2025 Design, Automation & Test in Europe Conference (DATE 2025)

SPIRE: Inferring Hardware Bottlenecks from Performance Counter Data

Nicholas Wendt
University of Michigan
Ann Arbor, Michigan, USA
nwendt@umich.edu

Mahesh Ketkar
Intel Corporation
Folsom, California, USA
mahesh.c.ketkar@intel.com

Valeria Bertacco
University of Michigan
Ann Arbor, Michigan, USA
valeria@umich.edu

***Abstract*—The persistent demand for greater computing efficiency, coupled with diminishing returns from semiconductor scaling, has led to increased microarchitecture complexity and diversity. Thus, it has become increasingly difficult for application developers and hardware architects to accurately identify low-level performance bottlenecks. Abstract performance models, such as roofline models, help but strip away important microarchitectural details. In contrast, analyses based on hardware performance counters preserve detail but are challenging to implement.**

This work proposes SPIRE, a novel performance model that combines the accessibility and generality of roofline models with the microarchitectural detail of performance counters. SPIRE (Statistical Piecewise Linear Roofline Ensemble) uses a collection of roofline models to estimate a processor's maximum throughput, based on data from its performance counters. Training this ensemble simply requires sampling data from a processor's performance counters. After training a SPIRE model on 23 workloads running on a CPU, we evaluated it with 4 new workloads and compared our findings against a commercial performance analysis tool. We found that our SPIRE analysis accurately identified many of the same bottlenecks while requiring minimal deployment effort.

***Index Terms*—Computer architecture, performance analysis**

I. Introduction

While advancements in semiconductor fabrication continue to improve device frequency, power, and area, the rate of improvement has been diminishing in recent decades [3], [8], [22]. As a result, microarchitectural optimizations have played a key role in meeting the demand for greater computing performance and efficiency. By optimizing how a processor performs its computations, architects can often improve performance, power, and area independently of the fabrication process.

Over years of processor optimizations, architectures have become increasingly complex and diverse. General-purpose processors now have wider and deeper pipelines, more sophisticated out-of-order execution mechanisms, and often multiple, potentially different, CPU cores. The use of specialized processors (*e.g.*, accelerators) has also become increasingly common, yielding a wide variety of distinct microarchitectures [5].

While beneficial, these microarchitecture trends complicate performance analysis. Whether optimizing hardware or software, understanding a workload's performance bottlenecks is vital. However, as microarchitectures grow in complexity, more factors come into play: more execution steps, more functional units, *etc.* Further, different processors may have different characteristics, and knowledge gained while studying one may not transfer to the other. The ideal solution to these challenges is an analysis method that *i)* accurately identifies bottlenecks in complex designs while *ii)* easily applying to any processor.

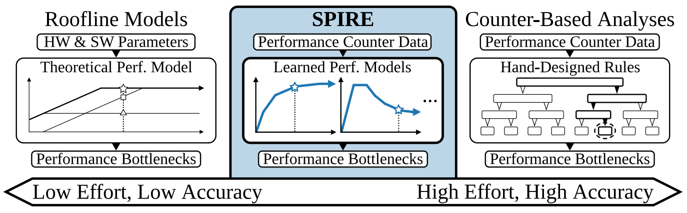

Fig. 1. A qualitative comparison between SPIRE and two related performance analysis methods. All three can be used to identify performance bottlenecks, but they entail different tradeoffs in both accuracy and implementation effort. SPIRE builds upon roofline models and performance counter analyses to achieve higher accuracy than the former without the latter's high effort.

Roofline models are a common analysis tool that has been applied to both general-purpose and specialized processors [7], [10], [15], [20], [23]. They use relatively simple information about a processor and workload to produce an intuitive graphical model that illustrates the workload's bottlenecks and the processor's performance capabilities. However, this approach tends to abstract away much of a processor's microarchitectural details, yielding inaccurate performance estimations. While some variants integrate more of this information (*e.g.*, [4], [18]), they lose generality by targeting a specific processor type.

Another common performance analysis tool are *hardware performance counters*. These special components, present in most modern processors, measure elements of hardware activity by counting retired instructions, cache misses, branches, *etc.* To accurately identify performance bottlenecks with the vast amounts of low-level data these produce, processor vendors often provide analysis tools. The main downside of counter-based analysis arises when such a tool is not available for a processor (*e.g.*, during its design). While some general strategies exist [24], implementing effective analyses from scratch can require significant engineering effort and hardware expertise.

In this paper, we propose SPIRE, a novel performance model that bridges the gap between the accessibility and generality of roofline models and the accuracy of counter-based analyses (Figure 1). The Statistical Piecewise Linear Roofline Ensemble (SPIRE) model fits to a processor using data sampled from its performance counters. This model can then be interpreted to infer likely microarchitecture-level performance bottlenecks. Our methodology is architecture-independent and can be immediately applied to any processor microarchitecture.

