2018 IEEE International Symposium on Performance Analysis of Systems and Software

# Extending the Performance Analysis Tool Box: Multi-Stage CPI Stacks and FLOPS Stacks

Stijn Eyerman, Wim Heirman, Kristof Du Bois, Ibrahim Hur
Intel Corporation
{stijn.eyerman,wim.heirman,kristof.du.bois,ibrahim.hur}@intel.com

***Abstract*—CPI stacks are an intuitive way to visualize processor core performance bottlenecks. However, they often do not provide a full view on all bottlenecks, because stall events can occur concurrently (e.g., an instruction cache miss and a data cache miss). To not double-count penalties, typically one of the events is selected, which means information about the non-chosen stall events is lost. Furthermore, we show that there is no single correct CPI stack: stall penalties can be hidden, can overlap or can cause second-order effects, making total CPI more complex than just a sum of components.**

**Instead of showing a single CPI stack, we propose to measure multiple CPI stacks during program execution: a CPI stack at each stage of the processor pipeline. This representation reveals all performance bottlenecks and provides a more complete view on the performance of an application. Additionally, we propose FLOPS stacks, targeted at HPC performance analysis. FLOPS stacks are a variant of CPI stacks at the issue stage, but instead of considering all instructions, they focus at floating point performance specifically, which is the common definition of useful work in the HPC domain.**

**Multi-stage CPI stacks and FLOPS stacks are easy to collect. We show that they can be included in a simulator with negligible slowdown, and we provide recommendations how to include them in a hardware core.**

## I. Introduction

Analyzing the performance of an application on current high-performance processors is a challenging task. Out-of-order execution, multiple cache levels, concurrent memory accesses and superscalar pipelines are only a few of the performance-enhancing techniques that introduce large complexity and that increase the inability of easily detecting the main performance bottlenecks. Cycles per instruction (CPI) stacks [2] have been proposed as an intuitive way of visualizing performance bottlenecks. CPI stacks divide total CPI (the reciprocal of IPC) into components that represent the impact of a certain event on overall performance, see Figure 1. The bottom component, called the base component, represents the lowest possible CPI (CPI is a lower-is-better metric), which is one over the pipeline width (the maximum number of instructions that can be executed per cycle). The other components show the impact of multiple stall events (e.g., cache and branch predictor misses) that make performance lower than the ideal case. The components are calculated such that each component is proportional to its impact, and the sum of all components equals total CPI. Hence its representation as a stacked bar, and its name: CPI stacks.

Due to the complexity of the processor, assigning stall cycles to a certain component is not always unambiguous.

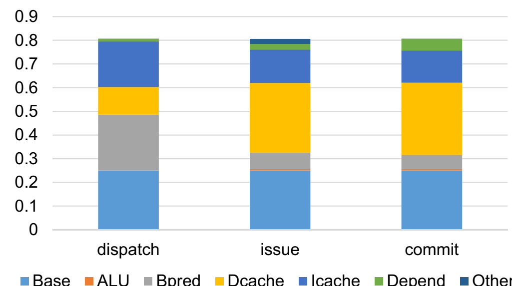


Fig. 1. Example CPI stacks at dispatch [8], issue and commit [14].

For example, if the frontend of the core has an instruction cache (Icache) miss, while the backend is seeing a data cache (Dcache) miss on another instruction, which event should be blamed? One should avoid double-counting cycles, which would lead to a stack that is higher than the total CPI. A rigorous approach is to account penalties at one particular stage in the processor pipeline and to determine the root cause of why this stage is not able to exploit its full width at the current cycle.

Eyerman et al. [8] propose to perform the accounting at the dispatch stage, which is the stage where instructions leave the frontend (consisting of fetch, branch prediction and decode) and are assigned to a reorder buffer (ROB) entry and a reservation station (RS) entry. If no or fewer than the dispatch width instructions can dispatch in a cycle, a stall cycle is detected, and the cause of the stall is determined. For example, if the frontend cannot deliver new instructions because of an Icache or branch predictor (bpred) miss, the stall cycle is assigned to the respective component. It also occurs that the ROB or RS are full, blocking dispatch despite the availability of new instructions. In that case, the instruction at the head of the ROB is inspected, because its inability to be committed from the ROB caused the ROB to fill up. If that instruction is waiting on a Dcache miss to complete, the stall cycle is assigned to the Dcache miss component. If not, it could be a long-latency instruction (e.g., a division) or an instruction whose execution is delayed due to inter-instruction dependences.

The IBM POWER CPI accounting approach [14], on the other hand, performs the accounting at the commit stage. A stall cycle is defined as a cycle in which fewer instructions than the commit width are committed. The causes here could

0-7695-6375-9/18/$31.00 ©2018 IEEE
DOI 10.1109/ISPASS.2018.00031



IEEE computer society

TABLE I
CPI COMPONENTS BY IDEALIZING STRUCTURES.

| App & core | Config | CPI | Diff. CPI |
|---|---|---|---|
| mcf on KNL | All real | 1.41 | |
| | 1-cycle ALU | 1.38 | 0.02 |
| | perfect Dcache | 1.11 | 0.30 |
| | perf. Dcache & 1-cyc. ALU | 1.05 | 0.36 |
| mcf on BDW | All real | 0.72 | |
| | perfect bpred | 0.39 | 0.33 |
| | perfect Dcache | 0.43 | 0.29 |
| | perfect bpred & Dcache | 0.25 | 0.47 |

be an empty ROB, caused by a frontend miss (Icache or bpred miss), or that the instruction at the head of the ROB is not yet finished, caused by a Dcache miss or a long-latency instruction.

Although both approaches look very similar, they have subtle differences, which could result in different CPI stacks, see Figure 1. For example, on an Icache miss, the dispatch stage accounting mechanism accounts the full miss latency to the Icache miss component (because the frontend is stalled for that period of time), while the commit stage mechanism starts accounting only after the ROB is completely empty, which could be several cycles later, or even never, if the ROB was relatively full and the miss latency is low (e.g., upon an L2 cache hit). On the other hand, a Dcache miss is accounted as soon as the instruction that causes it is at the head of the ROB for the commit mechanism. For the dispatch mechanism, accounting starts when the ROB is completely full, which is several cycles later, or even never if the ROB contained few instructions and the miss has low latency.

