A Top-Down Method for Performance Analysis and Counters Architecture

Ahmad Yasin
ahmad.yasin@intel.com
Intel Corporation, Architecture Group

v1.02

Abstract

Optimizing an application's performance for a given microarchitecture has become painfully difficult. Increasing microarchitecture complexity, workload diversity, and the unmanageable volume of data produced by performance tools increase the optimization challenges. At the same time resource and time constraints get tougher with recently emerged segments. This further calls for accurate and prompt analysis methods.

In this paper a Top-Down Analysis is developed – a *practical* method to *quickly* identify *true bottlenecks* in out-of-order processors. The developed method uses designated performance counters in a structured hierarchical approach to quickly and, more importantly, correctly identify dominant performance bottlenecks. The developed method is adopted by multiple in-production tools including VTune. Feedback from VTune average users suggests that the analysis is made easier thanks to the simplified hierarchy which avoids the high-learning curve associated with microarchitecture details. Characterization results of this method are reported for the SPEC CPU2006 benchmarks as well as key enterprise workloads. Field case studies where the method guides software optimization are included, in addition to architectural exploration study for most recent generations of Intel Core™ products.

The insights from this method guide a proposal for a novel performance counters architecture that can determine the true bottlenecks of a *general* out-of-order processor. Unlike other approaches, our analysis method is *low-cost* and already featured in *in-production* systems – it requires just *eight simple* new performance events to be added to a traditional PMU. It is *comprehensive* – no restriction to predefined set of performance issues. It accounts for *granular bottlenecks* in super-scalar cores, missed by earlier approaches.

1. Introduction

The primary aim of performance monitoring units (PMUs) is to enable software developers to effectively tune their workload for maximum performance on a given system. Modern processors expose hundreds of performance events, any of which may or may not relate to the bottlenecks of a particular workload. Confronted with a huge volume of data, it is a challenge to determine the true bottlenecks out of these events. A main contributor to this, is the fact that these performance events were historically defined in an ad-doc bottom-up fashion, where PMU designers attempted to cover key issues via "dedicated miss events" [1]. Yet, how does one pin-point performance issues that were not explicitly foreseen at design time?

Bottleneck identification has many applications: computer architects can better understand resource demands of emerging workloads. Workload characterization often uses data of raw event counts. Such unprocessed data may not necessary point to the right bottlenecks the architects should tackle. Compiler writers can determine what Profile Guided Optimization (PGO) suit a workload more effectively and with less overhead. Monitors of virtual systems can improve resource utilization and minimize energy.

In this paper, we present a Top-Down Analysis - a *feasible*, *fast* method that identifies *critical* bottlenecks in out-of-order CPUs. The idea is simple - a structured drill down in a hierarchical manner, guides the user towards the right area to investigate. Weights are assigned to nodes in the tree to guide users to focus their analysis efforts on issues that indeed matter and disregard insignificant issues. For instance, say a given application is significantly hurt by instruction fetch issues; the method categorizes it as Frontend Bound at the uppermost level of the tree. A user/tool is expected to drill down (only) on the Frontend sub-tree of the hierarchy. The drill down is recursively performed until a tree-leaf is reached. A leaf can point to a specific stall of the workload, or it can denote a subset of issues with a common micro-architectural symptom which are likely to limit the application's performance.

We have featured our method with the Intel 3rd generation Core™ codenamed Ivy Bridge. Combined with the hierarchical approach, a small set of Top-Down oriented counters are used to overcome bottleneck identification challenges (detailed in next section). Multiple tools have adopted our method including VTune [2] and an add-on package to the standard Linux perf utility [3]. Field experience with the method has revealed some performance issues that used to be underestimated by traditional methods. Finally, the insights from this method are used to propose a novel performance counters architecture that can determine the true bottlenecks of *general* out-of-order architecture, in a top down approach.

The rest of this paper is organized as follows. Section 2 provides a background and discusses the challenges with bottleneck identification in out-of-order CPUs. The Top-Down Analysis method and its abstracted metrics are introduced in Section 3. In Section 4, novel low-cost counters architecture is proposed to obtain these metrics. Results on popular workloads as well as sample use-cases are presented in Section 5. Related work is discussed in Section 6 and finally, Section 7 concludes and outlines future work.

2. Background

Modern high-performance CPUs go to great lengths to keep their execution pipelines busy, applying techniques such as large-window out-of-order execution, predictive speculation, and hardware prefetching. Across a broad range of traditional workloads, these high-performance architectures have been largely successful at executing arbitrary code at a high rate of instructions-per-cycle (IPC). However, with these sophisticated super-scalar out-of-order machines attempting to operate so "close to the edge", even small performance hiccups can limit a workload to perform far below its potential. Unfortunately, identifying true performance limiters from among the many inconsequential issues that can be tolerated by these CPUs has remained an open problem in the field.

From a bird's eye view, the pipeline of modern out-of-order CPU has two main portions: a frontend and a backend. The frontend is responsible for fetching instructions from memory and translating them into micro-operations (uops). These uops are fed to the backend portion. The backend is responsible to schedule, execute and commit (retire) these uops per original program's order. So as to keep the machine balanced, delivered uops are typically buffered in some "ready-uops-queue" once ready for consumption by the backend. An example block diagram for the Ivy Bridge microarchitecture, with underlying functional units is depicted in *Figure 1*.

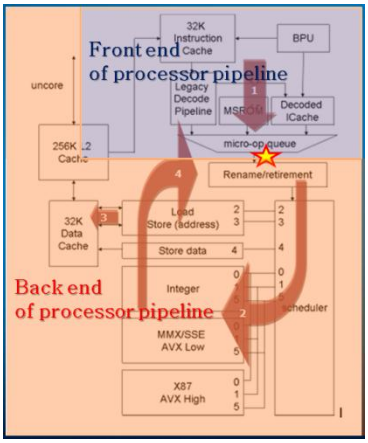


Figure 1: Out-of-order CPU block diagram - Intel Core™

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.

We address those gaps as follows. A major category named "Bad Speculation" (defined later) is placed at the top of the hierarchy. It accounts for stalls due to incorrect predictions as well as resources wasted by execution of incorrect paths. Not only does this bring the issue to user's first attention, but it also simplifies requisites from hardware counters used elsewhere in the hierarchy. We introduce a dozen truly Top-Down designated counters to let us deal with other points. We found that determining what pipeline stage to look at and "to count when matters", play a critical role in addressing (1) and (3). For example, instead of total memory access duration, we examine just the sub-duration when execution units are underutilized as a result of pending memory access. Calling for generic events, not tied to "dedicated miss events" let us deal with (4). Some of these are occupancy events[1] in order to deal with (5).