To demonstrate SPIRE, we modeled a real CPU using data from 23 high-performance computing workloads. We then analyzed this model and compared it against a counter-based analysis tool. In summary, we make the following contributions:

- We propose SPIRE, a novel performance model that automatically learns a processor's performance characteristics using data from its hardware performance counters.
- We demonstrate how a SPIRE model can be used to identify microarchitecture-level performance bottlenecks by analyzing a Xeon Gold 6126 CPU.
- We validate the SPIRE model against VTune [11], an industry-standard, counter-based performance analysis tool, and find that it identifies many of the same bottlenecks while requiring minimal effort to implement.

II. Background

A. Roofline Models

A basic roofline model [20], [23] has two processor-specific parameters: its maximum throughput (π in FLOP/s) and maximum memory bandwidth (β in byte/s). Each analyzed workload is characterized by its *operational intensity*, that is, the number of floating-point operations it performs per byte of memory traffic (I in FLOP/byte). The model estimates the maximum performance that a workload can achieve on the processor (P in FLOP/s) using the equation: $P(I) = \min(\pi, \beta I)$.

This model is typically plotted for visual analysis. As shown by the topmost lines in Figure 2's example, the βI and π terms produce memory and compute *ceilings*, respectively. Correspondingly, a workload is considered either *compute-bound* (*e.g.*, App A) or *memory-bound* (*e.g.*, App B). A common practice is to plot additional ceilings that represent lower throughput limits. Figure 2 includes ceilings for DRAM's lower bandwidth and scalar execution's lower throughput.

The simple structure of this model makes it easy to use and allows it to generalize to many types of processors. However, this simplicity limits its ability to help accurately identify low-level bottlenecks. First, it assumes that memory accesses and computations always happen in parallel. If this parallelism is limited by the workload or processor (*e.g.*, instruction dependencies), it can overestimate throughput [4], [18].

Second, the model is primarily one-dimensional; it focuses on how operational intensity influences performance. While ceilings can be added to incorporate more information, their discrete nature limits their modeling capabilities. In addition, they must be manually interpreted to decide which are relevant to a workload, reducing their scalability.

B. Hardware Performance Counters

Most modern microarchitectures include hardware performance counters, specialized registers that can track low-level quantities such as counts of cache misses, branch mispredictions, and execution stalls. Notably, many of these values would otherwise be nearly impossible to measure precisely, outside of a simulation. However, while a modern processor may expose hundreds or thousands of measurable quantities, most will only have a small number of counters (often <10 per core). So, only a small subset of these values may be measured simultaneously.

Accurately interpreting performance counter data (*e.g.*, to identify bottlenecks) can be very challenging due to the complexity of modern microarchitectures. For commercial processors, this data is typically analyzed using tools such as Linux Perf [21], Intel VTune [11], ARM Streamline [1], or NVIDIA Nsight Compute [19]. However, among the many stages of a processor's development, only a few may be equipped with a suitable analysis tool. While some general strategies exist (*e.g.*, Top-Down Analysis [24]), the analyses employed by these tools have been specifically engineered and tuned for their associated processors. Achieving a similar level of analysis quality for a new design can take significant engineering effort.

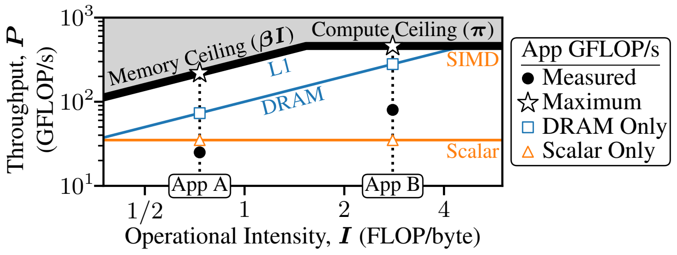

Fig. 2. A roofline model plot with I and P measured for 2 apps. Considering the processor's maximum capabilities, App A is memory-bound and B is compute-bound. However, both will be further limited if they only use scalar instructions or DRAM memory accesses.

III. SPIRE Performance Model

The SPIRE performance model estimates the highest throughput a workload can achieve on a processor, based on a set of *performance metrics*, that is, values measured by the processor's performance counters. The model is structured as an *ensemble* of many independent roofline models, each associated with one of these metrics. Given a workload's performance data, each roofline estimates the workload's maximum throughput using its associated metric. The lowest of these roofline estimations becomes the final, ensemble-wide throughput value. Beyond simply estimating attainable throughput, a workload's performance can be analyzed by inspecting the model's roofline estimations. The rooflines that produce the lowest values are most likely to be associated with a performance bottleneck.

Below we present the SPIRE model by discussing its *a)* input data, *b)* individual performance metric rooflines, and, finally, *c)* the overall ensemble structure. For clarity, we present our roofline fitting algorithms at the end of the section.

A. Input Data

Hardware performance counters can measure a limited number of values simultaneously. To account for this, SPIRE's input data consists of elements called *samples*.

