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware compute resources, or dependencies software's instructions are both categorized under Core Bound. Hence it may indicate the machine ran out of an OOO resources, certain execution units are overloaded or dependencies in program's data- or instruction- flow are limiting the performance (e.g. FP-chained long-latency arithmetic operations).
- Formula:

$$CoreBound = max(0,\ BackendBound - MemoryBound)$$

- Counter Information: This metric is calculated based on previously calculated metrics.
- Details: This metric shows how often the CPU's execution resources were the limiting factor as opposed to waiting for memory. If Core Bound is high, the program is execution-bound (not memory-bound) — meaning performance is limited by factors like: limited instruction-level parallelism (only one or two execution ports active out of several), contention on particular execution units (e.g., an overloaded divider or vector unit), or general ALU throughput issues. For example, if code has a dependency chain that forces mostly one port to be used (1 port bound) or a lot of divisions causing the divider to be a bottleneck, those slots would count as Core Bound.

## Level 3

This section will only focus on Memory Bound metrics. All other categories also have their respective Level 3 metrics, but the focus of this document is memory related metrics.

Level 3 metrics provide an even more detailed breakdown, especially splitting Memory Bound stalls into specific cache levels (L1, L2, L3, DRAM) and identifying Store-related stalls.

**These metrics are typically expressed as fraction of pipeline cycles (not just slots**) because they deal with specific stall cycles.

Before diving deep into individual metrics, it's worth mentioning all these metrics use ****`CPU_CLK_UNHALTED.THREAD` ****as the denominator for their formula. This counter measures "unhalted" CPU clock ticks for a given hardware thread.

The counter increments by 1 every cycle where the CPU is actively running instructions (or stalled on work) rather than halted/idle.

It does not require any special configuration to use in most profiling tools – **it's often the default event for counting "cycles."**

**Many performance analysis tools** (Linux perf, VTune, PAPI, etc.) **map their generic "cycles" or "total cycles" event to** `CPU_CLK_UNHALTED.THREAD`**.**

**L1 Bound**

- VTune Explanation: This metric shows how often machine was stalled without missing the L1 data cache. The L1 cache typically has the shortest latency. However, in certain cases like loads blocked on older stores, a load might suffer a high latency even though it is being satisfied by the L1.
- Formula:

$$L1Bound = \frac{(EXE\_ACTIVITY.BOUND\_ON\_LOADS : c5 \; - \; MEMORY\_ACTIVITY.STALLS\_L1D\_MISS : c3)}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: This metric subtracts the L1-miss stall cycles from the total load stall cycles, and then divides by total cycles. This gives the fraction of cycles where the CPU was stalled on a load without an L1 miss being the cause.

  In other words, it measures memory-related stalls that are not due to missing in L2/L3, meaning the data was actually found in L1. This means that even though the needed data was in the L1 cache (no miss), the CPU was still waiting on it. This can happen if there are dependency or concurrency issues: e.g., a load may hit L1 but if it's part of a dependent chain of loads, the CPU might still stall waiting for it to complete (L1 latency is short, ~4 cycles, but in a tight dependency chain that's enough to stall).

  This metric can also get increased if there are loads blocked by earlier operations like an unresolved store-forwarding or a full line buffer for a locked operation.

  **It represents the CPU being bottlenecked by L1 latency** or related short-latency memory issues, rather than longer cache misses. The formula structure ensures that any stall cycle that involved an L1 miss is excluded, leaving only stalls with L1 hits (or non-cache-related memory stalls like store-forwarding or lock acquisition which are

treated as L1-level issues).

**L2 Bound**

- VTune Explanation: This metric shows how often machine was stalled on L2 cache. Avoiding cache misses (L1 misses/L2 hits) will improve the latency and increase performance.
- Formula:

$$L2Bound = \frac{(MEMORY\_ACTIVITY.STALLS\_L1D\_MISS:c3 \ - \ MEMORY\_ACTIVITY.STALLS\_L2\_MISS:c5)}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: This metric represents the fraction of cycles stalled due to L1 misses that were serviced by L2 (no further miss). If L2 Bound is significant, it means the L1 cache wasn't large or effective enough (frequent L1 misses), but the L2 was able to supply those. Reducing L1 misses (through better code locality or data structure alignment) could improve performance.

