A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors

AHMAD YASIN, University of Haifa and Intel Corporation
JAWAD HAJ-YAHYA, ETH Zurich
YOSI BEN-ASHER, University of Haifa
AVI MENDELSON, Technion

The slowdown in technology scaling puts architectural features at the forefront of the innovation in modern processors. This article presents a Metric-Guided Method (MGM) that extends Top-Down analysis with carefully selected, dynamically adapted metrics in a structured approach. Using MGM, we conduct two evaluations at the microarchitecture and the Instruction Set Architecture (ISA) levels. Our results show that simple optimizations, such as improved representation of CISC instructions, broadly improve performance, while changes in the Floating-Point execution units had mixed impact. Overall, we report 10 architectural insights—at the microarchitecture, ISA, and compiler fronts—while quantifying their impact on the SPEC CPU benchmarks.

CCS Concepts: • General and reference → Performance; • Applied computing → Microcomputers;

Additional Key Words and Phrases: Performance analysis, performance comparison, benchmarking, microarchitecture, instruction set architecture, compiler code generation

ACM Reference format:
Ahmad Yasin, Jawad Haj-Yahya, Yosi Ben-Asher, and Avi Mendelson. 2019. A Metric-Guided Method for Discovering Impactful Features and Architectural Insights for Skylake-Based Processors. ACM Trans. Archit. Code Optim. 16, 4, Article 46 (December 2019), 25 pages.
https://doi.org/10.1145/3369383

1 INTRODUCTION

Motivation: The computer architecture community faces an important challenge—Moore’s law is slowing down, stressing traditional assumptions around faster systems every year, while at the same time the demand for performance is growing. Examples of the latter include datacenters with ever growing volumes of data, new diverse workloads in the cloud, and smarter edge devices [1]. A detailed profiling of Google’s datacenters [2] reported a high degree of workload diversity with bi-modal ILP: despite memory stalls, operations are executed in bursts, which exploits wide and deep pipelines. This proofs analysis of production systems is insightful and advocates for...
high-performance general-purpose processors. While performance improves with newer processors, it remains unclear what architectural enhancements affect a workload of interest—the problem that is addressed by this article.

**Benefits:** Multiple usages that ultimately improve system performance are enabled once impactful architectural features are identified. CPU architects can reflect what is actually critical to improve in an earlier generation design. Compiler writers can focus optimizations to exploit the beneficial features. For example, tune code generation to use instructions that become more efficient or avoid using inefficient ones. Platform engineers and application developers can focus their limited efforts on performance optimization. Resource managers can reasonably increase utilization of heterogeneous datacenters by leveraging characteristics of applications.

**Difficulty:** Evaluating an architectural feature in a real system is not an easy task. Multiple factors challenge this: general-purpose CPUs run a large spectrum of workloads with mixed bottlenecks. For example, profiling results shared by this article, indicate that integer benchmarks highly benefit from better instruction fetch, while Floating-Point (FP) benchmarks benefit from an optimized execution engine. Therefore, hardware vendors introduce several changes at once in new generations. While this can benefit certain workloads (as Figure 2 demonstrates), it induces inter-feature interactions. Background on modern architectures and associated challenges is provided in Section 2.

This article proposes a new analysis method that can solve the above-mentioned problem. Figure 1 depicts the building blocks of our Metric-Guided Method (MGM). MGM is based on two key ideas. First, MGM starts with a small set of microarchitecture-abstracted bottlenecks (obtained from the Top-Down analysis method) and additional performance metrics (e.g., IPC, Instructions per cycle). In order to find features that potentially impact a set of workloads, differential analysis steers the search to a particular workload that exposes drifts in certain metrics. Second, once a feature is found, we handcraft a designated microbenchmark (kernel) to examine it. Step helps to define a specific metric capturing the feature (the root-cause) and possibly a generic metric capturing the symptom. The latter enhances the search process in next iterations.

---

1Top-Down Microarchitecture Analysis [8] (TMA) is a cycle-accounting method that identifies the costs of performance bottlenecks for out-of-order cores using microarchitecture-abstracted metrics organized in one simplified hierarchy.
Fig. 2. Instruction-per-cycle (IPC) for 400.perlbench on recent x86 processors (configuration details in Table 2).

**Demo of the method:** MGM is better realized through examples. The following two scenarios demonstrate how using MGM we are able to uncover architectural insights of the Skylake [6] processor:

1. In the first scenario, we wanted to understand the IPC increase for the benchmark in Figure 2 when comparing Skylake to a previous microarchitecture (uarch) generation. Step 3 of Figure 1 pointed to a reduction in Frontend Bound [8], which helped to focus the analysis on a subset of fetch-related changes. A kernel was handcrafted in 5 in order to rule out few documented changes (see Section 5.2). The kernel, which iteratively increased code size via loop unrolling, verified the reduction was due to an arch feature: improved i-cache miss handling. The kernel further helped to mark the workloads that are impacted by this feature in 6. Such markup allows for other fetch-related changes to be discovered in next iterations.

2. The second scenario is triggered by a reduction in Core Bound [8] in the same experiment. A close analysis reveals that the conditional move (CMOV) instruction is decoded into a fewer number of uops (micro-operations). Therefore, in 7 we appended a generic metric called Uops Per Instruction (UPI; see Section 4.3) to the set of metrics to be monitored in next iterations. This action helped to find other changes with the same symptom of reduction in UPI (including when these other changes do not manifest in Abstracted_Bottlenecks).

It should be noted that such architectural insights may not be discovered by Top-Down metrics, which are abstract by definition. Thereby, MGM employs a strategic bottom-up step (the dashed upward loop in Figure 1) to facilitate the discovery of such insights. We provide a contrast of MGM versus Top-Down in Section 3.

**Experimental evaluations:** We demonstrate MGM through two evaluations on Intel processors: (1) Microarchitecture evaluation, where the same binaries are analyzed on different processors, and (2) ISA evaluation, where different binaries are analyzed on the same processor. Both evaluations identify concrete reasons that lead to performance differences and under what conditions. For example, in (1) we observe that improved representation of CISC instructions improves performance across the board while changes to certain FP instructions have mixed implications. In (2), we reveal that emitting Fused Multiply-Add (FMA) instruction by the compiler has also enabled better instructions for memory accesses (an unexpected side effect).
This article makes three primary contributions:

— A description of a novel Metric-Guided Method (MGM) that employs a structured analysis process and successfully identifies impactful performance features in modern processors (Section 3). MGM extends the Top-Down analysis method with three unique techniques: initial inclusion of a performance metric, dynamic inclusion of bottom-up metrics, and handcrafting a kernel per each architectural feature.

— A detailed performance characterization report for SPEC CPU2006 benchmarks on the Skylake processor (Section 5). Overall, we report 10 architectural insights at the microarchitecture (7), ISA (1), and compiler (2) fronts, and map how all these insights impact the individual benchmarks. Three out of these insights are uniquely documented by our analysis.

— A set of microbenchmarks that isolates the identified architectural features (Section 4). The microbenchmarks can be used as templates when exploring different architectures.

2 BACKGROUND AND MOTIVATION

This section provides some motivation and necessary architecture and performance analysis background needed for later sections.

2.1 More on Motivation

Figure 2 shows significant speedup for a particular workload on recent processors from Intel and AMD. While both Skylake [6] and Zen [5] microarchitectures show significant speedup over prior generations, the primary question this article explores is what are the architectural changes that enable this speedup?

Processor architects lump multiple architectural modifications into a new processor generation. The publicly available documentation covers a subset of those modifications and is often sanitized. More specifically, architects use a certain set of benchmarks in order to evaluate candidate architectural features and make decisions at design phase. These benchmarks can potentially have different characteristics than a particular workload of interest. Thus, a brute-force approach that attempts to evaluate the documented features is not only inefficient (see Section 3.1) but also can be misleading.

2.2 High-Performance Cores

There are three main sections in a modern processor, as depicted in Figure 3: the front-end, the execution engine, and the memory subsystem.