Sample composition. A sample represents some period during which the processor under analysis was executing a workload. Each sample contains five values that describe this period:

1) T, length of the period
2) W, quantity of work completed during the period
3) M_x, increase of performance metric x within the period
4) P, average throughput ($P = W/T$)
5) I_x, metric-specific operational intensity ($I_x = W/M_x$)

A sample is associated with a single performance metric, such as a cache miss count, a stall count, *etc.* Its M_x value measures how much the associated metric increased during the sample period (*e.g.*, M_{stalls} is the number of stalls that occurred during the period). As in a conventional roofline model, the operational intensity value (I_x) will be a model input. However, its specific meaning depends on the associated metric (*e.g.*, I_{stalls} is the quantity of work completed, on average, per stall).

Unlike M_x and I_x, the units of measure for time (T), work (W), and throughput (P) must be consistent across all samples. Ideally, the units of T and W are chosen so that P's units align with the analysis goals. For example, when analyzing a processor's IPC, W should measure completed instructions and T elapsed cycles to make P's units instructions/cycles.

Collecting samples. A sample is collected by measuring W and M_x during a period of time with length T, while the processor under analysis executes a workload. The values T, W, and M_x should be measured simultaneously to ensure the accuracy of throughput (P) and operational intensity (I_x). Error in either of these derived values could harm model accuracy. Most modern processors have counters capable of simultaneously measuring T, W, and M_x. An exception is low-end designs (*e.g.*, the ARM Cortex-A5 only has two counters [2]). When extra counters are present, multiple samples may be collected concurrently by measuring M_x for multiple metrics while sharing T and W.

While collecting training data, the goal is to gather samples that maximize performance (P) over a wide range of operational intensities (I_x) for each metric. Ideally, this is done using optimized workloads specifically designed to exercise each metric (*e.g.*, microbenchmarks). However, as our evaluation demonstrates (Section V), good model accuracy can also be achieved by collecting many samples from a variety of workloads. While analyzing a workload with a trained model, the goal is to collect samples that accurately characterize it. If parts of the workload's execution are over- or underrepresented, for example, its analysis may be inaccurate.

B. Performance Metric Rooflines

Each roofline model in a SPIRE ensemble uses one metric's operational intensity (I_x) to estimate max throughput. This relationship is automatically learned based on training samples.

Qualitative model trends. The true relationship between a metric's I_x and maximum throughput depends on the processor, metric, and how throughput is measured. However, we can make three reasonable assumptions.

First, if a metric x is *negatively* associated with performance, throughput *increases* with I_x. For example, since stalls are harmful, having more instructions per stall increases IPC.

Second, if a metric is *positively* associated with performance, throughput *decreases* with I_x. Since I-cache hits are beneficial, more instructions per hit (*i.e.*, less frequent hits) reduces IPC.

Third, increasing I_x has diminishing returns (*i.e.*, throughput will "flatten out" as I_x increases). For example, going from 2 to 4 instructions per stall will raise IPC more than going from 20 to 22, even though I_x increased by 2 in both cases. Similar logic applies to positive metrics.

Combining our assumptions, negative metrics are best fit with an increasing, concave-down function, and positive metrics should use a decreasing, concave-up function.

Piecewise model fitting. For flexibility, SPIRE's rooflines use piecewise linear functions to map operational intensity (I_x)

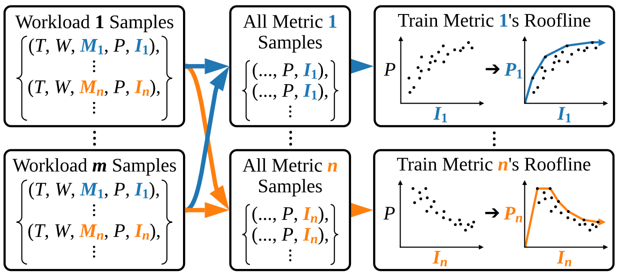

Fig. 3. SPIRE's ensemble-level training process. **Middle:** All training samples are grouped by performance metric. **Right:** Each group of samples is used to train an independent, metric-specific roofline model.

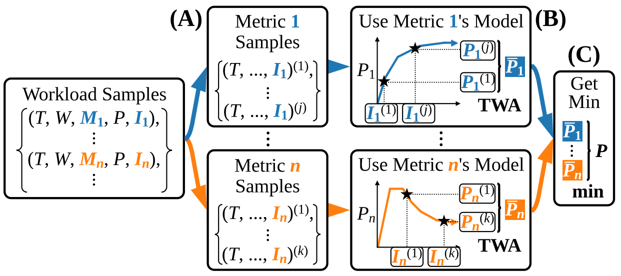

Fig. 4. SPIRE's ensemble-level estimation process. **A:** The workload's samples are grouped by performance metric. **B:** Each sample receives an estimation (${P_x}^{(i)}$) based on its operational intensity (${I_x}^{(i)}$). For each metric, these individual estimations are merged using a time-weighted average (TWA) based on each sample's period length (T). **C:** The workload's final max throughput estimation (P) is the minimum of the per-metric average estimations.

to max throughput (P). Since the rooflines estimate an upper bound, this function lies on, or above, all of its training samples. During training, the function is split into two regions, left and right, based on the training sample with the highest throughput.