**L3 Bound**

- VTune Explanation: This metric shows how often CPU was stalled on L3 cache, or contended with a sibling Core. Avoiding cache misses (L2 misses/L3 hits) improves the latency and increases performance.
- Formula:

$$L3Bound = \frac{(MEMORY\_ACTIVITY.STALLS\_L2\_MISS:c5 \ - \ MEMORY\_ACTIVITY.STALLS\_L3\_MISS:c9)}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: A high L3 Bound means there are many L2 misses (so either the working set exceeds L2, or L2 is being bypassed due to streaming patterns) while L3 is servicing them. It indicates the L3 cache latency is a significant part of execution time.

  Additionally, on multi-core systems, if the L3 is shared, contention or accessing data from another core's slice can also show up as L3 Bound stalls.

  Optimizing for better L2 usage or improving data locality might help in reducing L3 Bound. If L3 Bound plus DRAM Bound are both high, then the memory as a whole can be considered as a major bottleneck.

**DRAM Bound**

- VTune Explanation: This metric shows how often CPU was stalled on the main memory (DRAM). Caching typically improves the latency and increases performance.
- Formula:

$$DRAMBound = \frac{MEMORY\_ACTIVITY.STALLS\_L3\_MISS : c9}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: DRAM Bound indicates how often the processor was stalled on memory accesses that went all the way to external memory. A high DRAM Bound means the program is suffering many last-level cache misses, causing frequent slow DRAM fetches that stall the pipeline. This is often the most severe form of memory bottleneck since DRAM latency is much larger than cache latency.

  If DRAM Bound is high, it suggests that caching is not capturing the working set effectively or memory accesses are very large/streaming. Typical advice would be to improve data locality, use better algorithms, or possibly hardware solutions like prefetching. If the CPU has hardware prefetchers or memory-level parallelism, not every DRAM access will stall the core fully (some might be overlapped). These events usually measure stall cycles, which inherently account for overlap, meaning that DRAM Bound already captures the net stall fraction due to memory misses.

**Store Bound**

- VTune Explanation: This metric shows how often CPU was stalled on store operations. Even though memory store accesses do not typically stall out-of-order CPUs; there are few cases where stores can lead to actual stalls.
- Formula:

$$StoreBound = \frac{EXE\_ACTIVITY.BOUND\_ON\_STORES}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: This metric indicates the fraction of cycles where stores were the primary limiter of execution. A low Store Bound is common, but if it's non-zero or high that points to a store throughput problem (e.g., streaming stores, unaligned or split stores, heavy write traffic that exceeds what the core's buffering can hide).

## Level 4

Level 4 metrics are closely related. It is recommended to read them both so that the meaning can be better understood.

In fact, as it can be seen in the Counter Information section, although having different names, both metrics use the same event but with a different Cmask.

**Memory Bandwidth**

- VTune Explanation: This metric represents a fraction of cycles during which an application could be stalled due to approaching bandwidth limits of the main memory. This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider improving data locality in NUMA multi-socket systems.
- Formula:

$$MemoryBW = \frac{min(CPU\_CLK\_UNHALTED.THREAD,\ OFFCORE\_REQUESTS\_OUTSTANDING.ALL\_DATA\_RD : c4)}{CPU\_CLK\_UNHALTED.THREAD}$$

- Details: If there are a lot of cycles with 4 or more outstanding requests, it means a lot of memory hierarchy petitions are being performed. If many cycles are in this situation, it can be because one of two reasons:
    - Because **a lot** of memory petitions are being performed, even if they are solved fast. This is considered bandwidth bound because not all of them could fit in caches, and they are bound to be solved by main memory at some point.
    - There might be a little less petitions than in the previous situation, but they are **solved by DRAM** and that takes more time than resolving with caches. For this reason, there are 4 or more outstanding requests across multiple cycles.
- One could think that since the “problem” is that requests are taking time to respond, then it means it’s latency bound. Although this is an understandable reasoning, it’s not the case. The metric is not called “Bandwidth Bound” but “Memory Bandwidth”, and is under “DRAM Bound”. The metric indicates a percent of bandwidth used; if a lot of cycles present 4 or more outstanding requests in a given cycle during many cycles, then bandwidth will be high. However, **this metric must be understood together with Latency** to have a better picture of the analysis.