**Front-End:** The in-order front-end is the part of the processor that fetches the instructions to be executed next in the program and prepares them to be used by the later pipeline stages. It aims to supply a high-bandwidth stream of decoded instructions to the out-of-order execution engine, which will perform the actual completion of the instructions. The front-end is equipped with an advanced branch prediction unit that uses the past history of program execution in order to speculate what program paths would be executed next. A predicted instruction address, supplied by the branch predictor, is used to fetch instruction from the Level 2 (L2) cache should the i-cache be missed. In some microarchitectures, instructions are further decoded into basic operations called micro-operations (uops) that the execution engine is able to execute.

**Execution Engine:** At the back-end of a processor, uops are scheduled in an out-of-order manner for execution. The out-of-order execution logic has schedulers and several buffers that are used to re-order the flow of instructions in order to optimize performance as the instructions go down the pipeline and get scheduled for execution. Instructions are reordered such that they can execute
as quickly as their input operands are ready. This out-of-order execution allows instructions in the program following stalling instructions to proceed around them as long as they do not depend on those stalling instructions. Out-of-order execution enables one to maximize usage of the execution units. The execution units are where the instructions (or uops) are actually executed. It includes the register files that store the integer and FP data operand values that the instructions need to execute. The execution units include several types of integer and FP execution units that compute the results and also perform the memory operations (such as load and store accesses).

**Memory Subsystem:** This includes the L1 data-cache (L1D), the unified L2 cache, and the system-on-chip (SoC) interconnect. The L2 cache stores both instructions and data that cannot fit in the L1 caches. The SoC interconnect is connected to the backside of the L2 cache and is used to access off-core caches (e.g., an L3 cache), the main memory when missing all caches, and the system I/O as needed.

The high-level description of the microarchitectures shown in Figure 3 applies to the designs of Intel Skylake [6], AMD Zen [5], and Qualcomm Falkor [7] that power the most competitive mainstream servers [10]. While the principles of an in-order fetch and commit pipelines with an out-of-order execution are common, the details vary. For example, Falkor has a two-level instruction cache (i-cache) while Skylake and Zen offer a standard i-cache pipeline that decodes instructions into uops in addition to a Uop Cache. The Uop Cache avoids heavy x86 decoding by caching the decoded uops [11]; it improves performance by increasing the fetch sustained bandwidth and reduces power too. Intel’s Loop Cache [11] further optimizes short loops by streaming directly from the Uop Queue, which holds fetched uops that are ready for the back-end consumption and is common in all three designs.

### 2.3 Evaluation Challenges

Figuring out the contribution of a particular architectural change is not an easy task. Multiple factors challenge this at the processor and workload fronts.

High-performance CPUs go to great lengths to keep their pipelines busy, applying techniques such as out-of-order execution, superscalar and instruction speculation. While these features...
increase the rate of IPC, they complicate the hardware and hence make performance analysis significantly harder. The key hurdles in performance analysis and software optimization as listed by [8] are as follows:

- Stalls overlap due to out-of-order execution.
- Speculation when CPU fetches instructions or loads data.
- Workload-dependent penalties.
- Hardware performance counters’ limitations.
- Superscalar inaccuracy (poor sub-cycle visibility).

There is no direct way to measure the contribution of a particular processor feature in real systems (e.g., configure a processor with one ALU). Furthermore, even if that was feasible, the performance contribution would not be additive due to inter-feature interactions. In particular, if one tries to avoid a bottleneck, there is a good chance to get blocked by the next bottleneck, that is, “onion peeling.” This applies for both architects and performance engineers as they mitigate bottlenecks through hardware or software changes.

Additionally, the workloads get complicated [2, 12]. Full applications have multiple phases with distinct bottlenecks [13]. Furthermore, performance depends on the input dataset and other platform settings. Datacenter workloads have a high degree of diversity [2]. Even a self-contained benchmark suite like SPEC CPU shows diverse bottlenecks [8].

2.4 Definitions of Counters, Metrics, and Bottlenecks

A performance counter is the raw value of a performance monitoring event as reported by the processor. [14] and [15] are online pages that document all performance counters available for the two processors used by our evaluations.

A performance counter is formatted in a green Courier font. For example, INST_RETIRED.ANY counts the number of instructions that successfully retire.

A metric is a formula that may combine multiple performance counters possibly with some machine-specific parameters.

A bottleneck is a metric class denoting some stall. For example, Frontend_Bound is an abstract category that estimates the cost of all instruction-fetch related stalls in out-of-order cores. It is calculated as \( \frac{IDQ\_UOPS\_NOT\_DELIVERED\_CORE}{(4 \times CPU\_CLK\_UNHALTED\_THREAD)} \) [14] for recent Intel cores where 4 is the machine width.

Performance metric and generic metric are metric classes that quantify performance or generic info of the execution (e.g., IPC and UPI, respectively).

All metrics are formatted in a blue Consolas font. Bottlenecks are also made italic. For example, IPC is a metric, defined as \( \frac{INST\_RETIRED\_ANY}{CPU\_CLK\_UNHALTED\_THREAD} \), and Frontend_Bound is a bottleneck.

2.5 Top-Down Performance Analysis

This article leverages the Top-down Microarchitecture Analysis (TMA) method [8]. TMA simplifies cycle-accounting—the process of identifying costs of performance bottlenecks, also called CPI breakdown—for out-of-order cores using microarchitecture-abstracted metrics organized in one simple hierarchy [16].

At the top level, TMA employs a single point of division where issue-pipeline slots are divided into four main categories: Frontend Bound, Backend Bound, Bad Speculation, and Retiring. The latter two denote non-stalled slots while the former two denote stalls. A simple decision tree is used: if a slot is utilized by some operation, it would be classified as Retiring or Bad Speculation, depending on whether it eventually gets committed (retired). Unutilized slots are classified as Backend Bound.
Table 1. Implementation of TMA Top-Level Bottlenecks on Intel Core [8]

| Metric Name          | Implementation (Intel Core\textsuperscript{TM} events’ formula) |
|----------------------|---------------------------------------------------------------|
| 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                             |
| Backend Bound        | 1 - (Frontend Bound + Bad Speculation + Retiring)             |

if the back-end portion of the pipeline is unable to accept more operations, or otherwise as *Frontend Bound* when no operations are delivered.

The method further divides *Backend Bound* into *Memory Bound* or *Core Bound* in the second level, depending on whether Backend-stalls [8] are due to memory or non-memory operations, respectively. In this work, the stalls of an abstracted core, including but limited to the ones described in Section 2.2, are represented in a group of

\[
\text{Abstracted}_B\text{ottlenecks} = \{\text{Frontend Bound, Bad Speculation, Memory Bound, Core Bound, Retiring}\}
\]

The performance counter formulas for the TMA top-level bottlenecks are summarized in Table 1. Per Intel processor generation formulas of these as well as other metrics are posted in [14].

3 ANALYSIS METHODOLOGY

This section describes the analysis method. It covers the goal of the method, the MGM algorithm, why it was built on top of TMA, and what are the MGM’s new extensions.

Given a set of *workloads* executed under two setups (configurations), the goal of MGM is to identify the architectural features that have the most impact on the performance difference between these two setups. Specifically, the *workloads* can be a set of applications from a specific domain—e.g., FP benchmarks, hotspots from application(s)—e.g., time-consuming loops or functions, and so forth. The two setups can be two processors, two compilers, same compiler when examining an optimization flag, two revisions of an application (e.g., scalar vs. vector implementations).

MGM exploits *differential analysis* [9, 17] at the core of the method. The basic idea of *differential analysis* is to profile a program twice with two setups, whereas the difference of these profiles guides the analysis. However, it is not straightforward to apply differential analysis to achieve MGM’s goal because there is no explicit connection between arch feature and performance metrics. A key contribution of MGM is that it carefully constructs what metrics are fed into differential analysis in an iterative process.

Figure 4 depicts the discovery process in MGM. The steps are grouped and colored into four parts to match the abstraction in Figure 1:

1. **Collect Metrics**: MGM starts with a set (B) of performance metrics (*Abstracted Bottlenecks* and an evaluation-specific performance metric) in step 1. For example, *IPC* is a good evaluation metric for an iso-frequency uarch-only evaluation. Next, at 2 MGM collects the metrics on both setups under controlled settings (e.g., fixed iso-frequency).
Legend for symbols in Figure 4:

\( B \) = the current set of metrics monitored by MGM.
\( w \) = a particular workload analyzed in one iteration.
\( \beta, \gamma \) = particular metrics (bottlenecks) from \( B \).
\( \chi \) = a stall (or feature) that explains a drift in some \( \beta \).