To the left of this sample, we assume the metric is negatively associated with performance since the slope between the origin ($I_x = 0, P = 0$) and the sample is positive. Thus, we use line segments that produce an increasing, concave-down fit. In contrast, in the right region, we assume that the metric is positively associated with performance, and use a series of decreasing, concave-up segments. We present the algorithms for computing the specific line segments in III-D.

C. Ensemble Structure

A SPIRE model consists of an ensemble of roofline models. Each roofline is uniquely associated with one performance metric, which it uses for training and estimation.

Training. During training, all samples are grouped by their performance metric, and each group is used to create a distinct roofline model. For example, samples associated with a "stalls" metric would contain the value I_{stalls}. These samples would then create a roofline model that estimates max throughput based on I_{stalls}. Figure 3 illustrates the ensemble training process.

Throughput estimation. When estimating a workload's max throughput, its samples are grouped by their performance metric. Next, each sample receives a throughput estimation from its metric's roofline model. Then, for each metric, these per-sample estimations are merged by computing a time-weighted average based on the samples' period lengths (*i.e.*, T values), as shown in Eq. (1):

$$\bar{P}_x = \frac{\sum_i T^{(i)} P_x^{(i)}}{\sum_i T^{(i)}} \tag{1}$$

where $P_x^{(i)}$ is the estimate that metric x received for sample i, and $\bar{P}_x$ is the computed average for the metric. Finally, the minimum of these per-metric averages is the ensemble-wide throughput estimation. Figure 4 diagrams this process.

Performance analysis. Using a trained SPIRE model for performance analysis primarily consists of ranking metrics by their average estimations (*i.e.*, $\bar{P}_x$ values). Metrics with the lowest throughput estimations are more likely to be associated with performance bottlenecks. This is similar to how conventional roofline models operate: the lower of π or βI indicates that the workload is compute- or memory-bound, respectively.

While the minimum-valued metric is the primary bottleneck from the model's perspective, we suggest considering a range of low-valued metrics to all be potential bottlenecks. Factors such as measurement noise and imperfect modeling may cause some uncertainty in these values. Further, associations between metrics, such as causal and confounded relationships, can complicate subsequent testing and analyses. Proceeding with a pool of low-valued metrics can help account for estimation uncertainty and provide a more complete picture of performance.

D. Roofline Fitting Algorithms

Left fitting algorithm. We use a convex hull algorithm to choose line segments for a roofline's left region. This produces a series of segments that are increasing, concave-down, and lie on, or above, all training samples. Figure 5 demonstrates this approach using [12]'s simple convex hull algorithm.

Right fitting algorithm. A valid fit for the right region of a SPIRE roofline is a series of decreasing, concave-up line segments that lie on, or above, all training samples. Ideally, these will lie as close as possible to the training data.

Our algorithm starts by identifying a Pareto front of samples that maximizes throughput (P_x) and operational intensity (I_x) (*e.g.*, A-E in Figure 6). We ignore all other samples as they cannot be reached with a valid series of line segments. We then construct a weighted graph representing all valid line segments between Pareto samples. A vertex (X, Y) exists if a line can be drawn between samples X and Y without crossing below any other samples. An edge exists from (X, Y) to (Y, Z) if a line from Y to Z is steeper than the line between X and Y. This edge is weighted based on how closely the YZ line follows the Pareto samples (*i.e.*, its estimation error).

For example, in Figure 6, vertex (A, B) has an edge to (B, D) with a weight of 11. A line between samples B and D is steeper than one between A and B, so the concave-up property is maintained. The edge's weight is 11 because the BD line overestimates sample C with a squared error of 11.

The final elements of this graph are special $Start$ and End vertices. $Start$ represents starting the fitting process at a sample S with $I_x = \infty$ (*i.e.*, $M_x = 0$). If no such sample exists, a dummy S is temporarily added. The End vertex represents using a special horizontal line segment to reach the leftmost Pareto sample (E in Figure 6) from any other Pareto

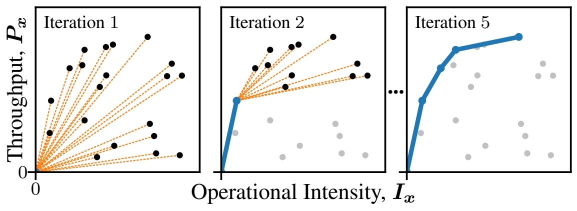

Fig. 5. An algorithm for fitting the left region of the SPIRE roofline model. Starting at the origin, (**Left**) compute slopes to all samples up and right of the current point. **Middle:** Add a line segment to the sample with the highest slope. Then, compute new slopes from this sample. **Right:** Repeat this process until the highest-throughput sample is reached.