The question that arises, and that has been subject of debate, is "which one is the correct one?" We claim that both stacks are in fact correct. The ambiguity is a result of the complexity and parallelism in the core organization, meaning that there is no single correct CPI stack representation. To show this, we performed the following experiment. We simulate the execution of an application on a core with all stall events modeled and record the CPI. Next, we make one component (e.g., Icache, bpred or Dcache) perfect (always hit) and simulate again. Intuitively, the CPI component of that event equals the difference in CPI between both simulations.

The results for two such experiments are shown in Table I (see Section IV for our setup). The first example, mcf simulated on an Intel Knight's Landing (KNL) core configuration sees a $0.02$ CPI reduction if all ALU instructions take 1 cycle. Making the Dcache perfect (but keeping realistic ALU latencies) reduces the CPI by $0.30$. However, when assuming a perfect Dcache *and* single-cycle ALU, CPI is reduced by $0.36$, which is larger than the sum of the individual improvements. Put differently, if the Dcache is perfect, the ALU component equals $1.11 - 1.05 = 0.06$, while it is only $0.02$ initially. The ALU stalls are initially mostly hidden by the Dcache misses, but become apparent when the Dcache misses are removed.

For mcf on an Intel Broadwell (BDW) core configuration, CPI decreases by $0.33$ with a perfect branch predictor and by $0.29$ with a perfect Dcache. Perfect branch prediction *and* a perfect Dcache reduces CPI by $0.47$, which is now *less* than the sum of the individual components. These miss penalties overlap: their combined improvement is smaller than the sum of their individual components. It is impossible to represent both hidden and overlapping stalls in a stacked representation that adds to the total CPI.

We can conclude that a single additive CPI stack is a too simple representation for performance bottlenecks. Instead, we propose to measure multiple CPI stacks at different stages in the pipeline. This reveals all bottlenecks at all stages. The different CPI stacks show the range of the possible CPI reduction if a certain stall event is eliminated. For example, the bpred component for mcf on a BDW core (second example in Table I) is $0.39$ for the dispatch CPI stack and $0.11$ for the commit CPI stack. The actual CPI reduction of a perfect branch predictor is $0.33$, which is in between both numbers.

Note that multi-stage CPI stacks cannot *exactly* predict the performance gain by removing a component. An exact prediction requires extensive critical path analysis [9], which considerably slows down simulation or requires complex hardware. Multi-stage CPI stacks give more information than single CPI stacks by providing an upper and lower bound, while at the same time being relatively easy to measure.

While CPI, and its counterparts IPC and MIPS (million instructions per second), are commonly used performance metrics for general purpose applications, the performance of (mainly scientific) high-performance compute (HPC) applications is often expressed in floating point operations per second (FLOPS). Floating point operations are considered "useful" operations in scientific applications, as opposed to peripheral instructions such as memory operations and branches. Because not all instructions in an HPC application are floating point operations, FLOPS cannot be directly calculated from IPC or MIPS. Therefore, we propose an alternative stack representation, targeted at the HPC domain, called FLOPS stacks. FLOPS stacks are in essence CPI stacks at the issue stage (where instructions start executing on the functional units), limited to floating point functional units. Optimal FLOPS is reached when all floating point compute capacity is in use. Stall cycles are defined as cycles where less than the full capacity is used, which could be due to a miss event as in CPI stacks, but also due to the unavailability of floating point instructions. FLOPS stacks are an alternative representation for HPC application developers or architects designing processors for the HPC industry.

In summary, this paper presents the following contributions:

- We show that there is no single correct CPI stack representation.
- We propose the concept of multi-stage CPI stacks and show that they include more information than a single CPI stack, by providing upper and lower performance bounds of the potential performance improvement.
- We develop low complexity algorithms for measuring CPI stacks in a simulator and in hardware, and show that adding this feature in a simulator has negligible impact on simulation speed.

- We propose an alternative representation for the issue stage CPI stack, called the FLOPS stack, targeted at the HPC domain.

After discussing related work, we present our algorithms and validate the premise that multi-stage CPI stacks provide more information than single CPI stacks using simulated data. We also show that FLOPS stacks can give additional causes of low FLOPS, which are not visible in CPI stacks. We present our conclusions at the end of the paper.

## II. Related Work

A CPI stack is a well-known and commonly used concept [2], [8], [13], [14]. Constructing CPI stacks on in-order processors is relatively straightforward: a stall at one stage will stall the whole pipeline, meaning that there is almost no overlap between miss events. In addition, most latencies in an in-order processor are fixed, meaning that a simple event count multiplied by latency provides a good CPI component approximation [2]. As explained in the introduction, overlaps in current superscalar out-of-order processors makes cycle accounting a lot more complicated.

Several CPI accounting mechanisms have been proposed. In the introduction, we discussed the proposal by Eyerman et al. [8], who do the accounting at the dispatch stage, and the IBM POWER approach [14], that uses the commit stage as the accounting point. A mixed approach is taken by Yasin [17]. In his hierarchical accounting mechanism, a top level stack is measured at the dispatch stage, discerning between frontend and backend stalls, but without subdividing these into specific miss events (Icache, Dcache, bpred misses, etc.). In the next levels, specific miss event penalties are measured at different stages in the pipeline: front-end miss events at the dispatch stage, and back-end miss events at the issue stage. As a result, the components at the lower levels do not add up to the total cycle count. Instead, one should start with inspecting the top level stack: if that has a large frontend component, only the frontend components on the next levels should be considered, despite the fact that the backend components on the next levels can also have a large magnitude, and vice versa. This representation is more complex than a simple CPI stack. Furthermore, we will show that a stack measured at the dispatch stage, which is the top level stack in Yasin's proposal, prioritizes frontend misses, potentially underestimating the impact of backend misses.

Eyerman and Eeckhout [7] proposed a mechanism to measure per-thread CPI stacks on a simultaneous multithreading (SMT) processor. Their starting point is a dispatch stage CPI stack. Their proposal could be easily extended to SMT CPI stacks at other stages, in line with the algorithms described in Section III.

The intuitiveness of a stacked representation has inspired other performance analysis proposals. Speedup stacks [6] show why a parallel application does not reach a speedup linear to the number of threads. Criticality stacks [4] visualize the criticality of each thread in a parallel application. Speeding up critical threads has more impact on performance than speeding up less critical threads. Bottle graphs [5] are an extension to criticality stacks, with an extra parallelism dimension on the Y-axis. They provide an intuitive visualization of parallel performance, in particular of irregular applications. All of these are targeted at parallel applications, while the CPI stacks presented in this paper are measured per core. For multithreaded applications, CPI stacks of each individual core can be aggregated to a single CPI stack [10].