3. Top-Down Analysis

Top-Down Analysis methodology aims to determine performance bottlenecks correctly and quickly. It guides users to focus on issues that really matter during the performance optimization phase. This phase is typically performed within the time and resources constraints of the overall application development process. Thus, it becomes more important to quickly identify the bottlenecks.

The approach itself is straightforward: Categorize CPU execution time at a high level first. This step flags (reports high fraction value) some domain(s) for possible investigation. Next, the user can drill down into those flagged domains, and can safely ignore all non-flagged domains. The process is repeated in a hierarchical manner until a specific performance issue is determined or at least a small subset of candidate issues is identified for potential investigation.

In this section we first overview the hierarchy structure, and then present the heuristics behind the higher levels of the hierarchy.

3.1. The Hierarchy

The hierarchy is depicted in *Figure 2*. First, we assume the user has predefined criteria for analysis. For example, a user might choose to look at an application's hotspot where at least 20% of execution time is spent. Another example is to analyze why a given hotspot does not show expected speedup from

[1] An occupancy event is capable to increment by more than 1 in a given cycle when a certain condition is met for multiple entities

one hardware generation to another. Hotspot can be a software module, function, loop, or a sequence of instructions across basic blocks.

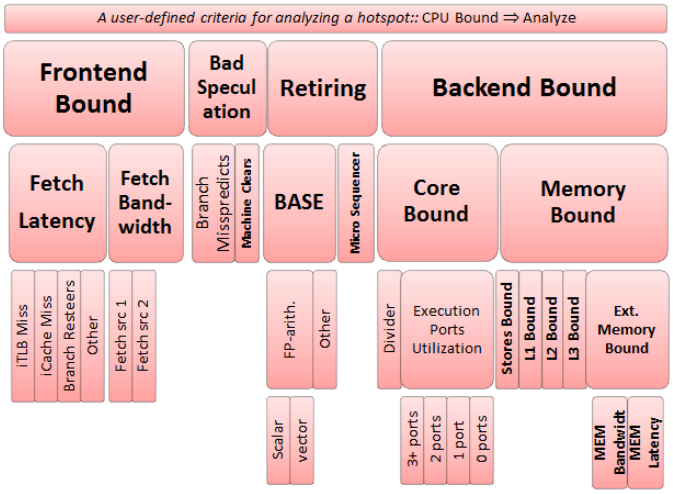


Figure 2: The Top-Down Analysis Hierarchy

Top-Down breakdown is applied to the interesting hotspots where available pipeline slots are split into four basic categories: Retiring, Bad Speculation, Frontend Bound and Backend Bound. These terms are defined in the following sub-sections. The best way to illustrate this methodology is through an example. Take a workload that is limited by the data cache performance. The method flags Backend Bound, and Frontend Bound will not be flagged. This means the user needs to drill down at the Backend Bound category as next step, leaving alone all Frontend related issues. When drilling down at the Backend, the Memory Bound category would be flagged as the application was assumed cache-sensitive. Similarly, the user can skip looking at non-memory related issues at this point. Next, a drill down inside Memory Bound is performed. L1, L2 and L3-Bound naturally break down the Memory Bound category. Each of them indicates the portion the workload is limited by that cache-level. L1 Bound should be flagged there. Lastly, Loads block due to overlap with earlier stores or cache line split loads might be specific performance issues underneath L1 Bound. The method would eventually recommend the user to focus on this area.

Note that the *hierarchical structure adds a natural safety net when looking at counter values*. A value of an inner node should be disregarded unless nodes on the path from the root to that particular node are all flagged. For example, a simple code doing some divide operations on a memory-resident buffer may show high values for both Ext. Memory Bound and Divider nodes in *Figure 2*. Even though the Divider node itself may have high fraction value, it should be ignored assuming the workload is truly memory bound. This is assured as Backend.CoreBound will not be flagged. We refer to this as **hierarchical-safety property**. Note also that only weights of sibling nodes are comparable. This is due to the fact they are calculated at same pipeline stage. *Comparing fractions of non-sibling nodes is not recommended*.

3.2. Top Level breakdown

There is a need for first-order classification of pipeline activity. Given the highly sophisticated microarchitecture, the first interesting question is how and where to do the first level breakdown? We choose the issue point, marked by the asterisk in *Figure 1*, as it is the natural border that splits the frontend and backend portions of machine. It enables a highly accurate Top-Level classification.

At issue point we classify each pipeline-slot into one of four base categories: Frontend Bound, Backend Bound, Bad Speculation and Retiring, as illustrated by *Figure 3*. If a uop is issued in a given cycle, it would eventually either get retired or cancelled. Thus it can be attributed to either Retiring or Bad Speculation respectively.

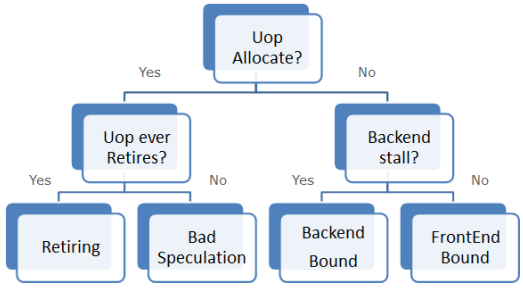


Figure 3: Top Level breakdown flowchart

Otherwise it can be split into whether there was a *backend-stall* or not. A *backend-stall* is a backpressure mechanism the Backend asserts upon resource unavailability (e.g. lack of load buffer entries). In such a case we attribute the stall to the Backend, since even if the Frontend was ready with more uops it would not be able to pass them down the pipeline. If there was no backend-stall, it means the Frontend should have delivered some uops while the Backend was ready to accept them; hence we tag it with Frontend Bound. This *backend-stall* condition is a key one as we outline in *FetchBubbles* definition in next section.

In fact the classification is done at pipeline slots granularity as a superscalar CPU is capable of issuing multiple uops per cycle. This makes the breakdown very accurate and robust which is a necessity at the hierarchy's top level. This accurate classification distinguishes our method from previous approaches in [1][5][6].

3.3. Frontend Bound category

Recall that Frontend denotes the first portion of the pipeline where the branch predictor predicts the next address to fetch, cache lines are fetched, parsed into instructions, and decoded into micro-ops that can be executed later by the Backend. Frontend Bound denotes when the frontend of the CPU undersupplies the backend. That is, the latter would have been willing to accept uops.