Fig. 4. MGM’s algorithm for identifying arch features.
(2) **Select a Workload:** At (1) MGM utilizes differential analysis of the metrics in \( B \) in order to select the best workload (at (3)) for analysis in each iteration. This part increases uarch coverage and analysis efficiency.

(3) **Find an Arch Feature:** At stage (3) a hotspot of the selected workload is examined—via ad-hoc or expert analysis—for a potential feature that can explain the drift in the triggering metric. Next, at (4), if a feature was found, the *Design a Kernel* (7) sub-process is used to handcraft a kernel that isolates that feature. Otherwise, MGM resumes analysis with (possibly other) workload per step (3b).

(4) **Define Metrics:** In (4), a *specific metric* capturing the feature is used to mark all impacted workloads. In addition, a *generic metric* can be considered in this step; in such case, it is appended to \( B \) for monitoring in future iterations.

The algorithm terminates when sufficient coverage is reached in terms of explaining the performance drifts for the workloads of interest. For example, at least one feature was associated with every workload.

### 3.1 Leveraging TMA

The employment of TMA’s *Abstracted Bottlenecks*—high-level categories of stalls—in MGM is a strategic choice. We emphasize that these metrics are instrumental since they (I) enable MGM to evaluate different setups including non-consecutive processor generations, as we demonstrate in Section 4, (II) help to cover all parts of the processor microarchitecture via steering the analysis to workloads with a major change in one uarch domain, and (III) ensure an efficient search process in terms of duration; an otherwise bottom-up approach that investigates every changed feature—e.g., test every ISA instruction in a brute-force—would be inefficient.

On the other hand, such abstraction may blur the characteristics of an individual architectural feature; especially when several features are introduced in one hardware generation. MGM compensates for this shortcoming via unique extensions described in the next subsection.

Note that an alternative choice that would use raw *performance counters* is not efficient. For example, the *Frontend Bound* bottleneck of TMA is a better way to capture the overall penalty of the front-end pipeline than a set of *performance counters* such as `ICACHE_MISSES`; whereas, these *counters* may not apply for every processor architecture (e.g, Intel vs. ARM) or even at different processor generations of the same vendor. Moreover, some counters may count different events across different architectures (e.g., the same icache-miss counter may count different kinds of cache misses on different processors).

### 3.2 Extensions Over TMA

The key extensions over Top-Down analysis (TMA) are as follows:

(I) **Safety-net using a performance metric** in the *initial* set of metrics, \( B \), to be monitored by the algorithm in step (1). This evaluation-specific performance metric serves as a goal-keeper should there be a change that impacts performance and yet not manifest in the chosen *Abstracted Bottlenecks*, e.g., 456.hmmer shows 27% improvement in IPC (Section 5.2), while all bottlenecks show negligible drifts. The symptoms of a particular change may get blurred in out-of-order cores; we call this “blurring effect.”

(II) **Dynamic adaptation through generic metrics**—As MGM’s iterative analysis proceeds, new generic metrics can be added to \( B \) should they apply in step (1c). For example, the CMOV finding facilitated the addition of `UPI` generic metric, which later assisted in detection of more findings with similar symptoms: ST-2src and DIV256 in Section 5.2. Similarly, `IpFLOP` assisted in the detection of “Extra vectorization” and...
“CISC-ified memory accesses” in Section 5.3. Sample performance and generic metrics are defined in Section 4.3.

(III) Using microbenchmarks to verify an insight. The design of a feature-specific kernel in step ② isolates the effects of a feature which helps in the architectural understanding—like answering “under what conditions does the feature is beneficial?” More importantly, a bottom-up step complements the Abstracted Bottlenecks because it enables the creation of generic metrics. Beyond that, this sub-process has key advantages from an analysis methodology’s perspective:

(a) Efficiency: A feature-specific metric—e.g., CMOV% of total instructions—enables the marking of all impacted workloads such that a repeated analysis of the same issue is avoided.

(b) Coverage: A bottom-up ruling out process enables MGM to find multiple issues using the same metric. For example, debug of 5% reduction in Core Bound has led to the discovery of both the CMOV and “Improved FP divide” features in Section 5.2.

(c) Identifying crosstalk (green-filled dotted components of the flowchart in Figure 4): A single architectural change can impact multiple bottlenecks. Analyzing TMA results of a kernel magnifies such crosstalk phenomenon. For example, decoding the CMOV instruction into a fewer number of uops not only shifts Core Bound stalls into Retiring (obvious), it also reduces Frontend Bound (less obvious) as illustrated by the CMOVE-x1 kernel in Figure 5(a) and Figure 6(a). This advantage can also be viewed as another efficiency that MGM employs in order to shorten the analysis duration.

The microbenchmarks developed by this work are described in Section 4.2.

4 EXPERIMENTAL EVALUATIONS
We demonstrate the proposed method through two experiments using SPEC CPU2006 INT and FP benchmarks on Intel processors at iso-frequency.

(1) Microarchitecture evaluation—The setups are Skylake (SKL) and IvyBridge (IVB) processors. That is a uarch-only comparison where the ISA is fixed (same AVX binaries).

(2) ISA evaluation—The setups are AVX vs. AVX2 [18]. That is an ISA-only comparison when the uarch is fixed (SKL).

This section includes the details of these experiments: the setup, the kernels we developed, and the metrics adopted by MGM. The use of kernels enables us to verify particular hypotheses and observations, while the use of benchmarks helps to assess implications on full applications. Although our demonstration is conducted on Intel processors, MGM is generic and can be applied to other architectures.

4.1 Experimental Setup
The system setup details are listed in Table 2. We used pmu-tools/toplev [19] to obtain the full TMA profile and Pin/SDE [20, 21] for ISA information.

Controlled setting: In order to expose effects of the architectural features, we perform all experiments in a controlled setting. DVFS and other power management features [6] are disabled, enabling an iso-frequency comparison and thus focus analysis on u/architectural features. The profiled application is affinitized to a one logical processor in a physical core, in order to avoid SMT interference. Also, all nonessential daemons of the operating system, like NMI watchdog, are disabled. All metrics are collected when filtered to the measure workload and with no counter-multiplexing [22].
Fig. 5. Abstracted_Bottlenecks and performance metric (IPC) for Skylake. MGM employs differential analysis on this initial set of metrics in order to select the best workload at each analysis iteration.
Fig. 6. Abstracted_Bottlenecks and performance metric (IPC) for IvyBridge. MGM employs differential analysis on this initial set of metrics in order to select the best workload at each analysis iteration.
Table 2. Systems’ Setup Parameters

| Processors | Intel Core™ i7-6700K (Skylake) and Intel Core™ i7-3770 (IvyBridge). Both processors have per-core:  
|            | • Separate L1 instruction cache (L1I) and L1 data-cache (L1D) of 32 KB each.  
|            | • Common L2-cache (for instructions and data) of 256 KB.  
|            | • Decoded uop cache (DSB) of 1500 uops.  
|            | • 4-wide issue-pipeline.  
|            | An 8 MB L3 cache shared among cores in the same socket and are configured to 3 GHz fixed frequency. |
| Memory     | 8 GB DDR4 @ 2133 MHz |
| OS         | Ubuntu 14.04: Linux kernel 3.19  
|            | perf version 3.19.8-ckt5 |
| Compiler and Benchmarks | SPEC CPU 2006 v1.2 (base/rate mode)  
|            | Intel Compiler 14 (three sets of binaries):  
|            | INT benchmarks target SSE ISA.  
|            | FP benchmarks target two ISAs: AVX (-xAVX) and AVX2 (-xCORE-AVX2).  
|            | 400.perlbench in Figure 2 uses a different AVX binary with common ISA for AMD and Intel. Iso-frequency was used in that case as well. |

4.2 Kernels for Arch Features

This section presents the kernels that were developed through the use of MGM in the two experiments. Pseudo code for sample kernels is provided in Listing 1. The kernel names in brackets match Table 3.

```
| fp-add-lat | fp-add-bw |
|------------|-----------|
| Loop:      | Loop:     |
| YMM0 = YMM15 + 1.23; | YMM0 = YMM0 + 0.11; |
| YMM1 = YMM0 + 1.23; | YMM1 = YMM1 + 1.11; |
| YMM2 = YMM1 + 1.23; | YMM2 = YMM2 + 2.11; |
| ...        | ...       |
| YMM15 = YMM14 + 1.33; | YMM15 = YMM15 + 15.11; |
| jcc Loop   | jcc Loop  |

CMOVA:  
X := Y > 0 ? a : b; // uses CMOVA/CMOVNBE inst.  
Update loop index & jump to CMOVA

CMOVE:  
X := Y == 0 ? a : b; // uses CMOVE/CMOVZ inst.  
Update loop index & jump to CMOVE
```

Listing 1. Pseudo Code Sample Kernels We Developed

**Code size** [COD$^{*}$-$u$]: a simple loop of move register-to-register operation with loop unrolling with factors, $u$, of 1, 2, 4...2,048. Avoiding memory accesses or compute operations lets this kernel focus on the front-end aspects. The purpose of these kernels is to measure **Frontend Bound** when the code is fetched from Loop Cache, Uop Cache, L1 instruction cache (L1I), and the L2 cache.