## III. Measuring Multi-Stage CPI Stacks and FLOPS Stacks

The general principle for constructing a CPI or FLOPS stack is similar for all stages. At each cycle, a stall is accounted when no instructions or fewer instructions than the width of the pipeline are processed. On a stall, the ground cause of the stall is inspected, which can be either the inability of the previous stage to deliver new instructions, or a stall event in the current or next stages. Note that each stage has a separate set of counters, e.g., the Icache miss counter at the dispatch stage is different from the Icache miss counter at the commit stage.

### A. CPI Stack Algorithm

Table II shows the accounting algorithm at three crucial stages in the core pipeline: dispatch, issue and commit. Similar accounting can be done at other stages (e.g., fetch and decode). In the algorithms, $W$ is the width of the stage (the maximum number of instructions that can be processed per cycle) and $n$ the number of (correct-path) instructions processed in this cycle (which is between 0 and $W$). FE stands for frontend pipeline (fetch, branch prediction, decode), ROB stands for reorder buffer and RS for reservation stations (or issue queue).

In these algorithms, simplified for clarity, we measure 6 components: the base component (base_comp) for time spent in actually executing instructions; the branch predictor component (bpred_comp) for time spent in resolving branch mispredictions; the instruction cache and data cache component (Icache_comp and Dcache_comp) for time spent in misses in the instruction and data cache (and TLB); the ALU latency component (ALU_lat_comp) for cycles lost due to multi-cycle latency instructions; and the instruction dependence component (depend_comp) for cycles lost due to limited instruction-level parallelism. An actual implementation could have more components, e.g., differentiating between the different cache levels and TLBs or more structural stalls in the issue stage. All components are initially zero, and keep accumulating until the end of the application.

The algorithms are executed every cycle. First, we calculate the fraction $f$ of the width that has been used this cycle and add that to the base component. Because we exclude wrong-path instructions (instructions fetched after a branch misprediction), the base component for all stacks is the same, as each correct-path instruction has to traverse all stages. In Section III-B, we discuss how we discern between correct-path and wrong-path instructions.

TABLE II
PER CYCLE CPI ACCOUNTING ALGORITHM AT DISPATCH, ISSUE AND COMMIT (PROD = PRODUCER).

| | Dispatch | Issue | Commit |
|---|---|---|---|
| 1 | f = n/W | f = n/W | f = n/W |
| 2 | base_comp += f | base_comp += f | base_comp += f |
| 3 | **if** f<1: | **if** f<1: | **if** f<1: |
| 4 | **if** FE empty: | **if** RS empty: | **if** ROB empty: |
| 5 | **if** Icache miss: | **if** Icache miss: | **if** Icache miss: |
| 6 | Icache_comp += 1−f | Icache_comp += 1−f | Icache_comp += 1−f |
| 7 | **else if** bpred miss: | **else if** bpred miss: | **else if** bpred miss: |
| 8 | bpred_comp += 1−f | bpred_comp += 1−f | bpred_comp += 1−f |
| 9 | **else if** ROB **or** RS full: | **else**: | **else if** ROB head **not** done: |
| 10 | i = ROB head | i=prod(first non−ready instr) | i = ROB head |
| 11 | **if** i has Dcache miss: | **if** i has Dcache miss: | **if** i has Dcache miss: |
| 12 | Dcache_comp += 1−f | Dcache_comp += 1−f | Dcache_comp += 1−f |
| 13 | **else if** latency[i] > 1 cyc: | **else if** latency[i] > 1 cyc: | **else if** latency[i] > 1 cyc: |
| 14 | ALU_lat_comp += 1−f | ALU_lat_comp += 1−f | ALU_lat_comp += 1−f |
| 15 | **else**: | **else**: | **else**: |
| 16 | depend_comp += 1−f | depend_comp += 1−f | depend_comp += 1−f |

If the useful fraction $f$ is smaller than 1, we try to find a reason for the stall. Lines 4 to 8 handle frontend stalls: branch mispredictions and Icache misses. Note the different point in time at which frontend miss accounting starts for each stage: when the frontend is empty for the dispatch stage, and when the reservation stations and the ROB is empty for the issue and commit stages, respectively. As a result, the frontend miss components at the dispatch stage are always larger than those at the issue stage, which in their turn are larger than those of the commit stage. Conceptually, the frontend stall component at the dispatch stage assumes that instructions that could have been fetched during the frontend stall will have no extra stalls in the backend (through dependences, multi-cycle latency instructions or Dcache misses) and thus can utilize all of the cycles in an ideal execution flow. On the other hand, the frontend stall component at the commit stage assumes that when the frontend stall is removed, the new instructions can only start executing after the ROB is drained, e.g., because they all depend on the last instruction. Clearly, the actual performance gain will be somewhere between these extremes, which is exactly the goal of multi-stage CPI stacks. Note that this reasoning assumes that all other events remain the same. Due to the coupling of events (e.g., removing data cache misses from a unified cache also has an impact on the instruction cache miss rate), this is not always the case, meaning that the actual performance gain might be larger or smaller than these boundaries.

Lines 9 through 16 handle backend stalls: Dcache misses, long-latency instructions or dependence chains (which are detected as single-cycle instructions that can only start executing when they are at the head of the ROB because of dependences on older instructions). Here, the commit stage will start accounting sooner than the dispatch stage: when the instruction at the head of the ROB is still executing versus when the ROB is completely full. The rationale is similar to frontend misses: the dispatch stage assumes that all instructions fetched after the instruction that caused the stall are independent of that instruction and cause no extra stalls. On the other hand, the commit stage assumes that these instructions have to wait until the stalled instruction is finished. Again, the actual penalty is somewhere in between, depending on the characteristics of the application.

The backend miss component is handled slightly differently at the issue stage. Instead of looking at the instruction at the head of the ROB, we look up the instruction that produces data for the first non-ready instruction. By definition, this instruction is still executing, and prevents its consumers from starting their execution. This is a more accurate instruction to blame than the head of the ROB, which could be an older instruction that is almost finished. However, at the dispatch and commit stages, we do not have the dependency information that is available at the issue stage. This ability makes the case for the issue stage CPI stack, which would otherwise seem redundant as its frontend and backend components are always in between those of the dispatch and commit stacks. Furthermore, we can also measure other structural stalls at the issue stage, such as the unavailability of functional units or issue ports, (predicted) memory address conflicts between loads and stores, etc.

