Extending the Roofline Model: Bottleneck Analysis with Microarchitectural Constraints

Victoria Caparrós Cabezas
Department of Computer Science
ETH Zurich, Switzerland
caparrov@inf.ethz.ch

Markus Püschel
Department of Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

***Abstract*—Software, even if carefully optimized, rarely reaches the peak performance of a processor. Understanding which hardware resource is the bottleneck is difficult but important as it can help with both further optimizing the code or deciding which hardware component to upgrade for higher performance. If the bottleneck is the memory bandwidth, the roofline model provides a simple but instructive analysis and visualization. In this paper, we take the roofline analysis further by including additional performance-relevant hardware features such as latency, throughput, capacity information for a multilevel cache hierarchy and out-of-order execution buffers. Two key ideas underlie our analysis. First, we estimate performance based on a scheduling of the computation DAG on a high-level model of a microarchitecture and extract data including utilization of resources and overlaps from a cycle-by-cycle analysis of the schedule. Second, we show how to use this data to create only one plot with multiple rooflines that visualize performance bottlenecks. We validate our model against performance data obtained from a real system, and then apply our bottleneck analysis to a number of floating-point kernels to identify and interpret bottlenecks.**

I. Introduction

Software performance is determined by the extremely complex interaction of compiled code and the computing platform's microarchitecture. Programs rarely achieve peak performance, and microarchitectural features such as out-of-order execution, complex memory hierarchies, and various forms of parallel processing make it hard to pinpoint the reason. Understanding possible bottlenecks, however, is useful for both obtaining hints on how to optimize the code or how to upgrade a processor to improve the performance. The roofline model [1] identifies and, equally importantly, visualizes performance bottlenecks. Fig. 1 shows an example of a roofline plot. It depicts the performance of three applications as a function of their operational intensity (a more detailed introduction will be provided in Section II). This representation makes it possible to include bounds on performance due to both the computational throughput, π, and the memory bandwidth, β, of the platform, thus making explicit the notions of compute and memory bound. These bounds can be tightened by considering specific properties of an application such as the instruction mix or spatial locality.

The roofline model can clearly identify bottlenecks due to a throughput resource; for example memory bandwidth for app_A and computational throughput for app_C. However, the original model is inherently blind to other bottlenecks, in particular non-throughput resources including cache capacity, latency of memory accesses or the functional units, and out-of-order (OoO) execution buffers. As an example, for app_B the bottleneck is undetermined.

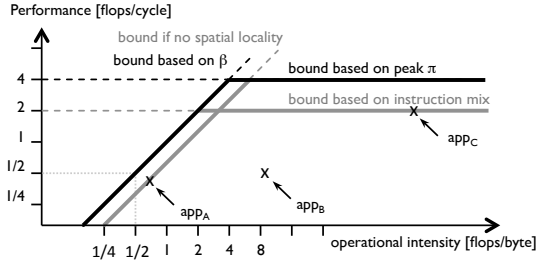


Fig. 1. Roofline plot for $\pi = 4$ and $\beta = 1$.

Contributions. The main contribution of this paper is an extension of the roofline model that provides a more detailed bottleneck analysis by considering a larger set of performance-relevant hardware parameters. In particular, this includes parameters not related to throughput, such as latency, capacity information for a multilevel cache hierarchy and out-of-order execution buffers. The result is a generalized roofline plot that simultaneously identifies associated constraints through multiple rooflines, thus identifying likely bottlenecks.

To achieve this goal we first present a novel DAG-based performance analysis technique. This approach schedules the computation DAG subject to the given hardware parameters and extracts the performance, utilization, and overlap data that is needed to create our generalized roofline plots (Sections III and IV). The design of the plots and the data they are based on is a main contribution of this paper. We validate our approach and compare our estimated performance against measured performance to show that the results are meaningful. In Section V we use our proposed analysis to perform a bottleneck study for a number of common numerical floating-point kernel functions that are relevant in different domains, such as signal processing or machine learning, and discuss the results. Finally, in Section VI we present related work, and conclude with a discussion of possible uses in Section VII.

II. Background

We present the background theory that underlies our performance bottleneck analysis. Specifically, we briefly discuss classical DAG-based performance analyses and present the previously introduced roofline model.

Throughout this work, we will use uppercase Roman letters to denote software properties, uppercase Greek letters to denote

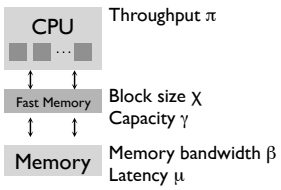


Fig. 2. Simple abstraction of a microarchitecture.

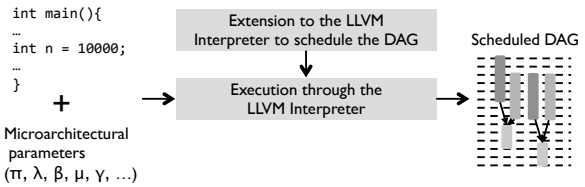


Fig. 3. Overview of our performance analysis.

software properties that depend on microarchitectural properties, and lowercase Greek letters to denote microarchitectural parameters.

A. DAG-based Performance Analysis

The dynamic executions of programs can be represented with directed acyclic graphs (DAGs), in which the nodes represent basic computations and the edges represent data dependences [2]. In this paper we assume mathematical programs in which the nodes are floating-point additions and multiplications. The DAG of a computation depends on its input size n. The total number of nodes in the DAG is referred to as work $W(n)$. The length of the longest dependence chain in the DAG is the *depth* or *span* $D(n)$. Computation DAGs can be used to provide performance estimates based on different models of computation. The PRAM (Parallel Random Access Machine) model [3], for example, assumes a CPU with parallel computational resources and an unbounded external memory. Basic computations (nodes in the DAG) are executed in unit time (we say cycle) and memory accesses are free. The external-memory model [4] extends it to include the I/O cost. It assumes a fast memory (cache) of size γ with a block size χ, and a slow memory with latency μ and bandwidth β, as shown in Fig. 2. These models provide coarse estimates and bounds on computation and communication time. However, due to their simplicity they usually fail to capture the real behavior of code executing on complex modern microarchitectures.

B. The Roofline Model

The roofline model [1] visualizes performance bottlenecks for a given program running on a processor arising from both its computational throughput π, and its memory bandwidth β. The program run on a given input is abstracted by its work W, and the memory traffic Q it causes, measured in bytes. The roofline model then plots on a log-log scale the measured performance against its operational intensity defined as

$$I = \frac{W}{Q}, \tag{1}$$

as shown in Fig. 1 for three examples (and associated inputs). If the computation time is larger than the memory time, i.e., $T_{\text{comp}} \geq T_{\text{mem}}$, the maximum performance P achievable by the application is bounded by the computational throughput π of the platform and the application is compute bound. If T_{mem} dominates (memory bound), then $T_{\text{mem}} \geq Q/\beta$ implies $P \leq I\beta$. Thus,

$$P \leq \min(I\beta, \pi), \tag{2}$$

which defines the roof (solid black lines) in Fig. 1. The horizontal bound can be refined (gray lines), e.g., if the program has imperfect balance of additions and multiplications or no instruction-level parallelism (ILP); the diagonal bound can be refined if the program has no spatial locality. Roofline plots can be created for every level of the memory hierarchy.