**Memory Latency**

- VTune Explanation: This metric represents a fraction of cycles during which an application could be stalled due to the latency of the main memory. This metric does not aggregate requests from other threads/cores/sockets (see Uncore counters for that). Consider optimizing data layout or using Software Prefetches (through the compiler).
- Formula:

$$MemoryLatency = \frac{\min\left(CPU\_CLK\_UNHALTED.THREAD,\ OFFCORE\_REQUESTS\_OUTSTANDING.CYCLES\_WITH\_DATA\right)}{CPU\_CLK\_UNHALTED.THREAD} - MemoryBW$$

- Details: Memory Latency measures situations in which there are many cycles with some outstanding requests, but the number of requests per cycle is not that big (less than 4). This can happen in situations where the memory system is not being used that much. For a given time, **there are not so many memory requests**, meaning that there is no situation in which there are 4 or more in the queue.

  If this number is high but bandwidth isn't, it means that there is always or almost always an outstanding request waiting, which means that they take a long time to resolve. Thus meaning that the analyzed application is latency intensive.

  If the application is not latency intensive and there are not many memory requests, the system should be "solving them fast", probably thanks to caches. This would result in low amount of cycles where the system is still "waiting for data". If wait times are high, but Memory Latency metric is low, it means it's little petitions that are waiting for a long time.

# Conclusions & Remarks

Although the TopDown Method is widely used and helpful indeed, it's also true that it has some caveats.

One of the main weaknesses is its inconsistency in how metrics are calculated in Level 1 and Level 2, compared to how it's done in Level 3 and Level 4. It might seem minor, and one can understand the reasoning behind this. However, in reality changing from calculating the metrics in percentage of slots to percentage of clocktics makes it really complicated to actually understand it. It's strange that in a tree of metrics where until that point they have been mutually exclusive, at a given depth they start to overlap.

It is also true, that sometimes some metrics can be misleading if the formulas are not checked. For example, L1 bound may seem like it would represent some ratio where the L1 misses are factored in, while in fact it excludes L1 misses because they imply L2 accesses.

Finally, while they say that the goal of this method is to provide the software developer with useful hardware-related metrics, it's also worth noting that TopDown summary (particularly when used with VTune) has some room for improvement on how the analyzed code and the results are related. If the software developer can't have precise information about specific areas where the code can be enhanced, part of the method value is lost in the process. It is also true, that the output information is not that close to the language commonly used by software developers, rather closer to that of computer architects. This generates a barrier that can detriment the ability to extract conclusions.

In the end, although this method has been designed trying to help software developers find areas of potential improvement in their code, it seems like it still has some room for improvement.

# References


**1. ROGERS, Ian; WANG, Weilin.** *Linux Perf Metrics Talk Slides*. In: *Linux Plumbers Conference 2023* [online]. [Accessed: 1 April 2025]. Available at: https://lpc.events/event/17/contributions/1514/attachments/1335/2675/LPC_2023_Linux_Perf_Metrics_Talk_Slides.pdf

**2. Intel Corporation.** *Sapphire Rapids Server – Hardware Performance Monitoring Events Reference* [online]. [Accessed: 1 April 2025]. Available at: https://perfmon-events.intel.com/spxeon.html

**3. rabexc.org.** *Sapphire Rapids metrics JSON – Linux kernel sources* [online]. [Accessed: 1 April 2025]. Available at: https://sbexr.rabexc.org/latest/sources/d3/9e42190115fca8.html

**4. KLEEN, Andi.** *toplev manual* [online]. GitHub Wiki. Last updated: 9 November 2021 [Accessed: 1 April 2025]. Available at: https://github.com/andikleen/pmu-tools/wiki/toplev-manual

**5. Intel Corporation.** *TMA Metrics Full CSV* [online]. GitHub repository. [Accessed: 3 April 2025]. Available at: https://github.com/intel/perfmon/blob/main/TMA_Metrics-full.csv

**6. KLEEN, Andi.** *spr_server_ratios.py* [online]. GitHub repository. [Accessed: 3 April 2025]. Available at: https://github.com/andikleen/pmu-tools/blob/master/spr_server_ratios.py

**7. Intel Corporation.** *Top-Down Microarchitecture Analysis Method* [online]. Intel VTune Profiler Cookbook, 2023. [Accessed: 3 April 2025]. Available at: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-0/top-down-microarchitecture-analysis-method.html