In some architectures, not all stages have the same width. In particular, the issue stage is often wider than the dispatch and commit stage, in order to support all instruction types. It is clear that the ideal CPI is determined by the narrowest stage, e.g., if the issue stage can issue 6 instructions per cycle, but dispatch and commit are 4-wide, the minimum CPI is $1/4$. As a result, the base component at the wider stages will be smaller than the base component at the other stages, and the wider stages can stall even in the absence of miss events, just because of the difference in width. Instead of using the actual width of the stage, we propose to set $W$ as the minimum of all stage widths. As a result, $f$ can be larger than 1 in wider stages. In that case, we assume $f = 1$ and 'transfer' the part larger than one to the next cycle. Because the other stages are narrower, this transferred part will never grow to large values. This way of accounting also models the impact of wider stages: by issuing long-latency instructions earlier

through the wider stage, part of its latency can be hidden for later, narrower stages.

### *B. Discerning wrong-path from correct-path instructions*

In the description of the algorithms, we assume that we can discern wrong-path from correct-path instructions in the dispatch and issue stage. This is straightforward in a functional-first simulator, where the branch target is known before the timing simulation starts. However, if functional simulation is done at the simulated execute stage (execute-at-execute simulation model) or the accounting mechanism is implemented in hardware, we cannot make this distinction before the branch is actually executed. Note that there is no problem at the commit stage: wrong-path instructions are never committed.

In case wrong-path instruction detection at the dispatch and issue stage is impossible, we can take two approaches: a simple, less accurate one, and a more complex, but more accurate one. The simple approach is to initially treat all instructions as correct-path instructions. The resulting CPI stacks will have a larger base component for the dispatch and issue stage than for the commit stage. Because the commit stage has the correct base component and all base components should be equal, we can take the difference between the dispatch/issue base component and the commit base component and add that to the branch miss component. This will account for the largest part of the branch miss component related to dispatching and issuing wrong-path instructions. This approach is taken by Yasin [17]: bad speculation slots are calculated as the number of issue slots minus the number of retire (commit) slots.

A more accurate approach is the use of speculative counters. Instead of adding stall cycles directly to a global counter, the cycle components are kept in speculative counters that are kept per instruction or per basic block (as in the CPI counter architecture proposed by Eyerman et al. [8]). If an instruction or the branch ending a basic block commits, it is proven to be a correct-path instruction and its speculative counters are added to the global counter. If a branch misprediction is detected, the speculative counters of all wrong-path instructions are added to the global branch miss counter. Because this technique incurs a storage and control overhead, we foresee that it will be implemented in simulators only. For hardware implementations, the simple approach is more appropriate.

### *C. FLOPS Stacks*

FLOPS stacks are an alternative representation of the CPI stack at the issue stage. Instead of focusing on all instructions, FLOPS stacks only consider floating point operations, as they are considered the only “useful” instructions in scientific HPC applications and are the target of performance analysis methods such as the roofline model [16]. FLOPS stacks are measured at the issue stage, because maximum FLOPS performance is determined by the number of (vector) floating point units, and not by the dispatch or commit width. The FLOPS stack accounting algorithm is given in Table III. The maximum FLOPS is obtained by using all vector floating point units (e.g., AVX2 or AVX512), without masking (which uses only part of the vector), and executing fused multiply-add (FMA) instructions. The latter condition stems from the fact that a multiply and add are considered two operations in the theoretical FLOPS calculation, which means that an FMA instruction doubles the floating point throughput compared to a vector addition or multiplication. So the maximum FLOPS equals $2 \cdot k \cdot v$ per cycle, with $k$ the number of vector floating point units, and $v$ the vector width (e.g., 16 single-precision floats for AVX512). The 2 reflects the 2 FMA operations.

TABLE III
FLOPS ACCOUNTING ALGORITHM.

```
1  f = a*n*m/(2*k*v)
2  base_comp += f
3  if f<1:
4    if a<2:
5      non_fma_comp += (2-a)*n*m/(2*k*v)
6    if m<v:
7      mask_comp += n*(v-m)/(k*v)
8    if n<k:
9      if no VFP insts in RS:
10       frontend_comp += (k-n)/k
11     else if VU used by non-VFP inst:
12       non_vfp_comp += (k-n)/k
13     else:
14       i = prod(oldest VFP inst)
15       if i is a memory load:
16         mem_comp += (k-n)/k
17       else:
18         depend_comp += (k-n)/k
```

In the algorithm, $n$ is the number of vector floating point operations issued, $a$ is the operation count per instruction (2 for FMA, 1 for additions or multiplications; it can be in between if multiple vector instructions are issued, e.g., 1.5 if one FMA and one addition is issued), and $m$ is the number of unmasked elements ($v$ when totally unmasked, 0 when every element is masked out). The total number of floating point operations issued in a cycle is therefore $a \cdot n \cdot m$. VFP stands for vector floating point, RS is reservation station, VU is the vector functional unit and prod(i) is the producer of instruction i (i.e., the instruction that produces a value needed by i).

FLOPS stacks have a few different components than CPI stacks. The base component (base_comp) are the cycles where the maximum FLOPS is reached. The non-FMA component (non_fma_comp) reflects the cycles lost due to non-FMA VFP instructions (simple additions, multiplications, etc.). The masking component (mask_comp) is the fraction of cycles lost due to masking vector elements. The frontend component (frontend_comp) measures the lost cycles because the frontend could not deliver VFP instructions, either because all instructions are non-VFP, or there are no instructions because of an Icache miss or bpred miss. This component could be further divided into these three components (not shown for brevity). The vector unit can also be used by non-VFP instructions (such as integer vector instructions or broadcasts), this lost VFP slot is collected by the non-VFP component (non_vfp_comp). Finally, the memory component (mem_comp) and dependence component (depend_comp) reflects losses due to VFP operations waiting on memory operations or other instructions.