Roofline plots are a valuable tool in understanding bottlenecks, in particular for software with low operational intensity. However, most programs do not reach the bounds provided by the model, meaning that other bottlenecks exist. The goal of our work is to extend these plots to include a larger set of bottlenecks using a more detailed abstraction (meaning more parameters than those in Fig. 2) of the microarchitecture. The main challenge in achieving this goal are hardware resources that are not throughputs (e.g., latencies or cache capacities) to which the roofline model seems inherently limited.

III. DAG-based performance model

To design our generalized roofline model, we propose a novel DAG-based analysis that uses a more detailed abstraction of a microarchitecture; specifically, it will use a much larger set of parameters than those in Fig. 2 to capture a multi-level memory hierarchy and out-of-order execution.

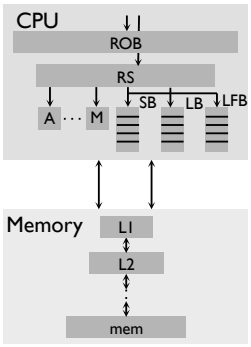
We implemented the model as a tool that first creates and schedules the computation DAG for a given program and input; then it extracts all the performance and utilization data needed for our generalized roofline model. In particular, our model needs data that requires a cycle-by-cycle analysis of the the computation DAG. This analysis cannot easily (or at all) be performed by measurement (on a processor) or by a simulator, thus our use of a DAG-based analysis.

Fig. 3 sketches the execution flow of our analysis tool. The input is an application source code with its input, and a set of parameters that describe microarchitectural resources. The code is then compiled to the LLVM intermediate representation (IR) [5], a virtual architecture that captures the key operations of traditional computing systems, but avoids machine-specific constraints such as physical registers, address calculation, or low-level calling conventions. The nodes of the computation DAG are the instructions of the LLVM IR that are either computations (additions or multiplications) or memory instructions (loads or stores). The IR of the application is then executed in the LLVM interpreter, which we extended to build a schedule of the nodes in the computation DAG. This schedule obeys the input microarchitectural constraints and yields a performance estimate without running the code on an actual processor. Further analysis of the DAG yields all the data needed for our generalized roofline model.

We now describe the main components of our approach in greater detail.

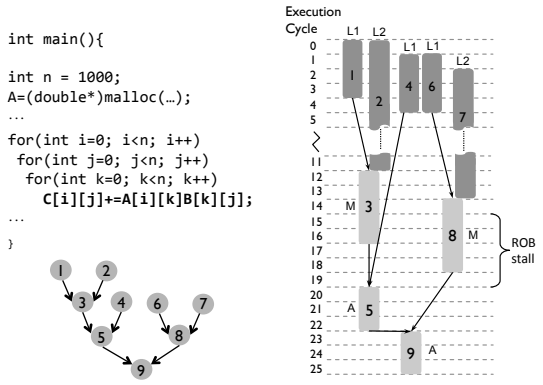
A. Microarchitecture Model

Our approach uses the abstraction of the microarchitecture shown in Fig. 4. We separate floating-point additions (A) and multiplications (M), and consider throughput and latency for both. Further, we consider a multi-level memory hierarchy



Microarchitectural Parameters	Parameter
Throughput	
Computation	π_A, π_M,
Memory bandwidth	β_{mem}, β_{L_i}
Instruction fetch bandwidth	φ
Latency	
Computation	λ_A, λ_M,
Memory	μ_{mem}, μ_{L_i}
Capacity	
Cache	γ_{L_i}
Cache block	χ
OoO Buffer	γ_{buf}

Fig. 4. Overview of our microarchitecture model and parameters that define it; $i = 1, 2, 3$ and buf $\in$ {ROB, RS, SB, LB, LFB}.



(a) Computation DAG. (b) Scheduled DAG.

Fig. 5. DAG scheduled according to the parameters in Table I.

(caches L_1, L_2, ..., and the main memory mem), and latency, bandwidth, and capacity for each. In this paper, we always use three levels of caches. Finally, we model the execution engine of the CPU via the instruction fetch bandwidth and five out-of-order (OoO) execution buffers: reorder buffer (ROB), reservation station (RS), load/store buffer (LB/SB) and line fill buffer (LFB) [6], [7]. Every parameter in the model can be set to zero or infinity to effectively remove the associated hardware feature.

B. Scheduling the DAG Based on a Microarchitecture Model

The computation DAG is scheduled by simulating its execution on a microarchitecture defined by the parameters listed in Fig. 4. The schedule is obtained using Tomasulo's greedy algorithm [8] for out-of-order execution, but extended to incorporate the additional OoO execution buffers mentioned above and also to obey the additional constraints on the memory reordering imposed by the platform's memory model. In this paper we only consider the x86 memory model [7]. Every level of the memory hierarchy is modeled as a fully-associative cache, and we use reuse distance analysis [9] to determine the level of the memory hierarchy to which an access hits.

C. Scheduled DAG: Properties and Performance Estimation

We use the scheduled DAG to extract the estimated performance and other data relevant for our bottleneck analysis (Section IV).

TABLE I. PARAMETERS TO MODEL AN INTEL XEON E5-2680; $i = 1, 2, 3$ AND BUF $\in$ {ROB, RS, SB, LB, LFB}.

Microarchitectural Parameters	Units	Value
Throughput		
π_{A}, π_{M}	flops/cycle	1,1
β_{mem}, β_{L_i}	doubles/cycle	1, 4, 4, 2
φ	instructions/cycle	4
Latency		
λ_{A}, λ_{M}	cycles	3, 5
μ_{mem}, μ_{L_i}	cycles	100, 4, 12, 30
Capacity		
γ_{L_i}	bytes	32K, 256K, 20M
χ	bytes	64
γ_{buf}	# slots	168, 54, 36, 64, 10

Example and node types. As an example, consider Fig. 5, which shows in (a) a piece of a larger computation multiplying two matrices, the corresponding computation DAG, and in (b) the result of scheduling the DAG with our tool using the microarchitecture model in Table I. The numbers in the nodes denote the order in the dynamic instruction trace. We assume all the loads hit the L1 and L2 caches as shown in the labels. The computation is delayed due to the load latency. Further, we assume prior computation has filled the ROB, so the execution of node 5 has to be delayed one cycle.