The COD-long-$u$ flavor uses instructions that require many bytes in their representation. For example, an instruction that moves a big immediate (constant) to a register.
Table 3. Performance Metrics and Generic Metrics Report for Kernels

| Group   | Kernel name | SKL      | Time [s] | IPC  | UPI   | Loop% | FLOP C | IVB | Time [s] | IPC  | UPI  | Loop% | FLOP C | Speed up (SKL/IVB) | Note (Fetch-Unit) |
|---------|-------------|----------|----------|------|-------|-------|--------|-----|----------|------|------|-------|--------|-----------------------|------------------|
| Frontend| COD-u1      | 274.7    | 1.38     | 0.91 | 0.98  | 0.98  | 313.6  | 1.21 | 0.91     | 0.98 | 0.91 | 0.96  | 0.96  | 1.14x                 | Loop Cache       |
|         | COD-u16     | 274.7    | 1.03     | 0.99 | 0.00  | 0.00  | 280.0  | 1.01 | 0.99     | 0.99 | 0.00 | 0.00  | 0.00  | 1.02x                 | Loop Cache       |
|         | COD-u128    | 274.8    | 1.00     | 1.00 | 0.00  | 0.00  | 278.1  | 0.99 | 1.01     | 1.01 | 0.00 | 0.00  | 0.00  | 1.01x                 | L1 Hit           |
|         | COD-u1024   | 274.8    | 1.00     | 1.00 | 0.00  | 0.00  | 281.9  | 0.98 | 1.01     | 1.01 | 0.00 | 0.00  | 0.00  | 1.03x                 | L1 Hit           |
|         | COD-long-u10k| 50.1     | 1.60     | 1.00 | 0.99  | 0.99  | 96.9   | 0.70 | 1.00     | 1.00 | 0.99 | 0.99  | 0.99  | 1.93x                 | L1 Hit           |
|         | COD-long-u100k | 59.3    | 1.34     | 1.00 | 0.99  | 0.99  | 194.1  | 0.35 | 1.01     | 1.01 | 0.99 | 0.99  | 0.99  | 3.27x                 | L2 Miss          |
| Control Flow | CMOSA      | 49.0     | 2.23     | 1.25 | 0.99  | 0.99  | 50.7   | 1.73 | 1.29     | 1.29 | 0.99 | 0.99  | 0.99  | 1.03x                 |                  |
|          | CMOSA-x30   | 62.2     | 1.14     | 0.97 | 0.98  | 0.98  | 65.1   | 1.98 | 1.00     | 1.00 | 0.92 | 0.92  | 0.92  | 1.52x                 |                  |
| Memory   | LOAD-128b   | 201.8    | 1.19     | 0.95 | 0.95  | 0.95  | 226.0  | 1.05 | 0.95     | 0.95 | 0.95 | 0.95  | 0.95  | 1.12x                 |                  |
|          | LOAD-256b   | 51.1     | 1.19     | 0.95 | 0.95  | 0.95  | 146.2  | 0.41 | 0.95     | 0.95 | 0.95 | 0.95  | 0.95  | 2.85x                 |                  |
|          | STORE-256b  | 94.2     | 1.61     | 0.80 | 0.80  | 0.80  | 129.7  | 1.16 | 0.80     | 0.80 | 0.80 | 0.80  | 0.80  | 1.38x                 |                  |
|          | STORE-128b  | 55.2     | 2.75     | 0.80 | 0.80  | 0.80  | 65.1   | 2.32 | 0.80     | 0.80 | 0.80 | 0.80  | 0.80  | 1.18x                 |                  |
|          | STORE-128b-2src | 60.2   | 2.02     | 0.75 | 0.75  | 0.75  | 65.3   | 1.85 | 1.00     | 1.00 | 0.75 | 0.75  | 0.75  | 1.08x                 |                  |
| Compute  | FP-add-lat  | 301.8    | 0.31     | 0.93 | 2.0   | 2.0   | 228.8  | 0.42 | 0.96     | 0.96 | 2.0  | 2.0   | 2.0   | 2.6x                  | SKL only          |
|          | FP-add-bw   | 44.1     | 2.42     | 0.94 | 16.0  | 16.0  | 90.1   | 1.19 | 0.95     | 0.95 | 7.8   | 7.8   | 7.8   | 2.04x                 |                  |
|          | ADD-scalar-u16 | 274.7  | 2.02     | 1.00 | 1.00  | 1.00  | 281.7  | 1.97 | 1.00     | 1.00 | 1.00 | 1.00  | 1.00  | 1.03x                 |                  |
|          | ADD-vec128-u16 | 34.8  | 4.01     | 1.00 | 7.9   | 7.9   | 70.2   | 1.98 | 1.00     | 1.00 | 3.9   | 3.9   | 3.9   | 2.02x                 |                  |
|          | ADD-vec256-u16 | 175.3  | 3.99     | 1.00 | 15.7  | 15.7  | 35.1   | 1.99 | 1.00     | 1.00 | 7.8   | 7.8   | 7.8   | 2.00x                 |                  |
|          | NMA-vec128-u16 | 68.7  | 3.03     | 1.00 | 8.0   | 8.0   | 34.7   | 2.98 | 1.00     | 1.00 | 15.7  | 15.7  | 15.7  | 0.51x                 |                  |
|          | NMA-vec256-u16 | 34.3  | 3.04     | 1.00 | 16.0  | 16.0  | 34.7   | 2.98 | 1.00     | 1.00 | 15.7  | 15.7  | 15.7  | 1.01x                 |                  |
|          | FMA-vec128-u16 | 34.8  | 4.00     | 1.00 | 15.8  | 15.8  | 1.97x  |                  | SKL NMA          |
|          | FMA-vec256-u16 | 17.5  | 3.99     | 1.00 | 31.3  | 31.3  | 1.96x  |                  |                  |

Conditional moves [CMOV\*] select a value to be assigned to a register based on status flags (e.g., whether the zero flag was set). CMOVs are often used to eliminate data-dependent conditional branches. IVB implements these using two uops. SKL reduces this down to one-uop for most but not all flavors of CMOVs (CMOVBE, CMOVNBE, CMOVA, and CMOVNA have inferior latency and throughput compared to other CMOVcc flavors) [11]. Two kernel flavors were developed: CMOVE (1 uop) and CMOVA (2-uop). Each kernel contains 4 or 30 CMOVs in the loop body.

Memory Accesses [LO\*, ST\*]: IVB implementation of AVX 256-bit memory accesses is done through two chunks of 128-bits each [11]. To evaluate this aspect, we sequentially read/write an array of the same size using 128- and 256-bit loads/stores. In addition, we examine if the actual memory addressing encoding affects performance by including 1src/2src flavors for the store kernels.

Divide: SKL improves the 256-bit divides by implementing the operation in one-uop rather than two-uops in IVB. The kernel uses div-64/div-128/div-256 for divide operations at 64- (scalar), 128- (XMM), and 256-bit (YMM) widths.

FP-add/mul [*-lat, *-bw]: The FP ADD/SUB instruction latency/throughput was changed from 3/1 in IVB to 4/2 in SKL [11] (i.e., throughput doubled while latency increased), while FP MUL was improved from 5/1 to 4/2, respectively.