The base component is the number of cycles to execute the useful work in the application assuming the maximum FLOPS ($M = 2 \cdot v \cdot k$ per cycle) is always reached. The total FLOPS of this application is therefore

$$\text{FLOPS} = \frac{\text{base_comp}}{\text{cycles}} \cdot \text{freq} \cdot M \quad (1)$$

where cycles is the total cycle count and freq is the core clock frequency. By multiplying each component by $\frac{\text{freq} \cdot M}{\text{cycles}}$, we obtain a stack with height freq $\cdot M$, which is the maximum obtainable FLOPS. The base component is the actually obtained FLOPS, and the other components reflect the causes why this maximum is not obtained. This makes the FLOPS stack an intuitive representation for FLOPS based performance analysis, allowing it to augment the roofline model by identifying specific causes why an application does not reach its theoretical performance.

The crucial difference between a FLOPS stack and a CPI stack is that when the full processor pipeline width is used by non-VFP instructions, a CPI stack detects a fully useful cycle, while the FLOPS stacks measures a full "stall" cycle. This is why FLOPS stacks can look totally different from CPI stacks, as we will show in Section V-B.

## IV. Experimental Setup

We implement the dispatch, issue and commit stage CPI stacks, and the FLOPS stacks in the Sniper [3] multi-core simulator. The simulation time increases by less than 1% compared to the original version of Sniper (which already includes measuring dispatch CPI stacks), which proves that adding multi-stage CPI stack and FLOPS stack accounting has a very small overhead. Although we have not verified it, we are convinced that including them in hardware should also be feasible with limited overhead.

To evaluate multi-stage CPI stacks, we simulate all SPEC CPU 2017 [1] single-threaded benchmarks with the reference input sets (36 benchmark-input combinations). To limit simulation time, we fast-forward 10 billion instructions and simulate 1 billion instructions into detail. Each benchmark is simulated on an Intel Broadwell (BDW; 4-wide out-of-order pipeline) and an Intel Knights Landing (KNL; 2-wide out-of-order) inspired core configuration. All uncore components are scaled down by the socket core count, in order to mimic a fully loaded processor. For example, shared cache size and memory bandwidth is divided by 18 for the BDW configuration, as our BDW socket configuration contains 18 cores.

In order to show the validity of the measured CPI stacks, we also perform simulations where certain components are idealized. In particular we simulate a perfect L1 Icache (each access hits in L1), a perfect L1 Dcache, perfect branch prediction (including perfect target prediction), and single-latency instructions (all arithmetic and logic instructions complete in 1 cycle). We collect the CPI of each simulation, and deduct it from the realistic simulation CPI to quantify the performance improvement.

Because the floating point SPEC CPU benchmarks are not well vectorized, their FLOPS value is very low. Therefore, and because FLOPS stacks are targeted at HPC applications, we evaluate FLOPS stacks on the DeepBench benchmark suite [15]. DeepBench consists of a set of kernels that are crucial for deep learning. In particular, we evaluate single precision general matrix multiplication (sgemm) and convolution (conv). We use the latest version of Intel Math Kernel Library (MKL) [12] for sgemm, and Intel MKL-DNN [11] for convolution. For sgemm, we simulated all 235 training and inference configurations. For convolution, we simulated the forward phase (fwd), the backward filter phase (bwd_f) and the backward data phase (bwd_d) of all training configurations (for a total of 3×94=282 configurations). We simulate these applications on a 68-core KNL and a 26-core Intel Skylake (SKX) processor configuration, both supporting AVX512 vectorization.

We aggregate the CPI stacks by averaging them component per component. This is possible because all threads show homogeneous behavior [10]. Similarly, we add the FLOPS stacks by their components.

## V. Experimental Validation

In this section we validate our premises that multi-stage CPI stacks provide more information than single CPI stacks by giving an upper and lower bound of the potential performance improvement, and that FLOPS stacks are a useful addition to the performance analysis of HPC applications.

### A. Multi-Stage CPI Stacks

To validate that multi-stage CPI stacks provide more information than individual stacks, we perform the following study. For the Icache, Dcache, bpred and ALU component, we select the SPEC CPU 2017 benchmarks for which the component is at least 10% of the total CPI (in any of the stacks). This filters out 'zeros': both the CPI component and the performance difference are close to 0, which means that a component has no impact for a benchmark. Keeping these zeros in would artificially increase the cases with zero error. For these benchmarks, we simulate a configuration with a perfect Icache, perfect Dcache, perfect bpred, or single-cycle ALU operations. Next, we calculate the 'error' on the component: the difference between the predicted CPI component and the actual CPI reduction. We calculate the error for each of the three stacks (dispatch, issue and commit), and the multi-stage stack representation, where we assume a zero error if the actual CPI reduction is within the minimum and maximum component. If it is not within the boundaries, the error equals the error of the closest component (of the three stacks). Note that this is not an error in the conventional meaning, it is a measure of how much information we can get out of one or multiple CPI stacks. Boundaries provide more information than single values, and are correct when the actual value falls in between them.

Figure 2 shows the results for BDW and KNL (the ALU component on BDW was larger than 10% for only one

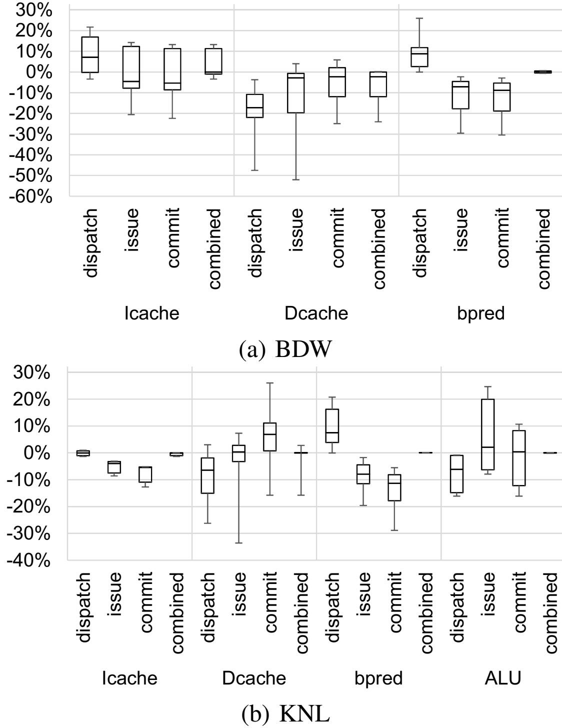


Fig. 2. Error on the components for the individual CPI stacks and the combined multi-stage CPI stacks, on (a) BDW and (b) KNL. Boxes are bound by the first and third quartile, the median is the line in the box, and the whiskers extend to the extreme values.

benchmark, so we do not show results for this one). Clearly, the multi-stage CPI stack representation has the lowest error (smaller box and median closer to zero). This reduction is most significant for the bpred and ALU components, where the error reduces to 0 (all perfect bpred or 1-cycle ALU CPI reductions are within the boundaries).