In general, the scheduled DAG has six different types of nodes: computations (additions and multiplications), and memory nodes (L1, L2, L3, and mem). For each of these, we define in Table II some general properties, e.g., the number of nodes of the corresponding type (N), or how many nodes can be scheduled at most in one execution cycle (Π). Note that the first two properties in Table II are determined by the table in Fig. 4; the others depend on the DAG. Depending on the type of node, a property such as throughput may have different symbols as shown in Table III.

Each of these properties is given or can be measured for each node type. We discuss their relationships next and then continue the example in Fig. 5.

Execution time of a node type. The total execution time of a node type x is a combination of the issue, latency, and stall times (see Table II), and depends on the scheduling of the DAG:

$$T_x = f(T_x^{\text{issue}}, T_x^{\text{lat}}, T_x^{\text{stall}}). \tag{3}$$

Cycles in T^{lat} and those in T^{issue}, by definition, will never overlap. However, stall and latency cycles as well as stall and issue cycles may overlap. For example, floating-point operations can be issued even if there is an LB stall; hence, in general, $T_x \neq T_x^{\text{issue}} + T_x^{\text{lat}} + T_x^{\text{stall}}$. The stall time T^{stall} can be further broken down into stalls due to the respective OoO buffers:

$$T^{\text{stall}} = g(T_{\text{RS}}^{\text{stall}}, T_{\text{ROB}}^{\text{stall}}, T_{\text{SB}}^{\text{stall}}, T_{\text{LB}}^{\text{stall}}, T_{\text{LFB}}^{\text{stall}}). \tag{4}$$

Again, g depends on the scheduling of the DAG.

Total execution time. The total execution time T of the scheduled DAG is a function of the execution times of all node types T_x. It is bound as

$$\max_x(T_x) \leq T \leq \sum_{\text{node type } x} T_x. \tag{5}$$

TABLE II. PROPERTIES OF NODE TYPE x IN THE SCHEDULED DAG.

Property	Description
Π	(Throughput) Maximum number of nodes issued per cycle
Λ	(Latency) Length of the node in cycles
N	Number of nodes in DAG
T^{issue}	Number of cycles (#cycles) in which nodes are issued
T^{lat}	#cycles in which nodes are executed, but not issued
T^{stall}	#cycles in which OoO buffers are full
T^{o}	#cycles in which execution overlaps with a different type
T_x	Total span of the node type x, including stall cycles

Overlap. Overlap cycles, T^o, can be defined for every set of node types. In this paper we only consider overlap of pairs of node types $\{x, y\}$. The total execution time of such a pair can then be expressed as a function of the individual times and the overlap cycles for this pair:

$$T_{\{x,y\}} = T_x + T_y - T^o_{\{x,y\}}, \tag{6}$$

where $T^o_{\{x,y\}} \leq \min(T_x, T_y)$. We define the overlap α as the following ratio:

$$\alpha = \alpha_{\{x,y\}} = \frac{T^o_{\{x,y\}}}{\min(T_x, T_y)}. \tag{7}$$

If $\alpha = 0$ there is no overlap and the total execution time is the sum $T_{\{x,y\}} = T_x + T_y$; if $\alpha = 1$, the overlap is maximal and hence $T_{\{x,y\}} = \max(T_x, T_y)$.

Performance estimation. The performance of the scheduled DAG is given as

$$P = \frac{W}{T}, \tag{8}$$

and is an estimate of the actual performance when run on a platform with the given microarchitectural parameters.

Continued example. In the scheduled DAG in Fig. 5(b) there are four types of nodes: Additions, multiplications, L1 and L2 accesses. $W = 4$ flops, $Q_{\text{L1}} = 3$ memory operations, and $Q_{\text{L2}} = 2$. According to the definition in Table II, $T^{\text{issue}}_{\text{L2}} = 2$ and $T^{\text{lat}}_{\text{L2}} = 12$ cycles. The span of the L2 nodes is 14 cycles but, since execution time includes the stall cycles, the total execution time for L2 nodes is $T_{\text{L2}} = 19$ cycles. For L1 accesses, $T^{\text{issue}}_{\text{L1}} = 2$ cycles, $T^{\text{lat}}_{\text{L1}} = 3$ cycles, and $T_{\text{L1}} = 10$ cycles. For both additions and multiplications, T^{issue} is 2 cycles, and the latency cycles are 2 and 4 cycles, respectively. $T^{\text{stall}} = T^{\text{stall}}_{\text{ROB}} = 5$ cycles for all the node types. Note that for multiplications, latency cycles and stall cycles overlap. $T_{\text{A}} = 11$ cycles, and $T_{\text{M}} = 8$ cycles. The total execution time of the scheduled DAG is $T = 26$ cycles, and the estimated performance is 0.15 flops/cycle.

Validation of the model. The proposed model estimates performance based exclusively on the nodes of the computation DAG and the set of 22 microarchitectural parameters that constrain the execution of the DAG. Many features including cache structure or branch predictors are not considered. To validate that despite its simplicity the model provides reasonable estimates, and to show possible limitations, we compare the performance measured from running applications on a real system against the performance estimated by (8) from the corresponding scheduled DAG.

As validation platform we use an Intel Xeon E5-2680 with a Sandy Bridge microarchitecture, characterized by the parameters in Table I, which are used for our model and analysis. We consider eight numerical kernels operating on double precision data that span the domains of signal processing (fast Fourier transform (FFT) from [10] and iterative and recursive Walsh-Hadamard transform (WHT) from [11]), linear algebra (a double loop matrix-vector multiplication (MVM), a triple loop and six-fold loop matrix-matrix multiplication (MMM)), scientific computing (3D 7-point stencil computation from [12]) and machine learning (a serial implementation of k-means clustering [13]). All the results shown correspond to a warm cache execution. Fig. 6 shows the comparison for these kernels. Each plot contains the measured performance of the code compiled with icc v13.1.3 with optimization flags -O3 -no-vec -no-simd[1], the measured performance of the code compiled with clang v3.4 and optimization flags -O3 -fno-vectorize -fno-slp-vectorize, and the performance estimated by the model. We include the performance of the icc-compiled code for reference, but the most meaningful comparison is between the latter two since the clang-compiled machine code is generated from the same IR used in the interpreter to generate the scheduled DAG.