Dealing with Frontend issues is a bit tricky as they occur at the very beginning of the long and buffered pipeline. This means in many cases transient issues will not dominate the actual performance. Hence, it is rather important to dig into

this area only when Frontend Bound is flagged at the Top-Level. With that said, we observe in numerous cases the Frontend supply bandwidth can dominate the performance, especially when high IPC applies. This has led to the addition of dedicated units to hide the fetch pipeline latency and sustain required bandwidth. The Loop Stream Detector as well as Decoded I-cache (i.e. DSB, the Decoded-uop Stream Buffer introduced in Sandy Bridge) are a couple examples from Intel Core [7].

Top-Down further distinguishes between latency and bandwidth stalls. An i-cache miss will be classified under **Frontend Latency Bound,** while inefficiency in the instruction decoders will be classified under **Frontend Bandwidth Bound**. Ultimately, we would want these to account for only when the rest of pipeline is likely to get impacted, as discussed earlier.

Note that these metrics are defined in Top-Down approach; Frontend Latency accounts for cases that lead to fetch starvation (the symptom of no uop delivery) regardless of what has caused that. Familiar i-cache and i-TLB misses fit here, but not only these. For example, [4] has flagged Instruction Length Decoding as a fetch bottleneck. It is CPU-specific, hence not shown in *Figure 2*. **Branch Resteers** accounts for delays in the shadow of pipeline flushes e.g. due to branch misprediction. It is tightly coupled with Bad Speculation (where we elaborate on misprediction costs).

The methodology further classifies bandwidth issues per fetch-unit inserting uops to the uops-ready-queue. Instruction Decoders are commonly used to translate mainstream instructions into uops the rest of machine understands - That would be one fetch unit. Also sophisticated instruction, like CPUID, typically have dedicated unit to supply long uop flows. That would be 2nd fetch unit and so on.

3.4. Bad Speculation category

Bad Speculation reflects slots wasted due to incorrect speculations. These include two portions: slots used to issue uops that do not eventually retire; as well as slots in which the issue pipeline was blocked due to recovery from earlier miss-speculations. For example, uops issued in the shadow of a mispredicted branch would be accounted in this category. Note third portion of a misprediction penalty deals with how quick is the fetch from the correct target. This is accounted in Branch Resteers as it may overlap with other frontend stalls.

Having Bad Speculation category at the Top-Level is a key principle in our Top-Down Analysis. It determines the fraction of the workload under analysis that is affected by incorrect execution paths, which in turn dictates the accuracy of observations listed in other categories. Furthermore, this permits nodes at lower levels to make use of some of the many traditional counters, given that most counters in out-of-order CPUs count speculatively. Hence, a high value in Bad Speculation would be interpreted by the user as a "red flag" that need to be investigated first, before looking at other categories. In other words, assuring Bad Speculation is minor not only improves utilization of the available resources, but also increases confidence in metrics reported throughout the hierarchy.

The methodology classifies the Bad Speculation slots into **Branch Misspredict** and **Machine Clears**. While the former is pretty famous, the latter results in similar symptom where the pipeline is flushed. For example, incorrect data speculation generated Memory Ordering Nukes [7] - a subset of Machine Clears. We make this distinction as the next steps to analyze these issues can be completely different. The first deals with how to make the program control flow friendlier to the branch predictor, while the latter points to typically unexpected situations.

3.5. Retiring category

This category reflects slots utilized by "good uops" – issued uops that eventually get retired. Ideally, we would want to see all slots attributed to the Retiring category; that is Retiring of 100% corresponds to hitting the maximal uops retired per cycle of the given microarchitecture. For example, assuming one instruction is decoded into one uop, Retiring of 50% means an IPC of 2 was achieved in a four-wide machine . Hence maximizing Retiring increases IPC.

Nevertheless, a high Retiring value does not necessary mean there is no room for more performance. Microcode sequences such as Floating Point (FP) assists typically hurt performance and can be avoided [7]. They are isolated under **Micro Sequencer** metric in order to bring it to user's attention.

A high Retiring value for non-vectorized code may be a good hint for user to vectorize the code. Doing so essentially lets more operations to be completed by single instruction/uop; hence improve performance. For more details see Matrix-Multiply use-case in Section 5. Since FP performance is of special interest in HPC land, we further breakdown the base retiring category into **FP Arithmetic** with **Scalar** and **Vector** operations distinction. Note that this is an informative field-originated expansion. Other styles of breakdown on the distribution of retired operations may apply.

3.6. Backend Bound category

Backend Bound reflects slots no uops are being delivered at the issue pipeline, due to lack of required resources for accepting them in the backend. Examples of issues attributed in this category include data-cache misses or stalls due to divider being overloaded.

Backend Bound is split into Memory Bound and Core Bound. This is achieved by breaking down backend stalls based on execution units' occupation at every cycle. Naturally, in order to sustain a maximum IPC, it is necessary to keep execution units busy. For example, in a four-wide machine, if three or less uops are executed in a steady state of some code, this would prevent it to achieve a optimal IPC of 4. These suboptimal cycles are called *ExecutionStalls*.

Memory Bound corresponds to execution stalls related to the memory subsystem. These stalls usually manifest with

execution units getting starved after a short while, like in the case of a load missing all caches.

Core Bound on the other hand, is a bit trickier. Its stalls can manifest either with short execution starvation periods, or with sub-optimal execution ports utilization: A long latency divide operation might serialize execution, while pressure on execution port that serves specific types of uops, might manifest as small number of ports utilized in a cycle. Actual metric calculations is described in Section 4.

Core Bound issues often can be mitigated with better code generation. E.g., a sequence of dependent arithmetic operations would be classified as Core Bound. A compiler may relieve that with better instruction scheduling. Vectorization can mitigate Core Bound issues as well; as demonstrated in Section 5.5.

3.7. Memory Bound breakdown (within Backend)

Modern CPUs implement three levels of cache hierarchy to hide latency of external memory. In the Intel Core case, the first level has a data cache (L1D). L2 is the second level shared instruction and data cache, which is private to each core. L3 is the last level cache, which is shared among sibling cores. We assume hereby a three-cache-level hierarchy with a unified external memory; even though the metrics are generic-enough to accommodate other cache- and memory-organizations, including NUMA.