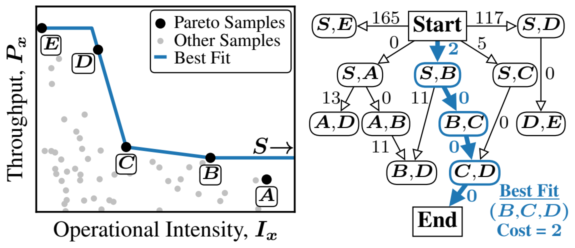

Fig. 6. An algorithm for fitting the right region of SPIRE's roofline models. **Left:** Fitting is performed based on the Pareto optimal samples (A-E). The best fit is shown. **Right:** The best fit is found by computing the shortest path from $Start$ to End on a weighted graph. Note that all vertices have an edge to End (most are omitted for clarity).

sample. All vertices have an edge to End weighted with the resulting segments' estimation errors. This minor exception to the concave-up rule allows for many more valid fits.

Finally, any path from $Start$ to End in the complete graph encodes a valid piecewise fit. Since edge weights represent estimation errors, we compute the best fit by finding the shortest path with Dijkstra's algorithm [6].

IV. Experimental Setup

To evaluate our methodology, we trained a SPIRE model to estimate a CPU's IPC using samples collected while running 23 high-performance computing (HPC) workloads. We then used this model to identify performance bottlenecks in four new HPC workloads. Finally, we compared the results of our SPIRE-based analysis with those of VTune [11], an industry-standard, counter-based performance analysis tool.

TABLE I
WORKLOADS USED TO EVALUATE SPIRE

Name	Configuration	Name	Configuration
Training Workloads			
numenta-nab	Relative Entropy	parboil	Stencil
qmcpack	O_ae_pyscf_UHF	onednn	IP Shapes 3D
remhos	Sample Remap	llamafile	wizardcoder-python
scikit-learn	SGDOneClassSVM	heffte	r2c, FFTW, F64, 256
mafft		scikit-learn	Feature Expansions
lammps	Model: 20k Atoms	npb	BT.C
graph500	Scale: 29	faiss	demo_sift1M
faiss	polysemous_sift1m	parboil	MRI Gridding
		openvino	Age Gen. Recog. F16
tensorflow-lite	Mobilenet Quant	arrayfire	BLAS CPU
openvino	Face Detect. F16-I8	scikit-learn	Random Projections
rodinia	CFD Solver	fftw	Stock, 1D FFT, 4096
Testing Workloads			
tnn	SqueezeNet v1.1	onnx	T5 Encoder, Std.
scikit-learn	Sparsify	parboil	CUTCP

Main TMA Bottleneck | Front-End | Bad Speculation | Memory | Core

TABLE II
Top 10 Performance Metrics for Each Testing Workload

TNN		Scikit		ONNX		Parboil	
2.11	F.E.	1.72	B.S.	0.24	Mem.	1.35	Core
0.77	FE.3	1.81	CS.2	0.04	VW	0.84	C1.3
1.03	DB.1	1.90	CS.6	0.23	L3	1.00	LK
1.62	DQ.2	1.92	DQ.K	0.23	L1.1	1.33	MS.1
1.64	FE.1	2.05	M	0.24	M	1.37	M
1.66	DB.3	2.07	C1.3	0.24	DQ.K	1.37	DQ.K
1.70	DQ.1	2.09	BP.2	0.24	CS.5	1.53	CS.3
1.71	DB.2	2.09	BP.1	0.24	CS.4	1.57	CS.5
1.74	DQ.C	2.10	DB.1	0.24	CS.1	1.57	MS.2
1.92	DQ.3	2.10	BP.3	0.25	L1.2	1.62	C1.1
1.94	FE.2	2.10	DB.4	0.25	L1.3	1.62	C1.2

Legend

Test Workload	
Measured IPC	Main TMA Bottleneck
Mean IPC Estimations	Metric Abbreviations

TABLE III
Performance Metric Abbreviations and Names, Organized by Microarchitecture Area

Abbr.	Expanded Metric Name	Abbr.	Expanded Metric Name
FE.1	frontend_retired.latency_ge_2_bubbles_ge_1	M	cycle_activity.cycles_mem_any
FE.2	frontend_retired.latency_ge_2_bubbles_ge_2	L1.1	cycle_activity.cycles_l1d_miss
FE.3	frontend_retired.latency_ge_2_bubbles_ge_3	L1.2	cycle_activity.stalls_l1d_miss
DB.1	idq.dsb_cycles	L1.3	l1d_pend_miss.pending_cycles
DB.2	idq.dsb_uops	L3	longest_lat_cache.miss
DB.3	frontend_retired.dsb_miss	LK	mem_inst_retired.lock_loads
DB.4	idq.all_dsb_cycles_any_uops	CS.1	cycle_activity.stalls_total
MS.1	idq.ms_switches	CS.2	uops_retired.stall_cycles
MS.2	idq.ms_dsb_cycles	CS.3	uops_issued.stall_cycles
DQ.1	idq_uops_not_delivered.cycles_le_1_uop_deliv.core	CS.4	uops_executed.stall_cycles
DQ.2	idq_uops_not_delivered.cycles_le_2_uop_deliv.core	CS.5	resource_stalls.any
DQ.3	idq_uops_not_delivered.cycles_le_3_uop_deliv.core	CS.6	exe_activity.exe_bound_0_ports
DQ.C	idq_uops_not_delivered.core	C1.1	uops_executed.core_cycles_ge_1
DQ.K	idq_uops_not_delivered.cycles_fe_was_ok	C1.2	uops_executed.cycles_ge_1_uop_exec
BP.1	br_misp_retired.all_branches	C1.3	exe_activity.1_ports_util
BP.2	int_misc.recovery_cycles	VW	uops_issued.vector_width_mismatch
BP.3	int_misc.recovery_cycles_any		