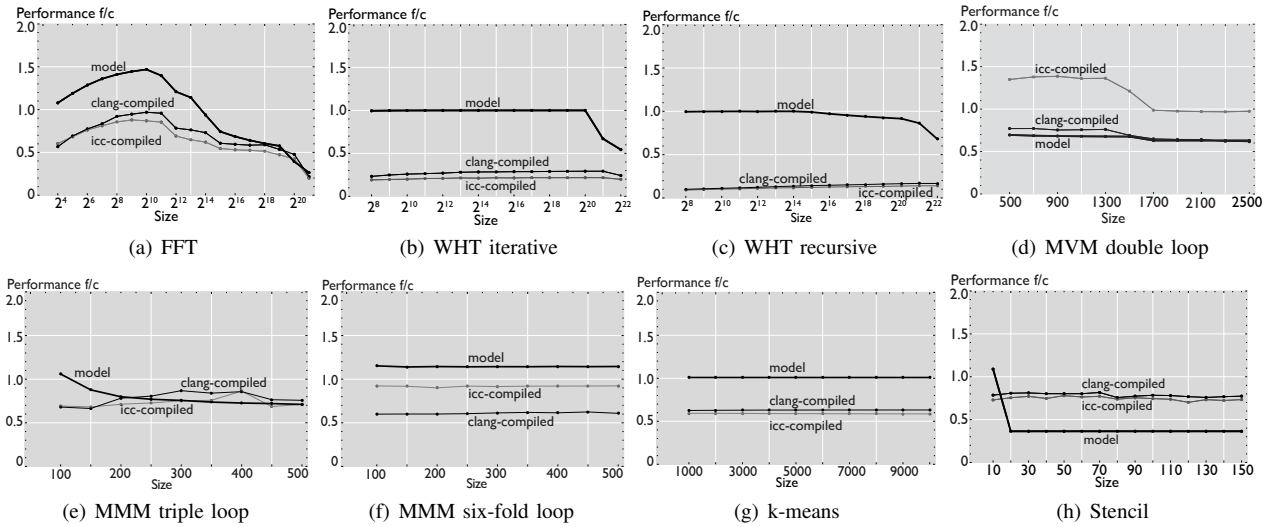
TABLE III. SPECIFIC NAMES OF THE PROPERTIES DEPENDING ON NODE TYPE.

Node type (x)	N # nodes	Π Throughput	Λ Latency	T_x Time
Computation	W	π		T_{comp}
A	W_{A}	π_{A}	λ_{A}	T_{A}
M	W_{M}	π_{M}	λ_{M}	T_{M}
Memory	Q			
L1	Q_{L1}	β_{L1}	μ_{L1}	T_{L1}
L2	Q_{L2}	β_{L2}	μ_{L2}	T_{L2}
L3	Q_{L3}	β_{L3}	μ_{L3}	T_{L3}
mem	Q_{mem}	β_{mem}	μ_{mem}	T_{mem}

For the FFT, MVM and MMM triple loop, the proposed model accurately estimates performance and performance trends. For the other computations, however, the difference is more significant. In the case of WHT, for example, the likely reason is the memory access pattern, which uses large power-of-two strides and thus suffers from conflict misses, which are not modeled by our approach. Hence in this case the estimated performance is higher than the measured performance. The performance can also be overestimated because we ignore all address calculations and control instructions, which may decrease performance for control-intensive applications like k-means. For the stencil computation, the predicted performance is lower than the measured. The likely reason is that it benefits from hardware stream prefetchers, which are not yet included in our model.

Note that none of the applications reaches the maximal possible 2 flops/cycle on this platform. The question is which hardware resources are responsible. In the following we present a methodology to provide insight into why this is the case. For example, we will see later in Fig. 10 that likely bottlenecks for the FFT of sizes 2^{10} and 2^{20} in Fig. 6(a) are the combination of L1 and computation latencies, and ROB stalls, respectively.

[1]We do not yet model SIMD vector architectures



(a) FFT (b) WHT iterative (c) WHT recursive (d) MVM double loop

(e) MMM triple loop (f) MMM six-fold loop (g) k-means (h) Stencil

Fig. 6. Comparison of performance estimated by the model and performance on a real system.

IV. Bottleneck Modeling and Generalized Roofline Plots

In this section we present the main contribution of this paper: We use the hardware platform properties (Fig. 4) and the performance data (Table III) obtained in the DAG analysis for modeling bottlenecks. The result is a generalization of the roofline model from Section II-B. As explained there, the original proposal of roofline plots [1] only captures bottlenecks due to throughput resources, and the performance bounds are tight only if the corresponding resource is fully utilized *and* computation and communication perfectly overlap. To illustrate these shortcomings, Figs. 7(a) and (b) show the roofline plots for the computation in Fig. 5(b) for both L1 and L2 cache. In both cases, the peak is not reached. As mentioned before, one of the reasons is that computation and L2 memory nodes only overlap in cycles 12–13, and in the ROB stall cycles, which are included in both types of nodes. Another reason is that the throughput resources are not fully utilized. Although in every cycle one multiplication and one addition can be executed, in cycles 12, 14, 20 and 23, only one operation is issued. Similarly, in cycles 21–22 and 24–25 no instructions are issued because of the latency of the floating-point additions, and in cycles 15–19, the execution is stalled due to the ROB capacity. In this section we present a model that turns these inefficiencies into performance bottlenecks that can be included in a roofline plot.

For a concise visualization, we first explain how to merge roofline plots associated with different levels of the memory hierarchy into one plot.

A. Merging Roofline Plots

In the original roofline model, the notion of operational intensity is specific to a chosen level of the memory hierarchy. To consider different levels simultaneously, we redefine operational intensity as flops per byte transferred to *all* levels of the memory hierarchy:

$$I = \frac{W}{Q} = \frac{W}{Q_{\text{L1}} + Q_{\text{L2}} + Q_{\text{L3}} + Q_{\text{mem}}}. \tag{9}$$

The operational intensity for a given level, say x, can then be computed as $I_x = IQ/Q_x$.

Note that with this definition, every access contributes to exactly one of the four summands in (9) and is thus counted exactly once.

The performance bound obtained for a specific level of the hierarchy, given by (2), now becomes:

$$P \leq \min(I_x \beta_x, \pi) = \min(I \frac{Q}{Q_x} \beta_x, \pi). \tag{10}$$

Using the redefined operational intensity and (10) we now create one roofline plot for all levels of the memory hierarchy. Fig. 7(c) shows the result for our running example. Note that there is one fundamental difference to the original roofline plot. Because of the factor Q/Q_x, the memory bounds in Fig. 7(c) now depend on program and input.

We note that the prior merging approach in [14] uses (9) but not (10). As a result the bounds are program independent but are only valid if all accesses hit one level of the hierarchy.

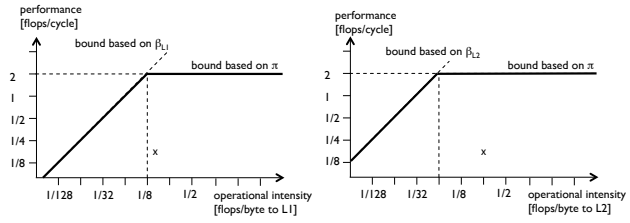
Note that at this point we can create roofline plots whose roofs correspond one-to-one to the types of nodes in our DAG (see Table III): compute nodes[2] yield horizontal roofs, memory nodes yield diagonal roofs. In the following, these roofs are always drawn as solid lines.