Vectorization [ADD-\*]: We use a simple ADD-MOV kernel on scalar, SSE (128 bits) and AVX (256 bits) ISA [23].

Multiply-Add [FMA-\*, NMA-\*]: Two types of kernels were developed each with a simple loop that does single-precision Multiply-Add between registers (one mov, two operations, no memory
accesses). The first kernel does the Multiply-Add using two distinct instructions—we call it NMA (Non-fused Multiply-Add), while the second one uses the FMA instruction [23]. Both kernels do the same amount of work. 128- and 256-bit versions are included.

4.3 Sample Metrics
The performance and generic metrics that were used throughout the two experiments:

- **IPC**—Instructions Per Cycle. Useful performance metric for comparing two microarchitectures at iso-frequency.
- **FLOPC**—Arithmetic Floating-point Operations retired per Cycle. A good performance metric for comparing two ISAs of FP applications [24].
- **UPI**—Uops Per Instruction is the average number of uops to which an ISA instruction is translated to. Performance-wise, this metric serves as an indicator for how efficient the system’s code generation and instruction selection were for the given machine [12].
- **IpFLOP**—Instructions Per arithmetic Floating-point Operation is the average number of instructions used to complete a single FP-arithmetic operation. This metric serves as an indicator for the effectiveness of the (vectorizing) compiler and the ISA and whether unnecessary instructions waste performance [25].

5 RESULTS AND ANALYSIS
In this section, we present experimental results of the evaluations described in the previous section. We show the results of our new kernels, then we analyze the results of the two SPEC experiments.

5.1 Kernels
The kernels described in the previous section are grouped by their focus domain in Table 3: Frontend (including control-flow), Memory, and Compute (including AVX2). The metrics that show significant changes are highlighted. The Memory kernels generally have better performance (higher IPC). Specifically, the CMOVE and STORE-128b-2src kernels show a reduced UPI in SKL. The Compute vector kernels show \( \sim 2 \times \) FLOPC as the number of FP execution units got doubled in SKL.

Key observation from Table 3 (font color matches key cells to focus on in the table):

- **Fetch stalls reduction** when code misses the L1I. The COD-long-u10k kernel sped up by \( 2 \times \) SKL over IVB. The Haswell uarch has an improved front-end with earlier iTLB and i-cache lookups [26]. Besides, SKL supports a more aggressive instruction prefetch [6].
- **Better L1D performance** due to the native 256-bit L1 Data-cache (L1D) bus. Loading an array with 256-bit chunks is \(+2 \times\) faster than 128-bit in SKL, unlike IVB. 256-bit stores show similar improvement.
- **Efficient store two-source addressing**. The store-128b-2src kernel shows significant reduction in UPI.
- **FP-add-lat** (latency) kernels is slower in SKL over IVB, while FP-add-bw (bandwidth) has \( 2 \times \) speedup.
- **FMA significantly improves FP performance**; it achieves \( \sim 2 \times \) better FLOPC over NMA in SKL.

Figure 5 and Figure 6 show the performance profile for SKL and IVB uarchs, respectively. The lines are IPC plotted on the right Y-axis, while the bars are the TMA [8] Abstracted_Bottlenecks per
the representation described in Section 2.4. Kernels are grouped per their domains. SPEC benchmarks are sorted by the IPC difference between SKL and IVB (which is correlated with the measured speedup due to the above-mentioned controlled setting).

5.2 Microarchitecture Features

**Improved i-cache miss handling:** In the first iteration, MGM points to *Frontend Bound* in step ➀ since it was the most reduced bottleneck for INT benchmarks with −8% on average. 400.perlbench, at the far right of Figure 5(b) and Figure 6(b), is picked for analysis by step ➀a since it has the biggest decrease in *Frontend Bound* from 38% in IVB down to 15% in SKL. The analysis in step ➀ has root-caused this to an improvement in i-cache miss handling, using the full TMA hierarchy through VTune [27]. Intel states that “state-of-the-art advances in branch prediction algorithms enable accurate fetch requests to run ahead of uop supply to hide instruction TLB and cachemisses” [26]. Generally, SPEC benchmarks do not have large code footprints and thus are not iTLB sensitive. Furthermore, this finding corroborates with Jaleel [28] who reported that 400.perlbench, 483.xalancbmk, 458.sjeng, 445.gobmk, 403.gcc, and 453 povray have a code footprint bigger than 32 KB. Our results show a sizable reduction in *Frontend Bound* (and *Frontend Latency* [8]) for these benchmarks. In step ❼, the COD-* kernels were designed.

Attribution—In step ➀, Benchmarks whose code footprint exceeds the L1I’s size and exhibit sizable (+5%) reduction in *Frontend Latency* were marked benefiting from “Improved i-cache misses.” No generic metric or crosstalk was identified by this particular MGM iteration.

**UPI reduction—CMOV, ST-2src, DIV256:** In a next iteration of MGM, we analyzed 456.hmmer, which had he biggest IPC speedup and yet was not marked by the previous feature. Thus, step ❼b was followed in this iteration since IPC has increased from 2.03 in IVB up to 2.53 in SKL for this benchmark, while none of the *Abstracted_Bottlenecks* showed a noticeable shift. This magnifies the *blurring effect* likely due to multiple features introduced in same uarch generation, hence the importance of including IPC as a performance metric.

A close analysis, in step ❼, showed significant reduction in *Core Bound* for the hottest loop, which had two CMOVL instructions. Intel documents that the latency of CMOV* got reduced by one cycle starting the Broadwell uarch [11]. We also noticed fewer numbers of the *UOPS RETIRED RETIRE_SLOTS* performance counter in the VTune profiles. Throughout the development of a designated kernel in step ➋, it was clear that (most but not all flavors of) CMOVs are now decoded into a single-uop. CMOVs represent 23% of dynamic instructions in this benchmark. This can explain the UPI reduction from 1.25 to 1.04 for this benchmark, per Figure 7.

Note that besides marking the impacted workloads in step ❼, UPI was identified as a *generic metric* and thus was appended to the group B to be monitored in future iterations. This has helped us to not miss two other findings (ST-2src, DIV256) with the same symptom of UPI reduction but was instead caused by other instruction types.

Attribution—Benchmarks who exhibit sizable reduction in the UPI metric in SKL over IVB and the number of CMOV instructions was 4% or more of the benchmark’s total instructions count (see Figure 7) were marked as benefiting from this feature. We similarly attributed benchmarks benefiting from Store two-source addresses.

**Removal of memory double-pumping:** 436.cactusADM has the biggest reduction in *Memory Bound* when comparing FP benchmarks in Figure 5(c) vs. Figure 6(c). Examining the TMA’s profile suggested that the fraction of cycles the core was waiting for the L1D as well as the utilization of memory execution ports were reduced in SKL. This corroborates with the improved handling of 256-bit memory accesses [26].

ACM Transactions on Architecture and Code Optimization, Vol. 16, No. 4, Article 46. Publication date: December 2019.
Fig. 7. Attribution of Uops-per-Instruction (UPI) improvements of Skylake over Ivybridge.

Attribution—A similar performance signature is observed for 435.gromacs, 465.tonto, and 481.wrf. They have Backend_Bound.Memory_Bound.L1_Bound as their critical bottleneck and high utilization of the memory execution ports.

FP-arithmetic has mixed impact: 435.gromacs had a significant increase in IPC with moderate drifts in Abstracted_Bottlenecks. The top loop showed high use of FP operations with MUL/ADD accounting for 33% of the dynamic instructions.

Thus, the double FP BW increased performance (the ADD latency was not exposed in this benchmark). On the other hand, 410.bwaves showed a reduction in Retiring with almost no change in UPI. The top loop had seven FP-ADD chained instructions; thus, the longer latency decreased performance. The inclusion of IPC as a performance metric was instrumental for the analysis of both benchmarks.

Attribution—We examined the hottest basic-blocks reported by Pin output reports. Whenever a data-dependency of sequential FP-ADD/SUB operations is detected, “FP-ADD 4-cycle latency” was marked as degrading performance. Otherwise, if we noticed multiple such instructions are data-independent (hence, can execute simultaneously in the same cycle), “FP-ADD Throughput” was marked as contributing to performance.

Improved FP divide (DIV256): 434.zeusmp and 437.leslie3d got nice speedups of 17% and 10% partially due to the improved divide latencies [11]. This explains the reduction in Core Bound and further helps the UPI reduction for both benchmarks (in addition to ST-2src), as Figure 7 shows.