Closest TMA Bottleneck Front-End | Bad Speculation | Memory | Core

Baseline analysis tool. VTune's *Top-Down Microarchitecture Analysis* (TMA) [24] leverages hardware counters to characterize a workload, based on how it utilizes a CPU's execution capacity, using four top-level categories:

1) *Retiring*: performance used to do useful work
2) *Front-End Bound*: perf. lost to front-end stalls
3) *Bad Speculation*: perf. lost to incorrect speculation
4) *Back-End Bound*: perf. lost to back-end stalls

For example, a workload that loses 75% of a CPU's performance to back-end stalls is considered 75% "back-end bound" and 25% "retiring." These categories are further divided into trees of subcategories that provide more specific classifications. For instance, back-end bound splits into *memory bound* (memory-related stalls) and *core bound* (non-memory stalls).

Hardware. Our test system had two Intel Xeon Gold 6126 CPUs, 256 GiB of 2666 MHz DDR4 memory, and was running Ubuntu 20.04.6 LTS. To maximize performance consistency on this shared system, we ran all workloads using single-threaded execution (*i.e.*, using a single CPU core).

Workloads. All of our workloads came from the Phoronix Test Suite's (PTS) HPC benchmark suite [9]. We chose our 27 workloads because they exhibit a variety of bottlenecks on our test system. Table I lists the workloads and uses colors to indicate their main, high-level TMA bottleneck. The workloads used to test our SPIRE model are shown at the bottom and were the strongest examples of their respective TMA bottlenecks.

Sample collection. We used Linux Perf's [21] *stat* mode to access our CPU's hardware performance counters for sample collection. To measure work (W) and time (T), we used the *inst_retired.any* and *cpu_clk_unhalted.thread* counter values, respectively. For performance metrics (M_x), we used 424 different counter values. We ran each workload once to completion for up to 10 minutes. During execution, we collected a sample for each metric every two seconds via Perf's counter multiplexing. For our workloads, this sampling had an average execution time overhead of 1.6% (4.6% maximum). In total, we collected 1.3M samples (~3k per metric).

V. Experimental Results and Analysis

SPIRE analysis. Table II presents our SPIRE model's top 10 performance metrics for each test workload (*i.e.*, the metrics with the lowest average IPC estimations). We use 10 here to conserve space; more metrics may be inspected in practice. Table III defines our abbreviations and indicates which high-level TMA bottleneck is most closely related to each metric.

SPIRE's top metrics indicate a front-end bottleneck in *TNN*. Specifically, the *DB.n* metrics suggest that the bottleneck is related to the core's decoded stream buffer (DSB). VTune's TMA agrees with this, as it classifies *TNN* as 51% front-end bound and 46% retiring. It mainly attributed this classification to heavy use of the slower legacy decode pipeline; the faster DSB only provided 5.4% of micro-ops (µOps).

Scikit presents a mix of categories, but most are core or branch misprediction-related (*BP.n*). The core metrics indicate that the back-end is stalling the front-end (*DQ.K*) and back-end utilization is poor (stalls and 1 port utilized). Similarly, our TMA baseline found *Scikit* to be 35% branch misprediction-bound, 13% core-bound, and 41% retiring. The core bottleneck was associated with high divider use and periods of low execution port utilization.

ONNX's metrics suggest it is both memory and core-bound. Specifically, the L3 miss metric (*L3*) indicates a DRAM bottleneck. The core metrics imply slowdown from mixing SIMD vector widths (*VW*) and back-end stalling (*DQ.K, CS.n*). TMA classified *ONNX* as 82% memory-bound, 9.5% core-bound, and 6.7% retiring. 90% of this memory-boundedness was attributed to DRAM accesses, while the back-end spent most of the time stalled with no ports utilized. TMA showed that 256 and 512-bit SIMD ops were mixed, but this was not a major bottleneck.