B. Modeling Bottlenecks From the Scheduled DAG

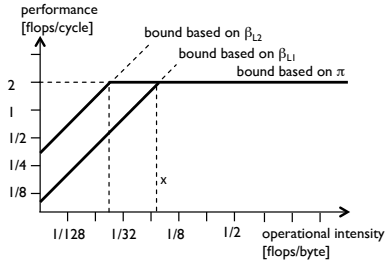
Our approach to modeling bottlenecks is based on one observation: In the standard roofline model, if a program is, e.g., L1-bound, it will hit the performance roof from the L1 bandwidth if and only if the bandwidth is 100% utilized throughout the program. In general, this is not the case as the L1 cache is utilized only in part of the program. By quantifying this utilization we create additional and tighter roofs to identify potential bottlenecks. They are drawn as dashed lines in the plots. In the following, we first quantify utilization and then derive these additional bottlenecks.

Utilization. The utilization of the throughput of a node type is defined as the ratio of the number of the nodes executed to

[2]For simplicity, additions and multiplications are from now on collected in a single *computation* node type. The total number of nodes for this type is $W = W_{\text{A}} + W_{\text{B}}$, and the throughput is $\pi = \pi_{\text{A}} + \pi_{\text{B}}$.



(a) Roofline plot for L1. (b) Roofline plot for L2.



(c) Roofline plot for L1 and L2 cache.

Fig. 7. Merging roofline plots of different levels of the memory hierarchy: Example for the code in Fig. 5(b).

the number of nodes that could have been executed at full throughput:

$$U = \frac{N}{T\Pi}, \tag{11}$$

where N, Π and T are defined in Table III. For example, the utilization of the L1 node type is $U_{\text{L1}} = Q_{\text{L1}}/(T_{\text{L1}}\beta_{\text{L1}})$.

Once the utilization U_x for a node type x is computed, the roof associated with x can be multiplied by U_x, which effectively makes the bound tighter. However, we do not do this since this definition of utilization (if suboptimal) does not distinguish between issue bottlenecks and stalls due to latencies or reorder buffers. To separate these into more specific bottlenecks, and thus create even tighter roofs, we define more specific forms of utilization in the following. Each will yield a roofline parallel to the original one. After that, we explain how the inclusion of overlap information may yield even tighter rooflines.

Issue bottlenecks. The issue bottleneck quantifies the throughput utilization considering only the issue cycles (see Table II):

$$U^{\text{issue}} = \frac{N}{T^{\text{issue}}\Pi}. \tag{12}$$

Issue bottlenecks show an inherent lack of ILP, or that data dependences prevent the program to run at full throughput.

For the scheduled DAG in Fig. 5(b), none of the throughput resources has an issue utilization of 1. For the L1 nodes, out of the 4 memory operations that could have been executed, only 3 are executed ($T^{\text{issue}}_{\text{L1}} = 2$, $\beta_{\text{L1}} = 2$, $Q_{\text{L1}} = 3$). The utilization is hence $U^{\text{issue}}_{\text{L1}} = 3/4$. A similar analysis for the computation nodes yields $U^{\text{issue}}_{\text{Comp}} = 1/2$, and for L2 nodes, $U^{\text{issue}}_{\text{L2}} = 1/4$.

Fig. 8 shows the extension of the roofline plot of Fig. 7, which now includes the three (compute, L1, L2) additional roofs due to issue bottlenecks. Note how the computation bound is lowered to 1 flop/cycle. In this case, the reason is not an inherent lack of ILP, but is a consequence of the scheduling of the DAG with the given microarchitectural constraints.

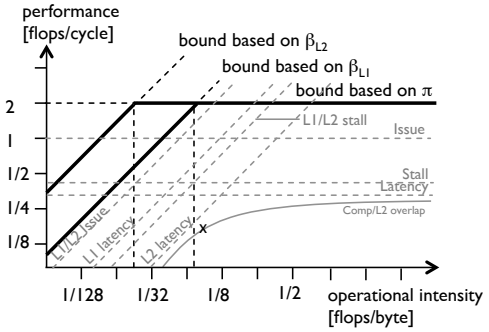


Fig. 8. Roofline plot for the scheduled DAG in Fig. 5(b) with additional bottlenecks.

The stall, latency, and overlap bottlenecks in Fig. 8 are explained in the following.

Latency bottlenecks. The latency bottleneck quantifies the performance loss due to latency cycles, i.e., cycles in which instructions are not being issued because of dependences with long latency operations and lack of ILP to hide its effect. This bottleneck is calculated by modifying (12) to include latency cycles:

$$U^{\text{lat}} = \frac{N}{(T^{\text{issue}} + T^{\text{lat}})\Pi}. \tag{13}$$

If the latency of a node type is 1, $T^{\text{lat}} = 0$, and $U^{\text{lat}} = U^{\text{issue}}$.

In Fig. 8, all three roofs (compute, L1, L2) have an associated tighter latency bottleneck. The latency utilizations are $U^{\text{lat}}_{\text{L1}} = 3/10$, $U^{\text{lat}}_{\text{L2}} = 1/28$, and $U^{\text{lat}}_{\text{Comp}} = 2/13$. This means that the achievable performance is 15% of the issue performance bound. Note that in this case, if latency cycles were removed, the maximum attainable performance would still be only 50% of the peak because of the issue bottleneck.

Stall bottleneck. The stall bottleneck quantifies loss of performance due to stall cycles caused by filled OoO buffers. Note that in this definition we have to take into account overlap with issue cycles to avoid counting cycles twice:

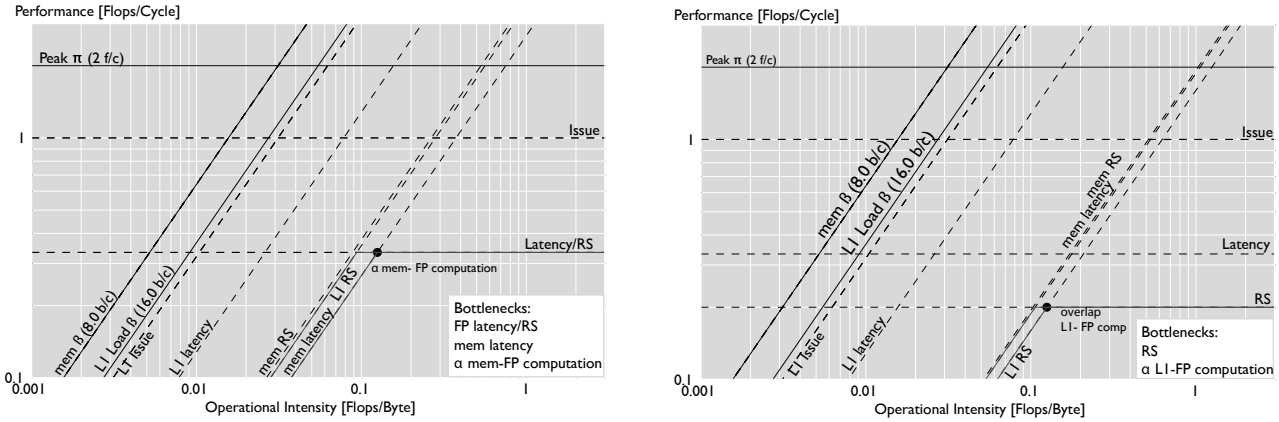
$$U^{\text{stall}} = \frac{N}{(T^{\text{issue}} + T^{\text{stall}} - T^{o}_{\{\text{issue,stall}\}})\Pi}. \tag{14}$$

As shown in (4), the stall cycles combine all the cycles spent in the OoO buffers. We actually do a more fine-grained stall bottleneck analysis by considering individual stall times and utilizations.