**8. ROGERS, Ian.** *[PATCH v2 13/13] perf metrics: Add metric group 'Backend Bound'* [online]. Patchew.org, 4 October 2022. [Accessed: 3 April 2025]. Available at:

https://patchew.org/linux/20221004021612.325521-1-irogers@google.com/20221004021612.325521-14-irogers@google.com/

**9. Intel Corporation.** *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2* [online]. December 2023. [Accessed: 3 April 2025]. Available at: https://cdrdv2-public.intel.com/835755/253669-sdm-vol-3b.pdf

**10. Stack Overflow.** *Why does perf stat not count cycles_u on Broadwell CPU with hyperthreading disabled?* [online]. Discussion thread, 2023. [Accessed: 3 April 2025]. Available at: https://stackoverflow.com/questions/75601428/why-does-perf-stat-not-count-cyclesu-on-broadwell-cpu-with-hyperthreading-disab

**11. Oracle Corporation.** *Generic Events* [online]. Oracle Solaris Studio 12.3 Documentation. [Accessed: 3 April 2025]. Available at: https://docs.oracle.com/cd/E26502_01/html/E29036/generic-events-3cpc.html

**12. Stack Overflow.** *What are stalled-cycles-frontend and stalled-cycles-backend in perf stat results?* [online]. Discussion thread, 2014. [Accessed: 3 April 2025]. Available at: https://stackoverflow.com/questions/22165299/what-are-stalled-cycles-frontend-and-stalled-cycles-backend-in-perf-stat-resul

**13. YASIN, Ahmad.** *A Top-Down Method for Performance Analysis and Counters Architecture* [online]. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014. [Accessed: 3 April 2025]. Available at: https://rcs.uwaterloo.ca/~ali/cs854-f23/papers/topdown.pdf

**14. Intel Corporation.** *Intel® 64 and IA-32 Architectures Optimization Reference Manual* [online]. Document 248966-049US, 2023. [Accessed: 3 April 2025]. Available at: https://cdrdv2-public.intel.com/814198/248966-Optimization-Reference-Manual-V1-049.pdf

**15. Intel Corporation.** *Top-Down Microarchitecture Analysis Method Cookbook (2023.1)* [online]. Intel VTune Profiler Documentation. [Accessed: 3 April 2025]. Available at: https://www.intel.com/content/www/us/en/docs/vtune-profiler/cookbook/2023-1/top-down-microarchitecture-analysis-method.html

**16. Intel Corporation.** *intel/perfmon Repository (TMA_Metrics.xlsx and related files)* [online]. GitHub repository. [Accessed: 3 April 2025]. Available at: https://github.com/intel/perfmon

**17. KLEEN, Andi.** *pmu-tools (toplev)* [online]. GitHub repository. [Accessed: 3 April 2025]. Available at: https://github.com/andikleen/pmu-tools

**18. perfwiki.github.io.** *perf Top-Down Analysis Wiki* [online]. [Accessed: 3 April 2025]. Available at: https://perfwiki.github.io/main/top-down-analysis/

**19. BAKHVALOV, Denis.** *Top-Down Performance Analysis Methodology* [online]. Easyperf Blog, 9 February 2019. [Accessed: 3 April 2025]. Available at: https://easyperf.net/blog/2019/02/09/Top-Down-performance-analysis-methodology

**20. BAKHVALOV, Denis.** *Performance Analysis and Tuning on Modern CPUs* [online]. Book and supplementary materials. [Accessed: 3 April 2025]. Available at: https://easyperf.net/perf-book

**21. YASIN, Ahmad.** *Software Optimizations Become Simple with Top-Down Analysis* [online video]. Intel Developer Forum (IDF), 2015. [Accessed: 3 April 2025]. Available at: https://www.youtube.com/watch?v=kjufVhyuV_AV

**22. Intel Corporation.** *How Top-Down Microarchitecture Analysis Addresses Performance Analysis Challenges* [online video]. [Accessed: 3 April 2025]. Available at: https://www.youtube.com/watch?v=NabAgEgopLs

**23. Arm Ltd.** *Arm CPU Telemetry Solution: Topdown Methodology Specification* [online]. [Accessed: 3 April 2025]. Available at: https://developer.arm.com/documentation/109542/latest/