The last paragraph of each feature described in this subsection lists how we identified which benchmarks are most likely to be affected by it. An aggregation of this data is provided in Table 4.

5.3 Architecture Features

Experiment (II) demonstrates MGM for analyzing the ISA. Specifically, we analyzed FP benchmarks compiled for AVX2 compared to AVX on same hardware (SKL). The AVX2 ISA extension
provides FMA instructions which double the peak FP throughput and gather/scatter instructions for vectorizing non-adjacent memory accesses [26].

Obviously, IPC cannot be used as a performance metric since two different ISAs are being evaluated. Instead, FLOPC was accompanied to B in step 1 of Figure 4. The number of FP operations is an inherent attribute of the application. The average speedup is 6% where the average FLOPC metric increased from 0.99 with AVX to 1.09 with AVX2.

Table 4. Mapping Features Impact on SPEC CPU2006 Benchmarks

| Benchmark     | warp speedup | AVX IpFLOP on SKL | Improved i-cache misses | UPI reduction: CMOV | UPI reduction: ST2arc | No double-pumping | Improved Divide 256-bit | FP-ADD 4-cycle latency | FP-ADD Throughput |
|---------------|--------------|--------------------|--------------------------|---------------------|-----------------------|------------------|--------------------------|------------------------|---------------------|
| 410.bwaves   | 0.98x        | 1.95               |                          | +                   | -                     | -                |                          | -                      | -                   |
| 433.milc     | 1.01x        | 3.62               |                          | +                   | -                     | -                |                          | -                      | -                   |
| 470.lbm      | 1.09x        | 1.73               |                          | +                   | -                     | -                |                          | -                      | -                   |
| 437.leslie3d | 1.10x        | 0.78               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 459.GemsFD1D | 1.11x        | 0.91               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 454.calculix | 1.12x        | 0.62               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 447.dealII   | 1.12x        | 1.06               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 481.wrf      | 1.14x        | 2.13               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 482.sphinx3  | 1.14x        | 7.35               | -                        | +                   | +                     | +                |                          | +                      | +                   |
| 444.namd     | 1.14x        | 9.22               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 450.soplex   | 1.15x        | 4.74               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 434.zeusmp   | 1.17x        | 1.21               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 416.gameess  | 1.20x        | 1.91               |                          | +                   | -                     | -                |                          | -                      | -                   |
| 435.gromacs  | 1.22x        | 2.72               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 465.tonto    | 1.24x        | 0.75               | +                        | +                   | +                     | +                |                          | +                      | +                   |
| 453.povray   | 1.25x        | 0.72               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 436.cactusADM| 1.31x        | 1.88               | +                        | +                   | -                     | -                |                          | -                      | -                   |
| 471.omnetpp  | 1.05x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 401.bzip2    | 1.07x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 429.mcf      | 1.08x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 403.gcc      | 1.10x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 445.gobmk    | 1.11x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 473.astar    | 1.13x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 462.libquantum| 1.14x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 464.h264ref  | 1.15x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 458.sjeng    | 1.19x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 483.xalancbmk| 1.21x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 456.hmmer    | 1.25x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
| 400.perlbench| 1.35x        |                    |                          | +                   | -                     | -                |                          | -                      | -                   |
Fused Multiply-Add (FMA): We analyzed 433.milc, which had biggest increase in FLOPC. Most loops of this benchmark were not vectorized. However, the compiler has exploited FMA for scalar code. Overall, FMAs accounted for over 22% of the AVX2 dynamic instructions—a significant reduction in instruction count. The corresponding feature-specific metric as well as IpFLOP were added in step \(\circ\) of the algorithm. IpFLOP is a generic metric to capture all cases where instruction count reduction helps FP performance.

Figure 8 presents a summary of the issues that were identified by experiment (II). The line is the AVX2 improvement for IpFLOP (i.e., \(\text{IpFLOP}_{\text{AVX}} - \text{IpFLOP}_{\text{AVX2}}\)) and is plotted on the right Y-axis. The bars are the fraction of FP operations that correspond to the issue. For example, IpFLOP was reduced by 0.24 for 433.milc while FMA (all flavors) represents 41% of the FP operations.

Extra vectorization: It was clear that 410.bwaves should be analyzed next, once IpFLOP was included in the differential analysis. The main loop of this benchmark used 128-bit vector instructions in the AVX binary while it exploited 256-bit vectors in AVX2. It turns out there was a bug in the specific compiler version that prevented the use of 256-bit vectors in the AVX binary. The “Extra Vectorization” metric in Figure 8 measures the fraction of FP operations that either become 256-bit vectorized in the AVX2 binary (i.e., were 128-bit vectorized in AVX) or got vectorized in AVX2 (i.e., were scalar in AVX).

“CISC-ified” memory accesses (CISC mem+op): Next, we analyzed 459.GemsFDTD, which had a significant reduction in IpFLOP and yet the two aforementioned issues did not account for all that reduction. Examining the hotspot revealed that the compiler not only used FMA but also eliminated instructions dedicated for memory loads (VMOVSD). These loads were lumped into compute instructions in AVX2, where a CISC instruction did the load and the compute. It turns out that the look-ahead for pairs of ADD-MUL to fuse as FMA during the compiler’s code-generation pass, has created this side effect. Such look-ahead was not performed in the AVX case.

482.sphinx3 is the worst outlier. It has FLOPC slowdown of 8%. Figure 8 suggests the IpFLOP metric was increased in AVX2 (which has 12% extra instructions!). We observe FMAs account for
<1% of net instructions, while FP-ADD was 9%. We believe the slowdown is due to the FP-ADD being on the critical path in the AVX2 binary.

6 RELATED WORK

Many tools including VTune [27], HPCToolKit [29], and Gooda [30] use performance counters to build a microarchitectural profile, primarily in order to facilitate software optimizations (tuning). They offer a comparison of two profiles as well. [17] and [31] employ differential analysis to identify scalability bottlenecks. Our work employs differential analysis in order to reverse engineer the architectural features leading to generational performance drifts.

A thorough ad-hoc analysis that attributes energy costs to uarch/SIMD features was performed by [32]. They conclude process technology is the dominant contributor to energy efficiency and outlined some uarch features that contribute as well. Compared to our work, [32] has spanned more microarchitectures but was restricted to kernels and has excluded the memory subsystem. Our work analyzed real workloads, and did a finer grain analysis (including evaluation of non-compute aspects) using a structured analysis method.

[33] propose to characterize workloads independent of the ISA. They profile memory, branch, and opcode behavior of an application using the intermediate representation of an application instead of the binary. Following up on this work, they propose using the ISA-independent profiles for speeding up the design of accelerators [34]. They achieve good speedups with minimal loss in accuracy for both performance and power estimations. In addition, [35] demonstrated an analytical performance and power model, based on micro-architecture-independent application profiles to enable the evaluation of large design spaces using a single application-specific but architecture-independent profile.

Thread Clustering [36] leverages platform-specific counters in order to speed up multithreaded workloads via mitigation of shared cache effects. Similarly, scheduling techniques for systems with shared resources [37, 3] or profiling techniques for producing instruction mixes [38, 39] rely on specific performance counters not available in every platform. The survey in [37] reports that many published approaches suffer from this drawback.

MGM advocates for generic metrics to achieve its goal, instead of specific performance counters. For example, IPC or IPfLOP are performance metrics that broadly apply to a wide range of platforms (albeit of x86 ISA). In addition, the Abstracted Bottlenecks are designed to accommodate multiple microarchitectures as shown by [16]. CPI^2 [40] uses CPI (cycles per instruction) history in order to improve utilization of datacenters that collocate jobs on shared machines. DOEE [13] uses energy and timestamp counters in order to improve energy efficiency. Similar to MGM, both of these works rely on generic metrics, not specific counters, to achieve their goals.

Architects use simulators to explore the impact of potential architectural features, such as vector [18] or heterogeneous ISA [41]. In addition, in order to understand the impact of microarchitectural features, simulators [42, 43] are used to measure the hardware related events and analyze the microarchitecture bottlenecks by examining their CPI-stacks. A CPI stack breaks down the execution time of an application into microarchitectural activities, showing the relative contribution of each activity. However, these simulators use mechanistic models [42, 43] that require deep microarchitectural knowledge for their construction. Thus, it is difficult to accurately simulate every feature of a modern complex processor, such as Intel Skylake and Ivy Bridge, especially with the lack of public microarchitecture documentation of these commercial processors.