The results show that the dispatch stage on average overestimates the Icache and bpred component, and underestimates the Dcache component. The commit stack behaves the other way around: underestimating the frontend components (Icache and bpred) and overestimating the Dcache component. This means that none of the stacks is more accurate than the others for all components: the dispatch stack is better at estimating frontend components, while the commit stacks is more accurate for backend components. The combined stacks for the Icache and Dcache components on BDW still show a significant error. The reason for this is the high coupling of Icache and Dcache misses through the unified cache, as we will show in the discussion of the individual examples in the next paragraphs.

Figure 3 shows a selection of interesting multi-stage CPI stacks. The top graph, mcf on the BDW core configuration, is the second example of Table I. The dispatch stack 'predicts' a CPI reduction of 0.39 for a perfect branch predictor, which is closer to the actual reduction of 0.33 than the 0.11 bpred component at the commit CPI stack. On the other hand, CPI reduces by 0.29 when the Dcache is made perfect, which

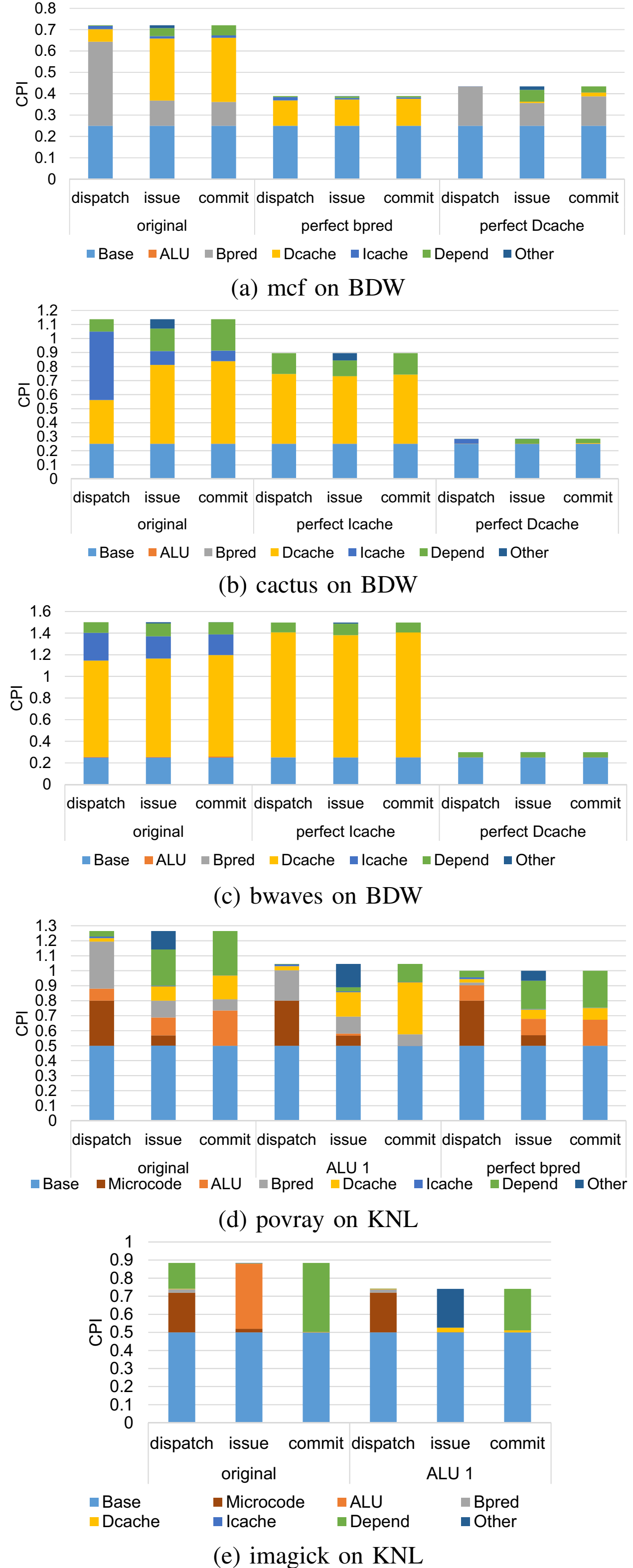


Fig. 3. Selection of multi-stage CPI stacks before and after making components perfect.

is better predicted by the commit stack (0.30) than by the dispatch stack (0.06).

Making the Icache perfect for cactus on a BDW core (Figure 3(b)), leads to a CPI reduction of 0.24, which is in between the dispatch Icache component (0.49) and the commit Icache component (0.08). A perfect Dcache, on the other hand, leads to a CPI reduction that is much larger than what any stack predicts. This occurs for some other benchmarks too, and is the cause for the non-zero error for the combined Icache and Dcache components in Figure 2. Icache and Dcache misses are highly intertwined, because all caches from level two and higher contain both instructions and data. When the L1 Icache is made perfect, no accesses to the L2 cache are made, and no data elements are evicted by instructions. So making the Icache perfect reduces the L2 miss rate for data, and therefore the Dcache miss component also reduces. And vice-versa, the Icache component reduces when the L1 Dcache is made perfect, which is the case in this example. This is a second-order effect, which is impossible to predict with a simple accounting mechanism. Furthermore, it leads to a larger CPI reduction than predicted, which is less harmful than having a smaller improvement than predicted.

Note that the dependence component also disappears when making the Dcache perfect. To explain this effect, consider a chain of dependent instructions that start with the Dcache miss. During the Dcache miss, all independent instructions can execute (out-of-order), which means that after the Dcache miss, the chain of dependent instructions are the only instructions left in the reservation stations. Because they need to execute one after another, the full pipeline width cannot be used, leading to dependence stalls. In the absence of the Dcache miss, the dependent instructions execute together with the independent instructions, causing a full usage of the pipeline width, and no stalls. This is another example of a second-order effect, which would require complex dependence analysis to correctly predict it.