To deal with the overlapping artifact, we introduce a novel heuristic to determine the actual penalty of memory accesses. A good out-of-order scheduler should be able to hide some of the memory access stalls by keeping the execution units busy with useful uops that do not depends on pending memory accesses. Thus the true penalty for a memory access is when the scheduler has nothing ready to feed the execution units. It is likely that further uops are either waiting for the pending memory access, or depend on other unready uops. Significant ExecutionStalls while no demand-load[2] is missing some cache-level, hints execution is likely limited by up to that level itself. *Figure 4* also illustrates how to break ExecutionStalls per cache-level.

For example, L1D cache often has short latency which is comparable to ALU stalls. Yet in certain scenarios, like load blocked to forward data from earlier store to an overlapping address, a load might suffer high latency while eventually being satisfied by L1D. In such scenario, the in-flight load will last for a long period without missing L1D. Hence, it gets tagged under **L1 Bound** per flowchart in *Figure 4*. Load blocks due to 4K Aliasing [7] is another scenario with same symptom. Such scenarios of L1 hits and near caches' misses, are not handled by some approaches [1][5].

Note performance hiccups, as the mentioned L1 Bound scenarios, would appear as leaf-nodes in the hierarchy in *Figure 2*. We skipped listing them due to scope limitation.

[2] Hardware prefetchers are of special treatment. We disregard them as long as they were able to hide the latency from the demand requests.

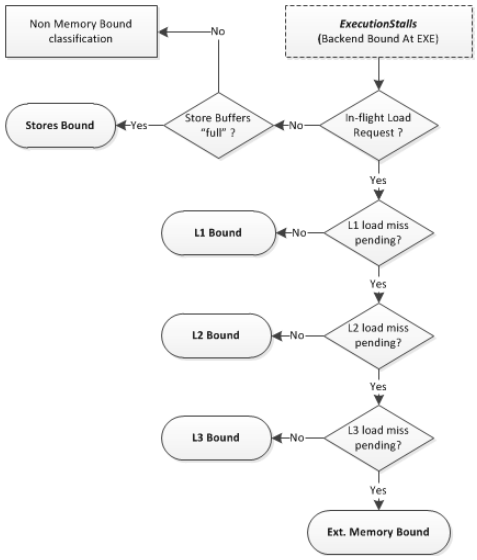


Figure 4: Memory Bound breakdown flowchart

So far, load operations of the memory subsystem were treated. Store operations are buffered and executed post-retirement (completion) in out-of-order CPUs due to memory ordering requirements of x86 architecture. For the most part they have small impact on performance (as shown in results section); they cannot be completely neglected though. Top-Down defined **Stores Bound** metric, as fraction of cycles with low execution ports utilization and high number of stores are buffered. In case both load and store issues apply we will prioritize the loads nodes given the mentioned insight.

Data TLB misses can be categorized under Memory Bound sub-nodes. For example, if a TLB translation is satisfied by L1D, it would be tagged under L1 Bound.

Lastly, a simplistic heuristic is used to distinguish **MEM Bandwidth** and **MEM Latency** under Ext. Memory Bound**.** We measure occupancy of requests pending on data return from memory controller. Whenever the occupancy exceeds a certain threshold, say 70% of max number of requests the memory controller can serve simultaneously, we flag that as potentially limited by the memory bandwidth. The remainder fraction will be attributed to memory latency.

4. Counters Architecture

This section describes the hardware support required to feature the described Top-Down Analysis. We assume a baseline PMU commonly available in modern CPU (e.g. x86 or ARM). Such a PMU offers a small set of general counters capable of counting performance events. Nearly a dozen of events are sufficient to feature the key nodes of the hierarchy. In fact, only eight designated new events are required. The rest can be found in the PMU already today – these are marked with asterisk in *Table 1*. For example, *TotalSlots* event can be calculated with the basic *Clockticks* event. Additional PMU legacy events may be used to further expand the hierarchy, thanks to the *hierarchical-safety property* described in Section 3.

It is noteworthy that a low-cost hardware support is required. The eight new events are easily implementable. They

rely on design local signals, possibly masked with a stall indication. Neither at-retirement tagging is required as in IBM POWER5 [6], nor complex structures with latency counters as in Accurate CPI Stacks proposals [1][8][9].

4.1. Top-Down Events

The basic Top-Down generic events are summarized in *Table 1*. Please refer to *Appendix 1* for the Intel implementation of these events. Notice there, an implementation can provide simpler events and yet get fairly good results.

Table 1: Definitions of Top-Down performance events

Event	Definition
TotalSlots*	Total number of issue-pipeline slots.
SlotsIssued*	Utilized issue-pipeline slots to issue operations
SlotsRetired*	Utilized issue-pipeline slots to retire (complete) operations
FetchBubbles	Unutilized issue-pipeline slots *while there is no backend-stall*
RecoveryBubbles	Unutilized issue-pipeline slots due to recovery from earlier miss-speculation
BrMispredRetired*	Retired miss-predicted branch instructions
MachineClears*	Machine clear events (pipeline is flushed)
MsSlotsRetired*	Retired pipeline slots supplied by the micro-sequencer fetch-unit
OpsExecuted*	Number of operations executed in a cycle
MemStalls.AnyLoad	Cycles with no uops executed and at least 1 in-flight load that is not completed yet
MemStalls.L1miss	Cycles with no uops executed and at least 1 in-flight load that has missed the L1-cache
MemStalls.L2miss	Cycles with no uops executed and at least 1 in-flight load that has missed the L2-cache
MemStalls.L3miss	Cycles with no uops executed and at least 1 in-flight load that has missed the L3-cache
MemStalls.Stores	Cycles with few uops executed and no more stores can be issued
ExtMemOutstanding	Number of outstanding requests to the memory controller every cycle

4.2. Top-Down Metrics

The events in *Table 1* can be directly used to calculate the metrics using formulas shown in *Table 2*. In certain cases, a flavor of the baseline hardware event is used[3]. *Italic #-prefixed metric* denotes an auxiliary expression.