Most recently, [44] indicate modern processors employ complex "non-disclosed strategy for execution," which motivates them to use machine learning in order to find architectural attributes to predict the performance of basic-block reordering.
7 DISCUSSION, EXTENSIONS, AND MODELS’ COMPARISON

Generality: While our evaluation has demonstrated the method on Intel processors, MGM is generic enough to be applied to other processors. The Abstracted_Bottlenecks are not specific to a particular uarch or ISA [16]. So are the employed performance and generic metrics such as IPC and IpFLOP, respectively. In fact, the first experiment successfully analyzed non-consecutive processor generations.

Limitations: While the method can identify what features impact performance, including whether a feature had a positive or negative impact, it cannot tell by how much. This is an inherent limitation of the real-system setup—the framework of choice by this article.

For uarch evaluations, the two processors have to be of the same machine width (superscalar size at the front-end and back-end crossover point) due to the formation of the Abstracted_Bottlenecks—fractions of slots. This can be addressed by incorporating absolute counters like slots. In addition, the method cannot identify the specific uarch generation where a change was introduced should two non-consecutive generations be analyzed. While we compensated for this by referring to the documentation, generally it can be addressed by evaluating the mezzanine processor itself.

Some (power-related) performance features, like Turbo [45], may not suit our particular experimental setup, since it assumed an iso-frequency. We explain how to address some of them next.

Extensions: In order to cover features that involve variable frequency, MGM may be extended with suitable metrics. For example, Instructions per Second may be used instead of IPC and the Abstracted_Bottlenecks may be normalized to a fixed-frequency based clock counter (e.g., the CPU_CLK_UNHALTED.REF_TSC or ref-cycles performance counter per Intel or Linux perf [22] naming conventions). The processors under evaluation must be on the same technology node in order to filter out the impact of the process technology.

The method can cover microarchitectures of other vendors. We hope that cross-vendor comparison can help the designers of one system to design the right improvements based on learnings from the other system.

Another possible extension of this work is to identify features impacting energy beyond just performance. [24] abstracts the power consumption of a processor core into few high-level domains while [46] isolates the contribution of the process technology to energy efficiency. Both of these works may be leveraged for such an extension.

Multithreaded workloads: MGM successfully identified architectural changes that were most influential on performance over two evaluations that use single-threaded workloads—the scope of this work. The method can be extended to multithreaded workloads. GFLOPs (Giga FP operations per second) is a potential performance metric for one evaluation that studies multi-core FP applications. The reason is that it can handle situations where operating parameters dynamically change, as is the case with Skylake-server that drops the frequency based on how many cores are utilized [25]. Understanding the architecture impact on multi-core scalability is a second interesting evaluation, where Kernel_Time may be a generic metric to gauge the system-induced non-scaling time [25].

With that said, handling hyper-threading (multiple threads share the same physical core) needs extra attention due to a higher degree of interference on shared resources (e.g., execution units). Multithreading requires building additional kernels and metrics (e.g., per-core IPC vs. per-thread IPC) that isolate these cases and quantify their performance impact.

The above-mentioned aspects and some additional parameters (we discuss after the chart) are quantitatively summarized in the spider chart in Figure 9. The reference points are the two far ends...
Fig. 9. Rough comparison of MGM to two extremes on the scale of existing models (higher is better).

of a hypothetical scale of potential models: a highly detailed single-core simulator and a shallow analytical model (e.g., basic spreadsheet that is based on some event counts/statistics). The axis has five score levels, from 1 (low) to 5 (high), for a given parameter.

While detailed simulators can provide high per-feature accuracy for a particular architectural feature (e.g., simulate Skylake uarch while taking out just the CMOV improvement), they require high development efforts. However, the overall accuracy when analyzing a real system (MGM) is better. Consider the “CISC-ified memory accesses” insight from Section 5.3; the use of a real compiler to build the whole application as well as profiling the entire workload enabled us to discover an unexpected side effect. A simulation trace that attempts to simulate the impact of the FMA instruction would do it for a fraction of the entire application and thus is likely to miss such side effect. “Extra Vectorization” in the same section is another insight of a similar reasoning. Highly accurate simulators have low accessibility as they are often vendor proprietary. Furthermore, limited public documentation prohibits extending open simulators [47] to accurately model some third-party’s processor.

On one hand, once a simulator is developed, it can be extended to forecast changes in next generation—a drawback of MGM. On the other hand, extending MGM to cover another vendor is easier than doing so for a vendor-specific simulator, thanks to MGM reliance on a limited set of uarch-abstracted metrics based on performance counters. Lastly, extending a performance simulator to cover energy requires yet another highly detailed power simulator. Simulating multithreaded shared-memory workloads is very challenging [42]. Thus, it is easier to extend MGM to such domains as we explained earlier.
Table 5. The Key Analysis Challenges and Techniques Employed by MGM to Address Them

| Analysis challenge                                                                 | MGM technique                                                                                       |
|-----------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Complicated architectures with limited documentation                               | Start with TMA’s Abstracted Bottlenecks as an initial set of metrics.                                   |
|                                                                                   | Append a suitable performance metric to the initial set (acts as a gate-keeper).                      |
|                                                                                   | Dynamically add generic metrics (a bottom-up step that complements TMA’s abstraction).                |
| Blurring effect                                                                   |                                                                                                                                                  |
| Many architectural features are introduced in same generation with interaction      | Design microbenchmarks to verify features in isolation.                                             |
| (crosstalk) effects                                                               | Define specific metrics that capture the identified features in order to assess their spread (a bottom-up step). |

Shaded cells highlight three unique additions by MGM.

Table 6. Summary of Architectural Insights Discovered by MGM and Applicability to the SPEC Benchmarks

| Architectural insight | # of impacted benchmarks |
|-----------------------|--------------------------|
|                       | FP (17 total) | Integer (12 total) |
| Area                  |              |                    |
| Microarchitecture     |              |                    |
| Improved i-cache miss handling | 1         |                         |
| Reduced UPI for conditional moves     | 3         |                         |
| Reduced UPI for Store two-source addresses | 6         |                         |
| Removal of memory double-pumping       | 4         |                         |
| Improved 256-bit vector Divide       | 2          |                         |
| FP-adder double bandwidth          | 5          | 4                        |
| FP-adder higher latency           | –10        |                         |
| ISA                       |              |                    |
| Fused Multiply-Add (FMA)     | 11          |                         |
| Compiler                 |              |                    |
| Extra Vectorization       | 6           |                         |
| CISC-ified memory accesses | 2           |                         |

Italic denotes insights uniquely reported by this work.

8 SUMMARY

This work presents MGM, a structured method that identifies the key architectural features impacting the performance. MGM adopts a set of carefully constructed metrics in order to steer the analysis process and hence keep the involved efforts affordable. This set starts off Top-Down’s microarchitecture-abstracted bottlenecks and is dynamically adapted per the evaluation target and findings. This work also contributes a new set of designated microbenchmarks for the identified features. These microbenchmarks are not only helpful in the understanding of Skylake (e.g., under what conditions a feature does help?) but also are instrumental for the bottom-up creation of generic metrics, which leads to an effective analysis process.

MGM leverages some commonly used practices and metrics. The recipe for how to use them in tandem and yet achieve the goal is summarized in Table 5.

MGM was demonstrated through two experiments on the SPEC CPU2006 benchmarks. In summary, our results suggest that optimized CISC instructions were beneficial. For example, fewer numbers of uops to represent two-source stores and conditional moves, as well as the FMA instruction helped most benchmarks in the two experiments.

A recap of all architectural insights reported by this article is provided in Table 6 along with the number of impacted SPEC benchmarks. Three of these insights are first published by our analysis.
(to the best of the authors’ knowledge). While vendor public documentation may describe some aspects of the other seven features; this work contributes a quantification of the impact on the SPEC benchmarks for all 10 insights. The reported insights are helpful for processor architects and compiler engineers. The quantification may guide performance engineers to assess impact on real applications or researchers who predict performance of new workloads or processors [48].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for constructive feedback as well as Mashor Housh for technical review of early drafts of this work. We thank Intel for providing systems’ access and, in particular, Andi Kleen, Ravi Rajwar, and Sergey Maslov for their help.