A particularly interesting case is that of the Icache component for bwaves on BDW, see Figure 3(c). All three stacks (at dispatch, issue and commit) have a larger than 0.19 Icache component, indicating that CPI would reduce with at least 0.19 when the Icache is perfect. However, the observed reduction is less than 0.01, which is not as expected. A further analysis shows that the contention for L2 miss status holding registers (MSHR) is very high. The miss rate in the L1 Icache is low, but because of the many hardware prefetches, the L2 MSHRs are contended, and Icache misses are queued for a long time until an MSHR is available. During the Icache misses, contention remains high because hardware prefetching continues even if there are no regular instructions in the ROB. If the L1 Icache is made perfect, the queuing time is transferred to the Dcache misses (the Dcache component increases), undoing the performance gain of the eliminated Icache misses. On the other hand, if the Dcache is made perfect, there are no triggers for hardware prefetches, and the CPI comes close to the ideal 0.25 value. Again, this is a higher-order effect, which cannot possibly be captured by a low-overhead accounting mechanism.

The fourth example (Figure 3(d)) shows the multi-stage CPI stacks for povray on the KNL core. Note the appearance of a new component called ‘Microcode’. Some multi-micro-operation instructions in a KNL core require a few cycles to be decoded, resulting in a stall in the dispatch stage, reflected by the Microcode component. When issue and commit stall on an empty RS or ROB, and the dispatch stage is stalled by a microcode penalty, they are also accounted microcode stall cycles. If all ALU latencies are set to one cycle (ALU 1), CPI reduces by 0.22, which is better approximated by the commit ALU component (0.23) than by the dispatch ALU component (0.08). On the other hand, making the branch predictor perfect reduces CPI by 0.26, which is in between the 0.08 predicted by the commit CPI stack and the 0.31 bpred component of the dispatch CPI stack.

For all examples, the issue stack components are in between the respective components of the dispatch and commit stack. Therefore, the issue stage CPI stacks may seem superfluous. However, as discussed in Section III, the issue stage has unique knowledge about the dependences between instructions: the cause of the issue stall is attributed to the instruction for which the first non-ready instruction is waiting. This information is not available at the dispatch and commit stack, where the ROB head is blamed for causing the stall. As an example, Figure 3(e) shows the CPI stacks for the imagick benchmark on the KNL core. The dispatch and commit stacks show that the largest stall component is dependences between single-cycle latency instructions. On the other hand, the issue stack shows that instructions mainly wait on multi-cycle ALU instructions. Setting the latency of these instructions to 1 indeed reduces CPI by 0.14, which is about the maximum possible performance improvement, as the microcode component cannot be removed.

Some of the issue CPI stacks have a relatively large ‘Other’ component. These occur when there are ready instructions, but structural stalls prevent issuing them, such as the unavailability of issue ports or predicted memory address conflicts (making a newer load wait for an older store to finish). These effects can also be separately measured in the issue CPI stack, revealing possible bottlenecks caused by structural stalls. Note that the issue stage is the only stage where these stalls can be detected.

### B. *FLOPS Stacks*

To show that FLOPS stacks complement the CPI stack analysis for HPC applications, we perform the following experiment. We measure issue stage CPI stacks and FLOPS stacks for all DeepBench applications on KNL (68 threads) and SKX (26 threads). Next, we normalize each stack, and take the difference between corresponding components of the CPI and FLOPS stack (the normalized FLOPS base component minus the normalized CPI base component, and similar for the frontend, memory and dependence components; all other components are either close to zero or exactly the same). We average all differences per set of benchmarks (sgemm train, sgemm inference, convolution forward, convolution backward

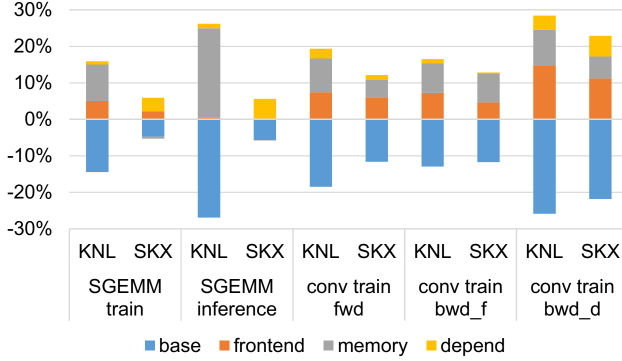


Fig. 4. Relative difference per component between the issue stage CPI stack and the FLOPS stack for the DeepBench applications on KNL and SKX.

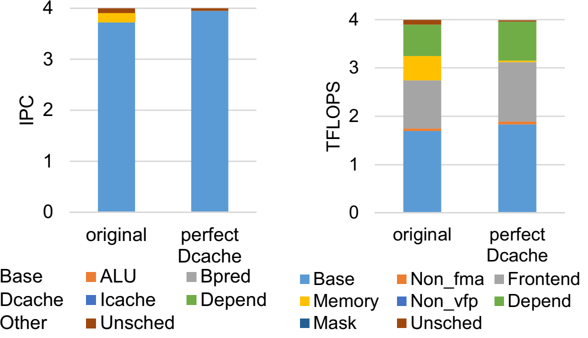


Fig. 5. IPC and FLOPS stack for one convolution train fwd configuration on SKX, without and with perfect Dcache.

filter and convolution backward data), see Figure 4. As all normalized components finally add to 1, the sum of the differences is zero (the bars above and underneath the 0% line have equal height).

The base component of the FLOPS stack is always smaller than that of the CPI stack (negative difference). The CPI base component equals the number of instructions divided by the pipeline width (2 for KNL and 4 for SKX) and the FLOPS base components is the number of VFP instructions divided by the number of VPUs (2 for KNL and SKX). For SKX, this means that to have a CPI base component equal to the FLOPS base component, at least half of the instructions need to be FMAs, which is not the case for these applications. For KNL, in order to have an equal base component, all instructions need to be FMAs, which explains why the base component difference is much larger for KNL than for SKX. Note that an ‘instruction’ here actually means a micro-operation, as the processor pipeline uses micro-operations (its width is 2 or 4 micro-operations). A VFP instruction that has a memory operand is split into two micro-operations: one load and one VFP calculation. This explains the low fraction of VFP instructions, although in terms of macro-instructions (x86 instructions), they can be in the majority.