Table 2: Formulas for Top-Down Metrics

Metric Name	Formula
Frontend Bound	FetchBubbles / TotalSlots
Bad Speculation	(SlotsIssued – SlotsRetired + RecoveryBubbles) / TotalSlots
Retiring	SlotsRetired / TotalSlots
Backend Bound	1 – (Frontend Bound + Bad Speculation + Retiring)
Fetch Latency Bound	FetchBubbles[≥ #MIW] / Clocks
Fetch Bandwidth Bound	Frontend Bound – Fetch Latency Bound
#BrMispredFraction	*BrMispredRetired / (BrMispredRetired + MachineClears)*
Branch Mispredicts	#BrMispredFraction * Bad Speculation
Machine Clears	Bad Speculation – Branch Mispredicts
MicroSequencer	MsSlotsRetired / TotalSlots
BASE	Retiring – MicroSequencer
#ExecutionStalls	*(ΣOpsExecuted[= FEW]) / Clocks*
Memory Bound	(MemStalls.AnyLoad + MemStalls.Stores) / Clocks
Core Bound	#ExecutionStalls – Memory Bound
L1 Bound	(MemStalls.AnyLoad – MemStalls.L1miss) / Clocks
L2 Bound	(MemStalls.L1miss – MemStalls.L2miss) / Clocks
L3 Bound	(MemStalls.L2miss – MemStalls.L3miss) / Clocks
Ext. Memory Bound	MemStalls.L3miss / Clocks
MEM Bandwidth	ExtMemOutstanding[≥ *THRESHOLD*] / ExtMemOutstanding[≥ 1]
MEM Latency	(ExtMemOutstanding[≥ 1] / Clocks) – MEM Bandwidth

Note *ExecutionStall* denotes sub-optimal cycles in which no or *few* uops are executed. A workload is unlikely to hit max IPC in such case. While these thresholds are implementation-specific, our data suggests cycles with 0, 1 or 2 uops executed are well-representing Core Bound scenarios at least for Sandy Bridge-like cores.

5. Results

In this section, we present Top-Down Analysis results for the SPEC CPU2006 benchmarks in single-thread (1C) and multi-copy (4C) modes with setup described in *Table 3*. Then, an across-CPUs study demonstrates an architecture exploration use-case. As Frontend Bound tends to be less of a bottleneck in CPU2006, results for key server workloads are included. Lastly, we share a few use-cases where performance issues are tuned using Top-Down Analysis.

Table 3: Baseline system setup parameters

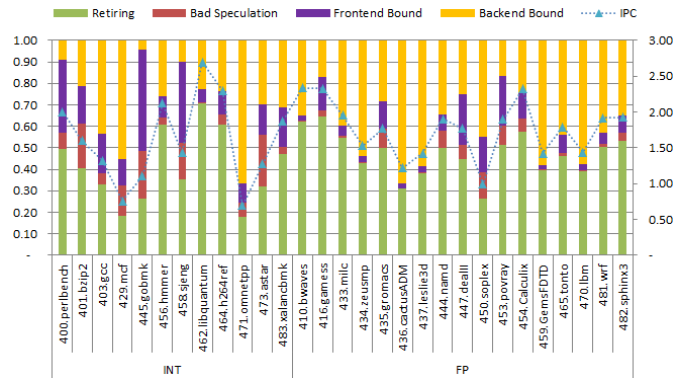
Processor	Intel® Core™ i7-3940XM (Ivy Bridge). 3 GHz fixed frequency. A quadcore with 8MB L3 cache. Hardware prefetchers enabled.
Memory	8GB DDR3 @1600 MHz
OS	Windows 8 64-bit
Benchmark	SPEC CPU 2006 v1.2 (base/rate mode)
Compiler	Intel Compiler 14 (SSE4.2 ISA)

5.1. SPEC CPU2006 1C

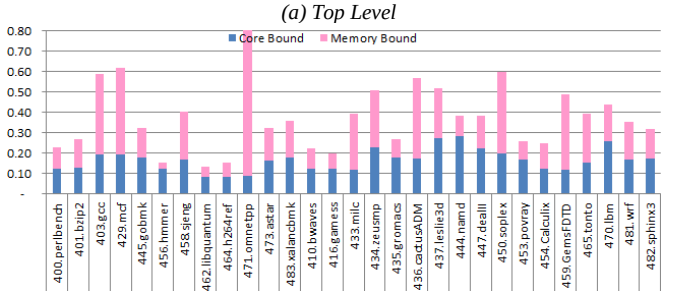
At the Top Level, *Figure 5a* suggests diverse breakdown of the benchmark's applications. Performance wise, the Retiring category is close to 50% which aligns with aggregate Instruction-Per-Cycle (IPC) of ~1.7 measured for same set of runs. Recall 100% Retiring means four retired uops-per-cycle while for SPEC CPU2006 an instruction is decoded into slightly more than one uop on average. Note how Retiring correlates well with IPC, included to cross-validate with an established metric.

Overall Backend Bound is dominant. So we drill down into it in next diagrams in *Figure 5*. The Backend Level diagram guides the user whether to look at Core or Memory issues next. For example, 456.hmmer is flagged as Backend.CoreBound. Close check of the top hotspots with VTune, indeed points to loops with tight data-dependent arithmetic instructions.

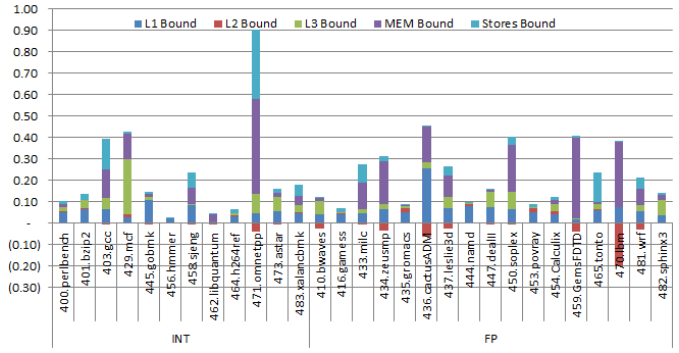
[3] For example, the FetchBubbles[≥ MIW] notation tells to count cycles in which number of fetch bubbles exceed Machine Issue Width (MIW). This capability is called Counter Mask ever available in x86 PMU [10].