**24. Arm Ltd.** *Arm Neoverse V1 Core: Performance Analysis Methodology* [online]. White Paper. [Accessed: 3 April 2025]. Available at: https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/neoverse-v1-core-performance-analysis.pdf

**25. SIFIVE, Inc.** *Top-Down Microarchitecture Analysis Approximation for RISC-V Cores* [online]. In: *SC Workshops (SC-W'24)*, 2024. [Accessed: 3 April 2025]. Available at: https://conferences.computer.org/sc-wpub/pdfs/SC-W2024-6oZmigAQfgJ1GhPL0yE3pS/555400b666/555400b666.pdf

**26. ICE-RLAB.** *Icicle: Open-Source Hardware Support for Top-Down Microarchitecture Analysis on RISC-V* [online]. GitHub repository and linked paper. [Accessed: 3 April 2025]. Available at: https://github.com/ice-rlab/Icicle

**27. Score-P Project.** *Score-P Top-Down Microarchitecture Analysis Plugin* [online]. GitHub repository. [Accessed: 3 April 2025]. Available at: https://github.com/score-p/scorep_plugin_topdown

**28. (Various authors).** *Performance Debugging through Microarchitectural Sensitivity Analysis* [online]. arXiv preprint arXiv:2412.13207, 2024. [Accessed: 3 April 2025]. Available at: https://arxiv.org/abs/2412.13207

**29. (Various authors).** *Dissecting RISC-V Performance using Top-Down Analysis* [online]. arXiv preprint arXiv:2507.22451, 2025. [Accessed: 3 April 2025]. Available at: https://arxiv.org/abs/2507.22451

# Appendix - All Metrics

A complete tree with all the metrics can be found here. Those metrics covered in this document have been highlighted.

- Retiring
    - Light Operations
        - FP Arithmetic
            - FP x87
            - FP Scalar
            - FP Vector
                - 128b FP Vector
                - 256b FP Vector
                - 512b FP Vector
        - Integer Operations
            - 128b Integer Vector Operations
            - 256b Vector Operations
        - Memory Operations
        - Fused Instructions
        - Non Fused Branches
        - Other
            - Nop Instructions
            - Shuffles_256b
    - Heavy Operations
        - Few Uops Instructions
        - Microcode Sequencer
            - Assists
                - Page Faults
                - FP Assists
                - AVX Assists
            - CISC
- Front-End Bound
    - Front-End Latency
        - ICache Misses
        - ITLB Overhead
        - Branch Resteers
            - Mispredicts Resteers
            - Clears Resteers
            - Unkown Branches
        - MS Switches
        - Length Changing Prefixes
        - DSB Switches
    - Front-End Bandwidth
        - Front-End Bandwidth MITE

            - Decoder-0 Alone
        - Front-End Bandwidth DSB
        - (Info) DSB Coverage
        - (Info) DSB Misses
- Bad Speculation
    - Branch Mispredict
    - Machine Clears
- Back-End Bound
    - Memory Bound
        - L1 Bound
            - DTLB Overhead
                - Load STLB Hit
                - Load STLB Miss
            - Loads Blockes by Store Forwarding
            - Lock Latency
            - Split Loads
            - 4k Aliasing
            - FB Full
        - L2 Bound
        - L3 Bound
            - Contested Accesses
            - Data Sharing
            - L3 Latency
            - SQ Full
        - DRAM Bound
            - Memory Bandwidth
                - MBA Stalls
            - Memory Latency
                - Local Memory
                - Remote Memory
                - Remote Cache
        - Store Bound
            - Store Latency
            - False Sharing
            - Split Stores
            - Streaming Stores
            - DTLB Store Overhead
                - Store STLB Hit
                - Store STLB Miss
    - Core Bound
        - Divider
        - Serializing Operations
            - Slow Pause
            - C01 Wait

        - C02 Wait
        - Memory Fence
- AMX Busy
- Port Utilization
    - Cycles of 0 Ports Utilized
        - Mixing Vectors
    - Cycles of 1 Ports Utilized
    - Cycles of 2 Ports Utilized
    - Cycles of 3+ Ports Utilized
        - ALU Operation Utilization
            - Port 0
            - Port 1
            - Port 6
        - Load Operation Utilization
        - Store Operation Utilization