The top metrics for *Parboil* imply that the back-end is often stalled (*DQ.K*, *CS.n*) or has 1 port utilized (*C1.n*). Memory metrics (*M*, *LK*) suggest that locked loads may be relevant, and the front-end *MS.n* metrics imply significant use of the microcode sequencer (MS). TMA found *Parboil* to be 40% core-bound, 12% memory-bound, and 43% retiring. The core-

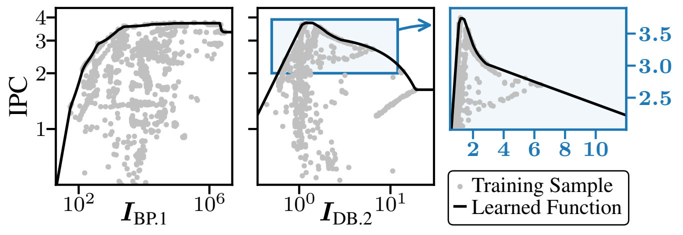

Fig. 7. Plots of two rooflines from our SPIRE ensemble along with their training samples. Note that the first two plots use log scales, which distort the models' line segments. **Left:** *BP.1*'s roofline (retired mispredicted branches). **Middle:** *DB.2*'s roofline (decoded stream buffer µOps). **Right:** Zoomed in view of *DB.2*'s plot on a non-distorting linear scale.

boundedness was associated with poor utilization of execution ports, while the memory category was attributed to lock latency.

In general, SPIRE's top metrics appear to accurately indicate performance bottlenecks. However, since they are statistically derived based on a specific set of workloads, follow-up analysis may be necessary. Even so, SPIRE can quickly reduce the number of metrics under analysis, thus speeding up the process of identifying true performance bottlenecks.

Learned roofline functions. Figure 7 shows two of our learned roofline models with their training samples. Note that the log scales visually distort the models' line segments. The *BP.1* model (left) demonstrates our left fitting algorithm while the *DB.2* model (middle, right) presents the right fitting one.

Overall, *BP.1*'s model accurately learned that branch mispredictions limit maximum IPC. The model's estimation increases as $I_{BP.1}$ (instructions per retired misprediction) grows. However, the right fitting algorithm kicked in for high $I_{BP.1}$ values and caused this estimation to drop, inaccurately. While this defect can be fixed with more training data, it shows that our method for detecting positive and negative metrics can be more robust.

The roofline for *DB.2* correctly learned that using the decoded stream buffer (DSB) is beneficial. As fewer µOps are served by the DSB, the model's IPC upper-bound decreases. However, the left, increasing side of the model implies that the DSB can harm performance. This trend, which follows the training samples, may be due to confounding factors that reduce both $I_{DB.2}$ (instructions per decoded µOp) and IPC. Misspeculated instructions, for example, may still decode µOps even though they never retire (reducing both $I_{DB.2}$ and IPC).

VI. Related Works

A. Roofline Models

Many extensions of the basic roofline model add ceilings to include aspects of performance such as bandwidth at different cache levels [10], throughputs of different floating-point operations [15], and throughputs under various types of parallelism [23]. In contrast, SPIRE automatically scales to incorporate more information through its ensemble structure. By having an independent roofline model for each metric, SPIRE can more accurately model how each influences performance. Further, this lets SPIRE scale to many performance metrics without overwhelming the user with many ceilings on one model.

Previous works have also used different values to measure throughput and to characterize workloads. In addition to FLOP/s, throughput has been measured with integer ops. per second [14], IPC [18], and GPU warp instructions per second [7]. Instead of conventional operational intensity, prior works have also used non-memory instructions per memory instruction [18] and memory latency [16]. In comparison, SPIRE's rooflines use flexible throughput and operational intensity definitions. Together, these let SPIRE easily generalize to different optimization goals, processors, and metrics.

Some works have developed more accurate roofline models by integrating more microarchitectural parameters (*e.g.*, sizes of OoO execution structures) [18] and more detailed workload analyses [4]. However, these are only applicable to certain classes of processors, losing the generality of the conventional roofline model. In contrast, SPIRE trains by observing a processor's performance counters; it uses no architecture-specific parameters or analyses.

B. Hardware Performance Counter Analysis

The methods used by counter-based performance analysis tools can vary widely and are inherently processor-specific. However, tools including Linux Perf [21], Intel VTune [11], and ARM Streamline [1] use a form of Top-Down Analysis (TMA) [24] for OoO CPUs. This approach organizes a CPU's counter data into a hierarchy to guide users from high-level bottlenecks (*e.g.*, "front-end bound") down to low-level causes (*e.g.*, instruction cache misses). While effective, TMA is specifically designed for OoO CPUs and can require significant effort to implement and tune. In contrast, SPIRE is architecture-agnostic and only requires infrastructure for sample collection.

Instead of relying on hand-designed strategies, like TMA, works such as [17] and [13] use machine learning to automate performance counter analysis. These two use standard ML algorithms (SGBRTs and linear regression, respectively) to identify which counter values are most important when predicting a processor's performance. However, this approach can lose useful causal information. For example, a standard algorithm may rely on a broad "frontend stall" count to predict performance while mostly ignoring values that indicate causes (*e.g.*, I-cache misses). In contrast, SPIRE's unique ensemble structure avoids losing causal information by fitting to each model input independently.