Fig. 8 shows the stall bottlenecks (only ROB stalls occur) for each roof. Note that for the computation nodes, latency and stall cycles overlap. This means that removing the stall cycles may not remove the bottleneck because the latency penalty could remain. This and other limitations of the proposed bottleneck analysis will be discussed at the end of this section.

Overlap bottleneck. The maximum performance cannot be reached if compute and memory nodes do not overlap completely in time. The overlap bottleneck quantifies the loss of performance due to imperfect overlap. In contrast to the previous bottlenecks, it is defined for pairs of node types including pairs of memory nodes. For a compute node and a memory node x with overlap α from (7), the associated performance roof is

$$P \leq \frac{IQ/Q_x}{(\frac{IQ}{U^{\text{Comp}}\pi Q_x} + \frac{1}{U^x\beta} - \alpha\min(\frac{1}{U^x\beta_x}, \frac{IQ}{U^{\text{Comp}}\pi Q_x}))}. \tag{15}$$



(a) Original parameters in Table I.

(b) Decrease γ_{RS} to 20.

Fig. 9. Generalized roofline plot for an iterative sum reduction, $N = 5 \times 10^6$, cold cache scenario.

If $\alpha = 1$, (15) is equivalent to (10), except that utilization is already taken into account. If $\alpha \neq 1$, then (15) yields a curved roof as shown in Fig. 8 for our running example. This roof now is tight, i.e., hits the performance point, which shows that the main bottleneck is the lack of overlap between computation and accesses to L2 (as one can confirm by inspecting Fig. 5(b)).

Second, the overlap bound for two memory type nodes x and y, where x has smaller execution time is given by

$$P \leq \frac{U^x \beta_x U^y \beta_y IQ}{U^x \beta_y (1-\alpha) Q_y + U^y \beta_y Q_y}. \tag{16}$$

C. Bottleneck properties and limitations of the analysis

As already mentioned, the various roofs that we derive become specific to program and input. Thus, each such plot can contain only one performance point. However, the major benefit is the integration of various roofs into only one plot that shows all bottlenecks.

Another limitation is that all bottleneck lines are interdependent. Modifying only one microarchitectural parameter implies a rescheduling of the entire computation DAG, and the rooflines may change unpredictably. There is no guarantee that a roof that hits the performance point is actually a bottleneck (the code could fully use the resource precisely).

We are working on the integration of SIMD vector instructions, which is relatively straightforward, but support for parallel code requires more research.

Finally, we want to emphasize that our notion of utilization enables the handling of code with different phases (e.g., parts dominated by memory operations, parts by computation). The distance of roofs to the performance point are an indicator of the relevance of the respective microarchitectural parameter.

V. Results

We apply our bottleneck analysis to a number of numerical kernels: first to an iterative vector sum reduction for basic validation, then to five of the kernels already used in Section III-C.

Experimental setup. Table I shows the microarchitectural parameters that we use in all our experiments; it models a recent Intel Xeon. In the roofline plots shown, the original performance bounds are shown as solid lines, our added issue, latency and stall bottlenecks as dashed lines, and overlap bottlenecks as gray solid lines. Due to the high (≥ 0.95) measured overlaps, the latter loose their curved shape.

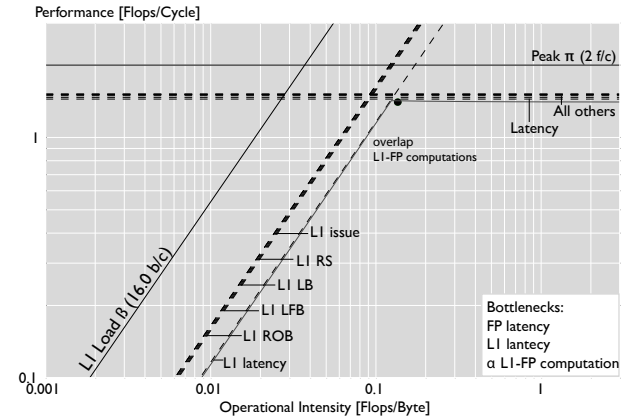
Basic validation: Reduction. Fig. 9(a) shows our generalized roofline plot for a large iterative sum reduction of a vector of doubles, run with cold cache. The operational intensity is $1/8$ (1 addition per 1 double). The performance is $1/\lambda_A$ due to the sequential dependence of the reduction. As expected, the plot identifies the floating-point latency as a bottleneck. However, the performance point hits two additional bottleneck lines, the memory bandwidth and the RS stall bottleneck. The first can be expected in a cold cache scenario, but the RS stall bottleneck may not be so obvious. The reason is that the instruction fetch rate (see Table I) is higher than the instruction execution rate, so the RS fills and becomes a bottleneck.

To show that the RS can indeed be a bottleneck, we reschedule the DAG by further reducing the RS size from 54 to 20 and obtain Fig. 9(b): the performance drops by 40% and the RS is the only bottleneck.

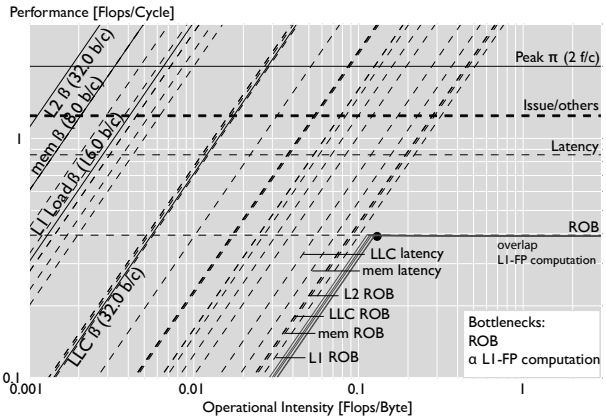
The reader may wonder why the performance point ends up on an intersection in both cases. In the original roofline plot this can happen only for $T_{\text{comp}} = T_{\text{mem}}$, and for the same reason it happens here (approximately). Namely, if buffer stalls and latencies are included, most cycles contribute to both T_{comp} and T_{mem} and thus they are roughly equal. What our analysis does is to remove from this count, in steps, cycles due to latencies and due to stalls. This tightens the roofs until the performance point is reached. If T_{comp} and T_{mem} have low overlap, then the performance will sit on the (curved) overlap line (e.g., Fig. 8).