(a) Top Level

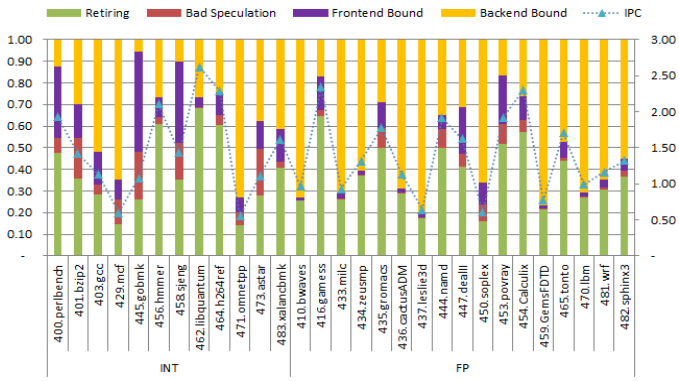


(b) Backend Level

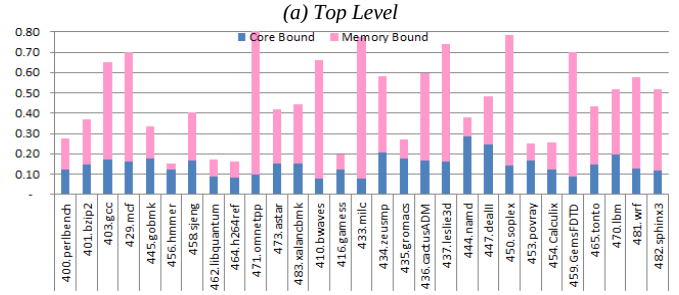


(c) Memory Level

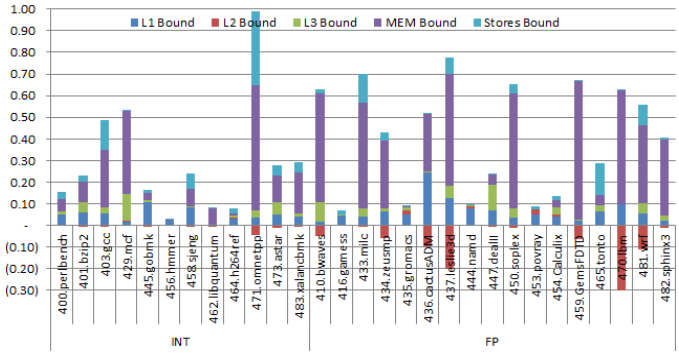
Figure 5: Top-Down Analysis breakdown for SPEC CPU 2006 benchmarks in single-thread mode



(a) Top Level



(b) Backend Level



(c) Memory Level

Figure 6: Top-Down Analysis breakdown for SPEC CPU 2006 benchmarks in multi-core mode (4-copy)

The Integer applications are more sensitive to Frontend Bound and Bad Speculation than the FP applications. This aligns with simulations data using a propriety cycle-accurate simulator, as well as prior analysis by Jaleel [11]. For example, Jaleel's analysis reported that gcc, perlbench, xalancbmk, gobmk, and sjeng have code footprint bigger than 32KB. They are classified as most Frontend Bound workloads. Note how the breakdown eases to assess the relative significance of bottlenecks should multiple apply.

5.2. SPEC CPU2006 4C

Results running 4-copies of these applications are shown in *Figure 6*. Top Level shows similarity to 1-copy. At a closer look, some applications do exhibit much increased Backend Bound. These are memory-sensitive applications as suggested by bigger Memory Bound fractions in *Figure 6b*. This is expected as L3 cache is "shared" among cores. Since an identical thread is running alone inside each physical core and given CPU2006 has minor i-cache misses, Frontend Bound and Bad Speculation in 4-copy roughly did not changed over 1-copy.

For the less-scalable applications, Memory Bound breakdown points to off-core contention when comparing *Figure 6c to 5c*[4]. The key differences occur in applications that are either (I) sensitive to available memory bandwidth, or (II) impacted by shared cache competition between threads. An example of (I) is 470.lbm which is known for its high memory bandwidth requirements [12]. Its large MEM Bound is the primary change between 1- and 4-copy.

A key example of (II) is 482.sphinx3. A close look at Memory Bound breakdown indicates the 4-copy sees reduced L3 Bound, and a greatly increased MEM Bound; capacity contention between threads in the shared L3 cache has forced many more L3 misses. This conclusion can be validated by consulting the working-set of this workload [11]: a single copy demands 8MB (same as LLC capacity) in 1-copy, vs 2MB effective per-core LLC share in 4-copy runs.

Figure 7 shows how off-chip resources are utilized for some FP applications, with 1- and 4-copy side-by-side. The

[4] Negative L2 Bound is due to PMU erratum on L1 prefetchers

bars' height indicates fraction of run time where the memory controller is serving *some* request. "MEM Bandwidth" is the relative portion where *many* requests are being serviced simultanously. Note we could plot these metrics at their native local units, thanks to the hierarchical-safety property. We should consider them carefully though.

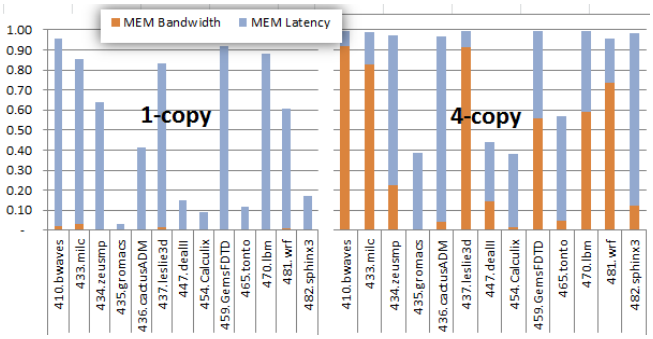


Figure 7: Off-chip comparison of memory-sensitive FP apps

The available 25GB/s bandwidth clearly satisfies demand of 1-copy. The picture changes in 4-copy in different ways. 435.gromacs, 447.dealII, 454.calculix and 465.tonto now spend more memory cycles due to increase of 1.3-3.6x in L3 misses per-kilo instructions as measured by distinct set of performance counters. Note however, they showed on-par Memory- and Core-Bound stall fractions in *Figure 6b*, likely because the out-of-order could mitigate most of these memory cycles. This aligns with measured IPC in range of 1.7-2.3 in 4-copy. In contrast, 410.bwaves, 433.milc, 437.leslie3d and 470.lbm become much more MEM Bound in 4-copy per *Figure 6c*. *Figure 7* tells us that was due to memory latency in 1-copy which turns into memory bandwidth in 4-copy (4x data demand). Top-Down correctly classifies 470.lbm as MEM Bandwidth limited [12].

5.3. Microarchitectures comparison

So far we have shown results for the same system. This section demonstrates how Top-Down can assist hardware architects. *Figure 8* shows Top Level for Intel Core 3rd and 4th generation CPUs, side-by-side for a subset of CPU2006 integer benchmarks. The newer Intel Core has improved front-end where speculative iTLB and i-cache accesses are supported with better timing to improve the benefits of prefetching [7]. This can be clearly noticed for the benefiting benchmarks with reduction in Frontend Bound. This validation adds to the confidence of underlying heuristics invented two generations earlier.