REFERENCES

[1] J. M. Shalf and R. Leland. 2015. Computing beyond Moore’s law. Computer 48, 12 (2015), 14–25.
[2] S. Kaney et al. 2016. Profiling a warehouse-scale computer. ACM SIGARCH Computer Architecture News 43, 3 (2016), 158–169.
[3] Y. Sfakianakis, C. Kozanitis, C. Kozyrakis, and A. Bilas. 2018. QuMan: Profile-based improvement of cluster utilization. ACM Transactions on Architecture and Code Optimization 15, 3 (Aug., 2018), 1–25.
[4] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan. 2018. Memory hierarchy for web search. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA’18). 643–656.
[5] M. Clark. 2016. A new x86 core architecture for the next generation of computing. In 2016 IEEE Hot Chips 28 Symposium (HCS’16). 1–19.
[6] J. Dowek et al. 2017. Inside 6th-generation Intel core: New microarchitecture code-named Skylake. IEEE MICRO 11 (2017).
[7] B. Wolford, T. Speier, and D. Bhandarkar. 2017. Qualcomm CentriqTM 2400 Processor. In 2017 IEEE Hot Chips 29 Symposium (HCS’17).
[8] A. Yasin. 2014. A top-down method for performance analysis and counters architecture. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’14). 35–44.
[9] P. E. McKenney. 1999. Differential profiling. Software: Practice and Experience 29, 3 (1999), 219–234.
[10] L. Gwennap. 2017. Server processor competition heats up. Microprocessor Report 4 (2017).
[11] Intel Corporation. "Intel® 64 and IA-32 Architectures Optimization Reference Manual," Intel. [Online]. Retrieved on May 2017 from http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.
[12] A. Yasin, Y. Ben-Asher, and A. Mendelson. 2014. Deep-dive analysis of the data analytics workload in cloudSuite. In 2014 IEEE International Symposium on Workload Characterization (IISWC’14). 202–211.
[13] J. Haj-Yihia, A. Yasin, and Y. Ben-Asher. 2015. DOEE: Dynamic optimization framework for better energy. In High Performance Computing Symposium, Society for Computer Simulation International.
[14] Intel Corporation. "Intel® Microarchitecture Code Named Ivy Bridge Events." [Online]. Retrieved on May 2017 from https://download.01.org/perfmon/index/ivybridge.html.
[15] Intel Corporation. "Intel® Microarchitecture Code Named Skylake Events." [Online]. Retrieved on May 2017 from https://download.01.org/perfmon/index/skylake.html.
[16] A. Yasin. 2017. Performance analysis in out-of-order cores. Technion (2017). [Online]. Retrieved on May 2018 from https://webcourse.cs.technion.ac.il/234267/Winter2016-2017/ho/WCFiles/Perf%20Analysis%20in%20OOO%20cores%20-%20Ahmad%20Yasin.pdf.
[17] X. Liu and B. Wu. 2015. ScaAnalyzer: A tool to identify memory scalability bottlenecks in parallel programs. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 1–12.
[18] P. Hammarlund et al. 2014. Haswell: The fourth-generation Intel core processor. IEEE MICRO, 2 (2014). 6–20.
[19] "toplev manual · andikleen/pmu-tools Wiki · GitHub." [Online]. Retrieved on May 2017 from https://github.com/andikleen/pmu-tools/wiki/toplev-manual.
[20] C.-K. Luk et al. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In ACM Sigplan Notices 40 (2005), 190–200.
[21] "Intel® Software Development Emulator Download | Intel® Software." [Online]. Retrieved on May 2017 from https://software.intel.com/en-us/articles/pre-release-license-agreement-for-intel-software-development-emulator-accept-end-user-license-agreement-and-download.
[22] A. Carvalho. 2010. The new linux ‘perf’ tools, presented at the linux kongress.
A Metric-Guided Method for Discovering Impactful Features and Architectural Insights  46:25

[23] Intel Corporation. 2014. "Intel® 64 and IA-32 Architectures Software Developer Manuals." [Online]. Retrieved on May 2017 from http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[24] J. Haj-Yihia, A. Yasin, Y. B. Asher, and A. Mendelson. 2016. Fine-grain power breakdown of modern out-of-order cores and its implications on Skylake-based systems. ACM Transactions on Architecture and Code Optimization 13, 4 (2016), 1–25.

[25] A. Yasin, A. Mendelson, and Y. Ben-Asher. 2019. Tuning performance via metrics with expectations. IEEE Computer Architecture Letters 18, 2 (2019), 91–94.

[26] P. Hammarlund et al. 2014. Haswell: The fourth-generation Intel core processor. IEEE Micro 2 (2014), 6–20.

[27] Intel Corporation. "Intel® VTune™ Amplifier XE." [Online]. Retrieved on May 2017 from http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[28] A. Jaleel. 2010. Memory characterization of workloads using instrumentation-driven simulation.” [Online]. Retrieved from http://www.jaleel.org/ajaleel/workload/.

[29] L. Adhianto et al. 2010. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience 22, 6 (2010), 685–701.

[30] Google. "Gooda - pmu event analysis package - Google Project Hosting," 03-Feb-2014. [Online]. Retrieved on May 2017 from http://code.google.com/p/gooda/.

[31] C. Coarfa, J. Mellor-Crummey, N. Froyd, and Y. Dotsenko. 2007. Scalability analysis of SPMD codes using expectations. In Proceedings of the 21st Annual International Conference on Supercomputing (ICS’07), 13.

[32] K. Czechowski et al. 2014. Improving the energy efficiency of big cores. In Proceedings of the 41st Annual International Symposium on Computer Architecture. 493–504.

[33] Y. S. Shao and D. Brooks. 2013. ISA-independent workload characterization and its implications for specialized architectures. In 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’13), 245–255.

[34] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks. 2014. Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures. In 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA’14), 97–108.

[35] S. Van den Steen et al. 2015. Micro-architecture independent analytical processor performance and power modeling. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’15).

[36] D. Tam, R. Azimi, and M. Stumm. 2007. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. ACM SIGOPS Operating Systems Review 41, 3 (2007), 47–58.

[37] S. Zhuravlev, J. C. Saiz, S. Blagodurov, A. Fedorova, and M. Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. ACM Computing Surveys 45, 1 (Nov. 2012), 1–28.

[38] A. Nowak, P. Szostek, A. Yasin, and W. Zwaenepoel. 2015. Low-overhead dynamic instruction mix generation using hybrid basic block profiling. In 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS’18), 189–198.

[39] A. Nowak, A. Yasin, A. Mendelson, and W. Zwaenepoel. 2015. Establishing a base of trust with performance counters for enterprise workloads. USENIX ATC 9 (2015).

[40] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. 2013. CPI 2: CPU performance isolation for shared compute clusters. In Proceedings of the 8th ACM European Conference on Computer Systems. 379–391.

[41] A. Venkat and D. M. Tullsen. 2014. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA’14), 12.

[42] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC’11), 1.

[43] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. 2011. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In 2011 IEEE International Symposium on Workload Characterization (IISWC’11), 38–49.

[44] A. A. Newell and S. Pupyrev. 2018. Improved basic block reordering. ArXiv:1809.04676 [cs]. (Sept. 2018).

[45] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. 2009. Evaluation of the Intel® Core™ i7 turbo boost feature. In IEEE International Symposium on Workload Characterization (IISWC’09), 188–197.

[46] A. Yasin and Y. Ben-Asher. 2018. A first-order approximation of microarchitecture energy-efficiency. In Proceedings of the 18th International Conference on Embedded Computer Systems Architectures, Modeling, and Simulation (SAMOS’18), 38–46.

[47] N. Binkert et al. 2011. The gem5 simulator, ACM SIGARCH Computer Architecture News 39, 2 (2011), 1–7.

[48] Y. Wang, V. Lee, G.-Y. Wei, and D. Brooks. 2019. Predicting new workload or CPU performance by analyzing public datasets. ACM Transactions on Architecture and Code Optimization 15, 4 (2019), 1–21.

Received June 2019; revised September 2019; accepted October 2019