Increasing size. Bottlenecks may also change with the input size. Fig. 10 shows our analysis for FFTs of sizes 2^{10} and 2^{20}. For the small size, L1 latency and bandwidth is the limiting resource and the peak is not reached because of latency effects. For larger sizes, more bottleneck lines appear because the application accesses more levels of the memory hierarchy, and new execution stalls due to OoO buffers appear, being the ROB stalls the one that contribute the most to execution time. Actually, all the possible bottleneck lines appear, which means that all buffers create execution stalls.

Different implementations of the same application. We compare how bottlenecks change across different implementations of matrix-matrix multiplication (MMM) of square matrices of size 500. Fig. 11 shows the generalized roofline plots for (a) a triple loop implementation and (b) a six-fold loop version, which is known to have better locality and, hence, better

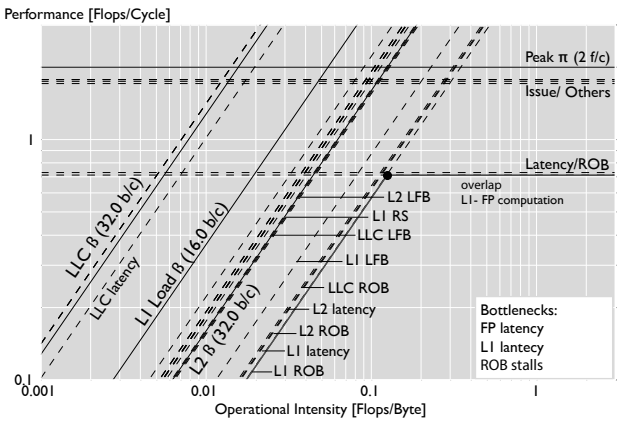


(a) Roofline plot for FFT, size 1024, warm cache.

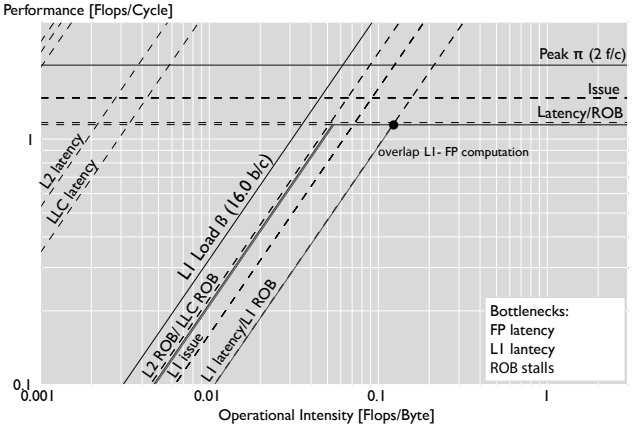


(b) Roofline plot for FFT, size 1048576, warm cache.

Fig. 10. Generalized roofline plot for FFT, warm cache.



(a) MMM triple loop.



(b) MMM six-fold loops, block size 50 (chosen to fit in the L1 cache)

Fig. 11. Generalized roofline plot for different implementations of MMM of size 500, warm cache.

performance as shown in the figure. In both implementations, execution stall cycles due to the ROB occupancy and floating-point latency cycles are important contributors to the execution time, and the associated bottleneck lines hit the performance point. While in the blocked implementation only the L1 latency appears as a bottleneck, in the triple loop implementation, the L2 latency limits performance as much as the L1 latency. In none of the cases the memory-related (mem) bottlenecks show up because the three matrices fit within the last-level cache. Note that although blocking is an optimization that targets improving locality, it also reduces the floating-point latency bottleneck.

Other applications: Stencil and k-means. Finally, we analyze the bottlenecks of a 3D 7-point stencil computation and k-means in Figs. 12 and 13, respectively. The stencil is calculated on a grid of 10^4 points, across 5 sweeps. Due to its access pattern, which exhibits poor spatial and temporal locality, the memory bandwidth is one of the main bottlenecks. Associated to these long-latency memory accesses, ROB stalls appear as an important bottleneck. In the case of the k-means kernel, again it is the latency of the floating-point computations and L1 accesses the main inhibitors of performance. It is interesting to see that across most of the kernels analyzed, these two hardware resources appear as the main bottlenecks, which suggests that improving them could have a direct impact on the performance of the applications.

Analysis time. In all benchmarks, the time for the analysis was about a factor 10^4 slower than the non-interpreted execution of the benchmark. Without the overlap analysis, the analysis time is about 5 times faster.

VI. Related Work

We give a brief overview of relevant work on performance and bottleneck analysis techniques.

Performance modeling and evaluation. Performance analysis techniques range from analytic models of computation to sophisticated software tools that measure performance on modern computing systems. In Section II we discussed basic analytic models of performance including work-span [15], and the external memory model used in the balance principles of computation [16]. There exist also semi-empirical and empirical modeling techniques that predict asymptotically tight bounds in performance by curve fitting [17] or machine learning techniques [18]. These models can also be used to predict hardware trends [19]. Our approach is analytic in that it quantifies performance properties from a (scheduled) DAG and does not include any measurements on an actual platform.

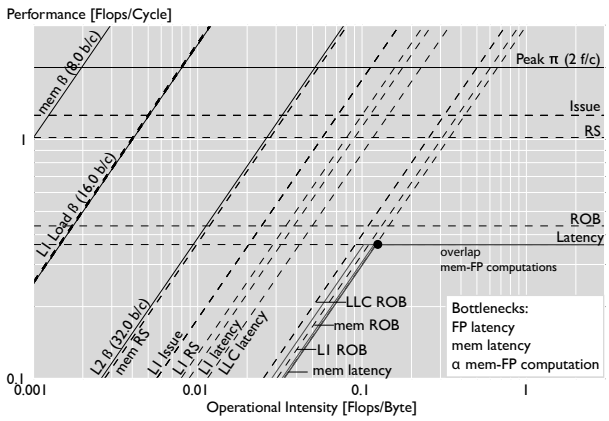


Fig. 12. Generalized roofline plot for 3D 7-point stencil, 10^4 elements.

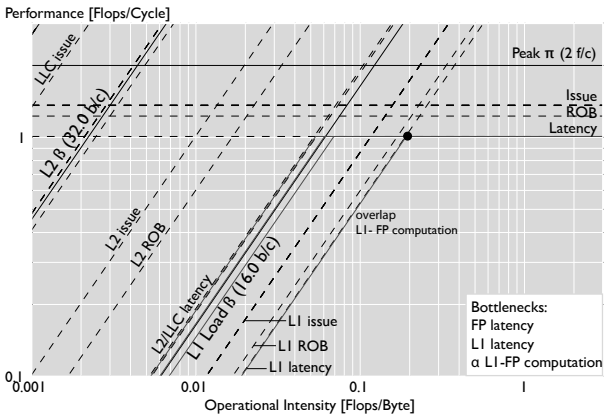


Fig. 13. Generalized roofline plot for k-means on 10^4 points, 1000 clusters.