VII. Conclusions

In this work, we presented SPIRE, a new performance model that combines roofline models and hardware performance counters in a novel and effective way. SPIRE uses an ensemble of innovative roofline models to automatically learn a processor's performance characteristics using data from its hardware performance counters. In our evaluation, we used SPIRE to identify low-level performance bottlenecks on a real CPU. When compared with an industry-standard performance analysis tool, SPIRE's analysis accurately identified many of the same bottlenecks. Considering how little manual effort it required, SPIRE can be a valuable tool for processor research and development.

Our SPIRE implementation is available at https://github.com/spire-umich/spire-date25.

References

[1] *Arm Streamline User Guide*, Arm Holdings plc. [Online]. Available: https://developer.arm.com/documentation/101816/latest/

[2] *Cortex-A5 Technical Reference Manual*, Arm Holdings plc. [Online]. Available: https://developer.arm.com/documentation/ddi0433/latest/

[3] M. T. Bohr and I. A. Young, "CMOS Scaling Trends and Beyond," *IEEE Micro*, vol. 37, no. 6, pp. 20–29, 2017. [Online]. Available: https://doi.org/10.1109/MM.2017.4241347

[4] V. C. Cabezas and M. Püschel, "Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints," in *IEEE International Symposium on Workload Characterization*, 2014, pp. 222–231. [Online]. Available: https://doi.org/10.1109/IISWC.2014.6983061

[5] W. J. Dally, Y. Turakhia, and S. Han, "Domain-Specific Hardware Accelerators," *Communications of the ACM*, vol. 63, no. 7, pp. 48–57, 2020. [Online]. Available: https://doi.org/10.1145/3361682

[6] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959. [Online]. Available: https://doi.org/10.1007/BF01386390

[7] N. Ding, M. Awan, and S. Williams, "Instruction Roofline: An Insightful Visual Performance Model for GPUs," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 20, 2022. [Online]. Available: https://doi.org/10.1002/cpe.6591

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *International Symposium on Computer Architecture*, 2011, pp. 365–376. [Online]. Available: https://doi.org/10.1145/2000064.2000108

[9] "HPC - High Performance Computing." [Online]. Available: https://openbenchmarking.org/suite/pts/hpc

[10] A. Ilic, F. Pratas, and L. Sousa, "Cache-Aware Roofline Model: Upgrading the Loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014. [Online]. Available: https://doi.org/10.1109/L-CA.2013.6

[11] *Intel VTune Profiler User Guide*, Intel Corporation. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide

[12] R. A. Jarvis, "On the Identification of the Convex Hull of a Finite Set of Points in the Plane," *Information Processing Letters*, vol. 2, no. 1, pp. 18–21, 1973. [Online]. Available: https://doi.org/10.1016/0020-0190(73)90020-3

[13] A. Karami, S. A. Mirsoleimani, and F. Khunjush, "A Statistical Performance Prediction Model for OpenCL Kernels on NVIDIA GPUs," in *CSI International Symposium on Computer Architecture and Digital Systems*, 2013, pp. 15–22. [Online]. Available: https://doi.org/10.1109/CADS.2013.6714232

[14] J. Khemka and A. M. Shinsel, *Integer Roofline Modeling in Intel Advisor*, Intel Corporation. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/a-brief-overview-of-integer-roofline-modeling-in-advisor.html

[15] T. Koskela, Z. Matveev, C. Yang, A. Adedoyin, R. Belenov, P. Thierry, Z. Zhao, R. Gayatri, H. Shan, L. Oliker, J. Deslippe, R. Green, and S. Williams, "A Novel Multi-level Integrated Roofline Model Approach for Performance Characterization," in *High Performance Computing*, 2018, pp. 226–245. [Online]. Available: https://doi.org/10.1007/978-3-319-92040-5_12

[16] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera, "3DyRM: A Dynamic Roofline Model Including Memory Latency Information," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 696–708, 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1163-4

[17] Y. Lv, B. Sun, Q. Luo, J. Wang, Z. Yu, and X. Qian, "CounterMiner: Mining Big Performance Data from Hardware Counters," in *IEEE/ACM International Symposium on Microarchitecture*, 2018, pp. 613–626. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00056

[18] D. Marques, A. Ilic, and L. Sousa, "Mansard Roofline Model: Reinforcing the Accuracy of the Roofs," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 6, no. 2, 2021. [Online]. Available: https://doi.org/10.1145/3475866

[19] *Nsight Compute Documentation*, Nvidia Corporation. [Online]. Available: https://docs.nvidia.com/nsight-compute/

[20] G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Püschel, "Applying the Roofline Model," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 76–85. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844463

[21] "perf: Linux Profiling with Performance Counters." [Online]. Available: https://perf.wiki.kernel.org

[22] T. N. Theis and H.-S. P. Wong, "The End of Moore's Law: A New Beginning for Information Technology," *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017. [Online]. Available: https://doi.org/10.1109/MCSE.2017.29

[23] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[24] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2014, pp. 35–44. [Online]. Available: https://doi.org/10.1109/ISPASS.2014.6844459