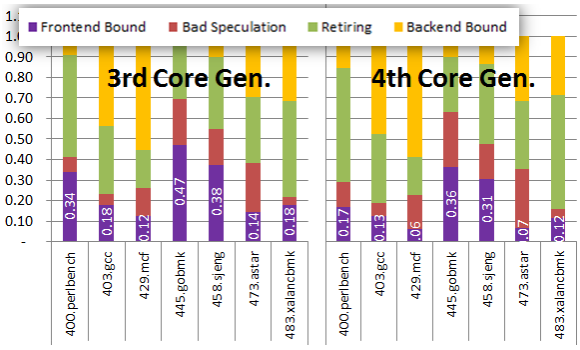


Figure 8: Top Down across-microarchitectures

5.4. Server workloads

Key server workloads' results on Sandy Bridge EP are shown in *Figure 9*. Retiring is lower compared to the SPEC workloads, which conform to the lower IPC domain (a range of 0.4 to 1 is measured). Backend- and Frontend-Bound are more significant given the bigger footprints.

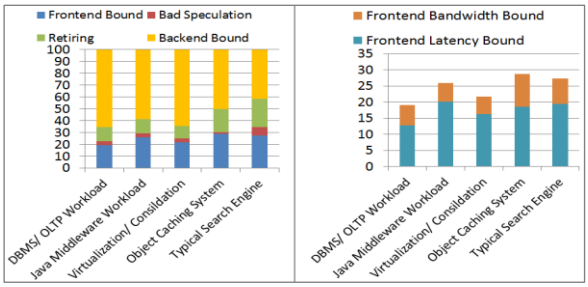


Figure 9: Top and Frontend levels for server workloads

It is interesting to see that the characterization of DBMS workloads generally conforms to [4] who reported these workloads are limited by last-level data cache misses and 1st level i-cache misses a while back.

Within the Frontend, Latency issues are dominant across all server workloads. This is due to more i-cache and i-TLB misses as expected there, in contrast to client workloads whose Frontend Bound was almost evenly split between Latency and Bandwidth issues (not shown due to paper scope limitation).

5.5. Case Study 1: Matrix-Multiply

A matrix-multiply textbook kernel is analyzed with Top-Down. It demos the iterative nature of performance tuning.

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.

5.6. Case Study 2: False Sharing

A university class educates students on multithreading pitfalls through an example to parallelize a serial compute-bound code. First attempt has no speedup (or, a slowdown)

due to False Sharing. False Sharing is a multithreading hiccup, where multiple threads contend on different data-elements mapped into the same cache line. It can be easily avoided by padding to make threads access different lines.

Table 5: Results of tuning False Sharing case

		Multi-thread	
Metric	Single Thread	False Sharing	Fixed
Speedup	1.00x	0.97x	3.77x
IPC	0.90	0.36	0.84
Frontend Bound	0.00	0.02	0.01
Retiring	0.31	0.11	0.30
Bad Speculation	0.00	0.00	0.00
Backend Bound	0.69	0.87	0.69
├-+ Memory Bound	0.19	0.49	0.19
| ├-- L1 Bound	0.19	0.16	0.19
| ├-- L2 Bound	0.00	(0.06)	0.00
| ├-- L3/MEM Bound	0.00	0.06	0.00
| └-- Stores Bound	0.00	0.33	0.00
└-- Core Bound	0.33	0.36	0.36

The single-thread code has modest IPC. Top-Down correctly classifies the first multithreaded code attempt as Backend.Memory.StoresBound (False Sharing must have one thread writing to memory, i.e. a store, to apply). Stores Bound was eliminated in the fixed multithreaded version.

5.7. Case Study 3: Software Prefetch

A customer propriety object-recognition real application is analyzed with Top-Down. The workload is classified as Backend.Memory.ExtMemory.LatencyBound at application scope. Ditto for biggest hotspot function; though the metric fractions are sharper there. This is a symptom of more non-memory bottlenecks in other hotspots.

Table 6: Results of tuning Software Prefetch case

	Original Workload		Software Prefetch	
Metric / *Scope*	*Application*	*Function*	*Application*	*Function*
Time [sec]	70.94	33.91	58.62	13.54
Speedup	1.00x	1.00x	1.21x	2.50x
IPC	1.161	0.85	1.490	*1.36*
Frontend Bound	0.14	0.07	0.17	0.15
Retiring	0.17	0.04	0.19	*0.08*
Bad Speculation	0.04	0.04	0.04	*0.07*
Backend Bound	0.67	0.87	0.62	0.73
├-+ Memory Bound	**0.31**	**0.56**	0.22	*0.42*
| ├-- L1 Bound	-	-	-	-
| ├-- L2 Bound	0.03	-	-	-
| ├-- L3 Bound	0.01	0.01	0.01	*0.02*
| ├-+ MEM Bound	**0.29**	**0.60**	0.22	*0.62*
| | ├ MEM Bandwidth	0.04	0.08	0.10	*0.31*
| | └ MEM Latency	**0.54**	**0.92**	0.39	0.69
| └-- Stores Bound	0.02	-	0.03	*0.01*
└-+ Core Bound	0.24	-	**0.28**	*0.34*

Software Prefetches[10] are planted in the algorithm's critical loop to prefetch data of next iteration. A speedup of 35% per the algorithm-score is achieved, which is translated to 1.21x at workload scope. Note the optimized version shows higher memory-bandwidth utilization and has become more Backend.CoreBound.

6. Related Work

The widely-used naïve-approach is adopted by [4][5] to name a few. While this might work for in-order CPUs, it is far from being accurate for out-of-order CPUs due to: stalls overlap, speculative misses and workload-dependent penalties as elaborated in Sections 2.

IBM POWER5 [6] has dedicated PMU events to aid compute CPI breakdown at retirement (commit) stage. Stall periods with no retirement are counted per type of the next instruction to retire and possibly a miss-event tagged to it. Again this is a predefined set of fixed events picked in a bottom-up way. While a good improvement over naïve-approach, it underestimates frontend misses' cost as they get accounted after the point where the scheduler's queue gets emptied. Levinthal [5] presents a Cycle Accounting method for earlier Intel Core implementations. A flat breakdown is performed at execution-stage, to decompose total cycles into retired, non-retired and stall components. Decomposition of stall components then uses the inadequate naïve-approach as author himself indicates.

In contrast, Top-Down does breakdown at issue-stage, at finer granularity (slots) and avoids summing-up all penalties into one flat breakdown. Rather it drills down stalls in a hierarchical manner, where each level zooms into the appropriate portion of the pipeline. Further, designated Top-Down events are utilized; sampling (as opposed to *counting*) on frontend issues is enabled, as well as breakdown when HT is on. None of these is featured by [5].