In contrast to prior work many more hardware features are included in the analysis to obtain more realistic results.

On the other end of the spectrum of performance evaluation techniques are cycle-accurate microarchitectural simulators [20]. They model and mimic all the components of a microarchitecture, and report accurate performance predictions. Finally, performance and other properties can be measured using the performance counter infrastructure available in most modern computing systems. Tools like VTune [21] or PAPI [22] provide interfaces to access performance counters and report extensive statistics including cache misses or resource stalls cycles. As we argue next, our kind of bottleneck analysis cannot be easily be performed with either simulators or performance counters.

Bottleneck modeling and evaluation. VTune includes specific analyses to extract bottleneck information from the data collected from the performance counters [23]. These analyses classify and break down execution stalls until they identify a unique bottleneck. However, the bottleneck is specific to the processor the code is run on, the data does not take into account program phases with different characteristics, and does not provide overlap information. For this reason, it cannot be used to generate our roofline plots.

Other techniques to analyze bottlenecks consist of obtaining CPI stacks [24], or using dynamic binary instrumentation [25].

Cycle-accurate simulators can also provide bottleneck information by progressively modifying hardware parameters. Again they consider a completely specified microarchitecture, and the measurements do not extract program phases nor overlap information. Thus, our roofline plots cannot be generated with that data. Also, the simulation time may be considerably larger than the one needed by our tool.

In contrast to cycle-accurate simulators, our work also simulates the code but on a much higher level model that consists of a set of microarchitectural parameters that is common to many computing systems but is independent of any specific one. This may reduce accuracy but allows for faster design space exploration of platforms including those for which microarchitectural details are unknown. Moreover, our cycle-by-cycle analysis and utilization-based methodology can identify bottlenecks with a single execution of the application. We also emphasize again that our analysis is designed to produce the data needed to produce the generalized roofline plots that we introduced.

Roofline model. As mentioned at the end of Section IV-A, [14] merges roofs for different levels of the memory hierarchy but does not include roofs for non-throughput resources. [26] proposed a roofline model of energy but is not closely related to our work.

VII. Conclusions

In this paper we proposed an extension of the roofline model that includes an extended set of hardware-related bottlenecks including throughput, latency, and capacity information of a multi-level memory hierarchy and out-of-order execution buffers. Associated with the model we presented a novel generalization of the roofline plot that integrates all derived bottlenecks as bounds into one viewgraph that shows their relative importance.

At the core of our approach is a detailed DAG-based analysis done by a tool that we implemented. The tool, in a sense, simulates the execution of the DAG on a high-level model of a microarchitecture specified by a set of parameters relevant for the above bottlenecks. The key idea behind our tool is a cycle-by-cycle analysis to extract utilization and overlap data that is needed to create generalized roofline plots. This data is hard to extract from alternative techniques like cycle-accurate simulators or performance counters, which motivates our use of DAG-based analysis.

We validated our performance modeling technique against actual executions on a real system and showed reasonable results. Then we applied our bottleneck analysis to a number of important numerical kernel routines showing that it provides and visualizes tight bounds on performance, where execution cycles are spent, how they are distributed across resources, and their contribution to the application's performance. Although throughout the paper we used the modern Intel Sandy Bridge platform to both validate our models and experiments, other microarchitectures, for example the in-order Atom microprocessor, can be model with minor changes in the configuration parameters.

Possible uses of our work are in guiding manual code optimization or in deciding which processor components to upgrade to improve performance. The main current limitations

of our approach include not modeling prefetchers, and vectorized and threaded code. Also, the roofline plots are specific to the program input. Addressing these issues is future work.

In this paper we focused on developing the theory and approach; thus we could only provide a cursory experimental evaluation on a few kernels to show that we get meaningful results. However, even this simple evaluation and our visualization provided interesting insights into the interplay between different microarchitectural components and the associated bottlenecks. We believe we have only scratched the surface in developing our approach and interpreting the results.

The source code of the LLVM-based tool to obtain the performance bottleneck data is available at [27].

ACKNOWLEDGMENTS

The authors would like to thank Phillip Stanley-Marbell for his valuable feedback and helpful discussions.

REFERENCES

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, 2009.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Introduction to algorithms," 2009.

[3] W. J. Savitch and M. J. Stimson, "Time bounded random access machines with parallel processing," *Journal of the ACM*, vol. 26, no. 1, pp. 103–118, 1979.

[4] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Commun. ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[5] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Code Generation and Optimization (CGO)*, 2004.

[6] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann Publishers Inc., 2006.

[7] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2013.

[8] R. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, 1967.

[9] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Programming Language Design and Implementation (PLDI)*, 2003, pp. 245–257.

[10] W. H. Press, B. P. Flannery, T. S. A., and V. W. T., *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed., 1992.

[11] J. Johnson and M. Püschel, "In search of the optimal Walsh-Hadamard transform," in *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, 2000, pp. 3347–3350.

[12] "S. Kamil, Stencil probe, 2012," http://people.csail.mit.edu/skamil/projects/stencilprobe/.

[13] R. M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system," in *International Symposium on Workload Characterization (IISWC)*, 2009, pp. 198–207.

[14] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 99, 2013.

[15] G. E. Blelloch, "Programming parallel algorithms," *Commun. ACM*, vol. 39, pp. 85–97, 1996.

[16] K. Czechowski, C. Battaglino, C. McClanahan, A. Chandramowlishwaran, and R. Vuduc, "Balance principles for algorithm-architecture co-design," in *Hot topic in parallelism (HotPar)*, 2011, pp. 9–9.

[17] T. Hoefler, W. Gropp, W. Kramer, and M. Snir, "Performance modeling for systematic performance tuning," in *Supercomputing (SC)*, 2011, pp. 6:1–6:12.

[18] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz, "A regression-based approach to scalability prediction," in *International Conference on Supercomputing (ICS)*, 2008.

[19] K. Czechowski and R. Vuduc, "A theoretical framework for algorithm-architecture co-design," *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 791–802, 2013.

[20] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, pp. 50–58, 2002.

[21] "Intel®VTune™Amplifier XE 2013."

[22] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[23] A. Yasin, "A top down method for performance analysis and counters architecture," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

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

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Programming Languages Design and Implementation (PLDI)*, 2005, pp. 190–200.

[26] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 661–672, 2013.

[27] "Source code for the tool," http://www.spiral.net/software/extended-roofline.html.