For the sgemm benchmarks, the difference between KNL and SKX is large. The FLOPS base component for SKX is on average only 5% smaller than the CPI base component, meaning that it has a relatively high ratio of VFP instructions. The ‘compensating’ component is mainly the dependence component, meaning that even if the fraction of VFP instructions increases, they would still have to wait for dependences amongst themselves, and the FLOPS would not increase. For SGEMM train, there is small headroom to increase FLOPS by increasing the VFP fraction, because it has a 2% frontend component difference, which measures the cycles where no VFP instructions are available (there are almost no frontend misses in these benchmarks).

For sgemm on KNL, there is a much larger memory component, despite the fact that these applications do not have many Dcache misses. On KNL, the MKL just-in-time (jit) code engine uses FMA operations with a memory operand, meaning that the instruction is split into a L1 Dcache access and an FMA calculation. The latter has to wait until the load from L1 Dcache is finished, explaining the large memory component. On SKX, the same function is implemented by first loading data from memory, broadcasting the values in an AVX512 register, and using this register in multiple FMA operations without memory operand. The FMA instructions are dependent on the broadcast instruction, which is reflected in a larger dependence component.

For the convolution benchmarks, the difference between the FLOPS stack and CPI stack components is high both for KNL and SKX. The frontend component difference is a large contributor to a low base component, meaning that FLOPS can be improved by increasing the fraction of VFP instructions. There is also a 5% to 10% memory component, which means that FLOPS will improve more than CPI by reducing Dcache misses and VFP dependences on memory operations.

Figure 5 shows an example of a FLOPS stack (right). For comparison, the IPC stack is also shown. An IPC stack uses the same counters as a CPI stack, but instead of dividing each component by the number of instructions (to obtain CPI), we divide by the number of cycles and multiply with the maximum IPC. This is a similar procedure as for FLOPS stacks, see Equation 1. The base component is now the obtained IPC, while the top is the maximum IPC. IPC stacks and FLOPS stacks have the same unit, namely instructions per time, which makes them more intuitive to compare. The maximum IPC is 4 and the maximum FLOPS is 4 TFLOPS (coincidentally the same number). The ‘Unsched’ component is the IPC or FLOPS lost because some of the threads are yielded due to synchronization.

In this example, the IPC is almost ideal (3.7), while the obtained FLOPS is only 43% of the maximum (1.7 out of 4 TFLOPS). The FLOPS stack shows the reasons for the low FLOPS: (a) frontend, which means too few VFP instructions, (b) memory, that is VFP instructions waiting on memory operations, and (c) depend, which reflects dependences between instructions. The instruction type mix indeed shows that only 35% of the instructions are vector FMA instructions, and each have a memory operand, which effectively halves the

FMA instruction count. Interestingly, the memory component in the FLOPS stack is bigger than that of the IPC stack, suggesting a bigger FLOPS gain with an ideal memory. Making the Dcache perfect, however, lets both IPC and FLOPS increase with 0.2. In the new FLOPS stack (far right), the frontend and depend component have grown. VFP instructions that originally waited for memory, are now stalled by other dependences. Furthermore, by removing the memory waiting stalls, the fraction of cycles with non-FP instructions also increases (frontend component).

## VI. Conclusions

CPI stacks are an intuitive way to visualize performance bottlenecks in a processor core. However, because of superscalar and out-of-order execution, single CPI stacks are an overly simple representation and hide important stall information. We propose to measure multiple CPI stacks at different stages in the pipeline. This representation shows all stall events, indicating the range of the performance improvement that is expected when a stall event is eliminated.

Additionally, we present FLOPS stacks, which are alternative representations for issue stage CPI stacks. FLOPS stacks are targeted at HPC applications, where floating-point performance is more important than raw CPI. By focusing on (vector) floating point operations, we obtain a stack that can be very different from the issue stage CPI stack, and that indicates whether a low FLOPS value is attributed to a low fraction of FP instructions, dependence chains, Dcache misses, or other structural stalls.

Our experiments show that in most of the cases, the actual performance improvement is within the boundaries predicted by the dispatch, issue and commit CPI stacks. The cases where the performance improvement is not within these boundaries can be attributed to second-order effects that cannot be predicted with a simple accounting mechanism. Furthermore, none of the three stacks is consistently more accurate than the others. We also show that FLOPS stacks are complementary to CPI stacks, providing more information on how a low FLOPS number can be increased, even when IPC is close to optimal.

Adding the infrastructure to measure the stacks has a negligible impact on simulator performance. We can conclude that multi-stage CPI stacks and FLOPS stack are a valuable addition to the analysis toolbox of a simulator or to the performance counter infrastructure on a core.

## References


[1] “SPEC CPU2017 benchmark suite,” https://www.spec.org/cpu2017/.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood, “DBMSs on a modern processor: Where does time go?” in *VLDB 99, Proceedings of 25th International Conference on Very Large Data Bases*, 1999, pp. 266–277.

[3] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, “An evaluation of high-level mechanistic core models,” *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014.

[4] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, “Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13, 2013, pp. 511–522.

[5] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, “Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications,” in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’13, 2013, pp. 355–372.

[6] S. Eyerman, K. Du Bois, and L. Eeckhout, “Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications,” in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, 2012, pp. 145–155.

[7] S. Eyerman and L. Eeckhout, “Per-thread cycle accounting in SMT processors,” in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009, pp. 133–144.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, “A performance counter architecture for computing accurate CPI components,” in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XII, 2006, pp. 175–184.

[9] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn, “Interaction cost and shotgun profiling,” *ACM Trans. Archit. Code Optim.*, vol. 1, no. 3, pp. 272–304, Sep. 2004.

[10] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, “Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads,” in *2011 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2011, pp. 38–49.

[11] Intel, “Intel math kernel library for deep neural networks (mkl-dnn),” https://github.com/01org/mkl-dnn.

[12] ——, “Intel math kernel library (mkl),” https://software.intel.com/en-us/mkl.

[13] Y. Luo, J. Rubio, L. K. John, P. Seshadri, and A. Mericas, “Benchmarking internet servers on superscalar machines,” *Computer*, vol. 36, no. 2, pp. 34–40, 2003.

[14] A. Mericas, “Performance monitoring on the POWER5 microprocessor,” *Performance Evaluation and Benchmarking*, pp. 247–266, 2006.

[15] S. Narang, “Deepbench,” https://svail.github.io/DeepBench/.

[16] S. Williams, A. Waterman, and D. A. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Communications of the ACM*, 2009.

[17] A. Yasin, “A top-down method for performance analysis and counters architecture,” in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2014, pp. 35–44.