Some researchers have attempted to accurately classify performance impacts on out-of-order architectures. Eyerman et al. in [1][9] use a simulation-based interval analysis model in order to propose a counter architecture for building accurate CPI stacks. The presented results show improvements over naïve-approach and IBM POWER5 in terms of being closer to the reference simulation-based model. A key drawback of this approach (and its reference model) is that it restricts all stalls to a fixed set of eight predefined miss events. In [1][4][5] there is no consideration of (fetch) bandwidth issues, and short-latency bottlenecks like L1 Bound. Additionally, high hardware cost is implied due to fairly complex tracking structures as authors themselves later state in [8]. While [8] replaces the original structure with smaller FIFO; extra logic is required for *penalty calculation* and *aggregation to* new *dedicated* counters. This is in comparison with the simple events adopted by our method with no additional counters/logic. We have pointed to more drawbacks in previous sections.

More recently, [13] and [12] proposed instrumentation-based tools to analyze data-locality and scalability bottlenecks, respectively. In [13], average memory latency is sampled with a PMU and coupled with reuse distance obtained through combination of Pin and a cache simulator, in order to prioritize optimization efforts. An offline analyzer maps these metrics back to source code and enables the user to explore the data in hierarchal manner starting from *main* function. [12] presents a method to obtain speedup stacks for a specific type of parallel programs, while accounting for three bottlenecks: cache capacity, external memory bandwidth and synchronization.

These can be seen as advanced optimization-specific techniques that may be invoked from Top-Down once Backend.MemoryBound is flagged. Furthermore, better metrics based on our *MemStalls.L3Miss* event e.g. can be used instead of raw latency value in [13] to quantify when speedup may apply. Examining metrics at higher program scope first, may be applied to our method as already done in VTune's General Exploration view [2]. While [12] estimates speedups

(our method does not), it accounts for subset of scalability bottlenecks. For example, the case in 5.6 is not be covered by their three bottlenecks.

7. Summary and Future Work

This paper presented Top-Down Analysis method - a comprehensive, systematic in-production analysis methodology to identify *critical* performance bottlenecks in out-of-order CPUs. Using designated PMU events in commodity multi-cores, the method adopts a hierarchical classification, enabling the user to zero-in on issues that directly lead to sub-optimal performance. The method was demonstrated to classify critical bottlenecks, across variety of client and server workloads, with multiple microarchitectures' generations, and targeting both single-threaded and multi-core scenarios.

The insights from this method are used to propose a novel *low-cost* performance counters architecture that can determine the true bottlenecks of a *general* out-of-order processor. Only *eight simple* new events are required.

The presented method raises few points on PMU architecture and tools front. Breakdown of few levels require multiple events to be collected simultaneously. Some techniques might tolerate this; such as Sandy Bridge's support of up to eight general-purpose counters [10], or event-multiplexing in the tools [2][3]. Still a better hardware support is desired. Additionally, the ability to pinpoint an identified issue back to the user code can benefit much software developers. While PMU precise mechanisms are a promising direction, some microarchitecture areas are under-covered. Yet, enterprise-class applications impose additional challenges with flat long-tail profiles.

Correctly classifying bottlenecks in the context of hardware *hyper-threading* (HT) is definitely a challenging front. While it was beyond the scope of this paper, the design of some Top Down events, does take HT into account, letting the Top Level works when HT is enabled; but that is just the start. Lastly, While the goal of our method was to *identify* critical bottlenecks, it does not *gauge the speedup* should underlying issues be fixed. Generally, even to determine whether an issue-fix will be translated into speedup (at all) is tricky. A workload often moves to the next critical bottleneck. [12] has done nice progress to that end in scalability bottlenecks context.

Acknowledgements

The author would like to thank the anonymous reviewers at Intel; Joseph Nuzman and Vish Viswanathan in particular for their insightful comments, Tal Katz for data collection and Mashor Housh for a close technical write-up review.

References

[1] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *ACM SIGOPS Operating Systems Review*, 2006, vol. 40, pp. 175–184.

[2] Intel Corporation, "Intel® VTune™ Amplifier XE 2013." [Online].

[3] A. Carvalho, "The New Linux 'perf' tools," presented at the Linux Kongress, 2010.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," in *Proceedings of the 25th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1999, pp. 266–277.

[5] D. Levinthal, "Performance Analysis Guide for Intel Core i7 Processor and Intel® Xeon 5500 processors." Intel, 2009.

[6] A. Mericas, Vianney, Duc, Maron, Bill, Thomas Chen, Steve Kunkel, and Bret Olszewski, "CPI analysis on POWER5, Part 2: Introducing the CPI breakdown model," 2006. [Online]

[7] Intel Corporation, "Intel® 64 and IA-32 Architectures Optimization Reference Manual," *Intel*. [Online].

[8] O. Allam, S. Eyerman, and L. Eeckhout, "An efficient CPI stack counter architecture for superscalar processors," in *GLSVLSI*, 2012.

[9] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A top-down approach to architecting CPI component performance counters," *Micro, IEEE*, vol. 27, no. 1, pp. 84–93, 2007.

[10] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 2013. [Online]

[11] A. Jaleel, "Memory Characterization of Workloads Using Instrumentation-Driven Simulation," 2010. [Online]. Available: http://www.jaleels.org/ajaleel/workload/.

[12] D. Eklov, N. Nikoleris, and E. Hagersten, "A Profiling Method for Analyzing Scalability Bottlenecks on Multicores," 2012.

[13] X. Liu and J. Mellor-Crummey, "Pinpointing data locality bottlenecks with low overhead," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, 2013, pp. 183–193.

Appendix 1

Intel Core™ microarchitecture is a 4-wide issue machine. *Table 7* summarizes the metrics implementation using the Ivy Bridge PMU event names. Some of the Top-Down designated events are not directly available in hardware; instead a formula is supplied to approximate metric from available events. Note metrics that do not appear in the table have nothing specific to the Intel implementation and can be used as is from *Table 2*.

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)*
#RetireUopFraction	*UOPS_RETIRED.RETIRE_SLOTS / UOPS_ISSUED.ANY*
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	#L3HitFraction * CYCLE_ACTIVITY.STALLS_L2_MISS / Clocks
Ext. Memory Bound	(1 - #L3HitFraction) * CYCLE_ACTIVITY.STALLS_L2_MISS / Clocks
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