PCCS: Processor-Centric Contention-aware Slowdown Model for Heterogeneous System-on-Chips

Yuanchao Xu
North Carolina State University
Raleigh, North Carolina, USA
yxu47@ncsu.edu

Mehmet E. Belviranli
Colorado School of Mines
Golden, Colorado, USA
belviranli@mines.edu

Xipeng Shen
North Carolina State University
Raleigh, North Carolina, USA
xshen5@ncsu.edu

Jeffrey Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
vetter@ornl.gov

ABSTRACT

Many slowdown models have been proposed to characterize memory interference of workloads co-running on heterogeneous System-on-Chips (SoCs). But they are mostly for post-silicon usage. How to effectively consider memory interference in the SoC design stage remains an open problem. This paper presents a new approach to this problem, consisting of a novel *processor-centric slowdown modeling* methodology and a new *three-region interference-conscious slowdown model*. The modeling process needs no measurement of co-running of various combinations of applications, but the produced slowdown models can be used to estimate the co-run slowdowns of arbitrary workloads on various SoC designs that embed a newer generation of accelerators, such as deep learning accelerators (DLA), in addition to CPUs and GPUs. The new method reduces average prediction errors of the state-of-art model from 30.3% to 8.7% on GPU, from 13.4% to 3.7% on CPU, from 20.6% to 5.6% on DLA and demonstrates much improved efficacy in guiding SoC designs.

CCS CONCEPTS

• **Hardware → Emerging architectures**; • **Computer systems organization → System on a chip**; • **Computing methodologies → Model development and analysis**.

KEYWORDS

System-on-Chips, Accelerator Architectures, Performance Models

ACM Reference Format:
Yuanchao Xu, Mehmet E. Belviranli, Xipeng Shen, and Jeffrey Vetter. 2021. PCCS: Processor-Centric Contention-aware Slowdown Model for Heterogeneous System-on-Chips. In *MICRO'21: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3466752.3480101

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480101

1 INTRODUCTION

As domain specialization is proven to be a promising path to achieve high performance at low energy [18], integrated shared memory heterogeneous architectures have coupled CPUs with other accelerators on the same die to serve the demanding needs of autonomous, mobile, and edge computing. System on chips (SoC), such as NVIDIA's Jetson AGX Xavier [3], Qualcomm's SnapDragon [17], and Apple's A1X Bionic [39], embedded specialized processing units, such as vision processors (PVA), deep learning accelerators (DLA), and digital signal processors (DSP), under the same memory bus to efficiently run a variety of computations with distinct characteristics.

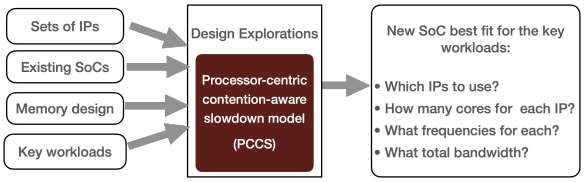

Figure 1: The SoC design problem focused in this study.

The diversity of processing units (PU) amplifies the complexity in SoC design; an issue studied in this work. Figure 1 illustrates the specific problem. An SoC design team needs to build an SoC to support the execution of some important workloads, such as an autonomous vehicle workload that consists of a set of related modules (e.g., object recognition, trajectory prediction, etc.). The team has access to a set of PUs as well as some existing SoCs, which each is equipped with some of the PUs. The team needs to decide on the design of the new SoC that best fits the workload of interest. Specifically, they need to determine (1) what PUs should be put onto the SoC, (2) how many cores of each PU, and (3) what frequencies or other configurations each PU should use. (4) what total memory bandwidth the SoC should have.

The team may be able to run every module (or kernel) of the workload on the PUs on existing SoCs to measure the performance of the module's standalone executions on different types of PUs. The challenge is on figuring out how the modules and the workload would perform if multiple apps are co-run on a new SoC. As PUs on a SoC typically share the memory and bus, the co-location would

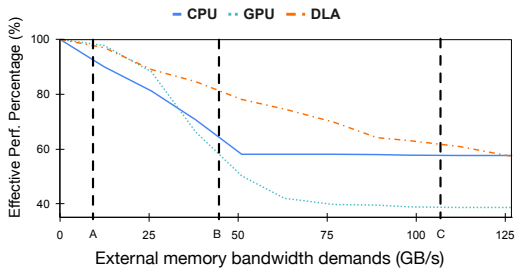

Figure 2: The percentage of the requested memory BW that is met on a processor under various degrees of external memory pressure. The requested memory bandwidths are 30GB/s, 93GB/s, and 127GB/s on the DLA, CPU, and GPU respectively. The peak memory BW of the SoC (NVIDIA Xavier AGX) is 137GB/s. A, B, C on the X-axis mark the points where requested BW + external BW = DRAM peak BW, for GPU, CPU, DLA respectively. The effects of contention are being observed even when the sum of requested BW and the external memory pressure is less than the total DRAM BW.

incur memory interference and hence performance degradations of various degrees as Figure 2 illustrates.

Many prior performance models have included memory interference into considerations [9, 10, 12, 15, 16, 23, 27, 29–31, 38, 41, 42, 46, 47]. They are however for *post-silicon runtime optimizations* rather than the usage in the SoC *design stage*. Among efforts in creating performance models to guide hardware designs [7, 19, 40, 48], the state of the art work is Gables [19]. The work is valuable as the first attempt to integrate memory interference into Roofline models. Its proposed performance model is however remarkably rough, with many oversimplifications. It, for instance, assumes available memory bandwidth is proportionally distributed among the heterogeneous PUs. Evidence has shown large errors from those simplifications. The model for example suggests zero slowdowns for co-running applications if the sum of their standalone-run memory bandwidth consumption is less than the total memory bandwidth of the SoC, which contradicts the empirical results shown in Figure 2. How to effectively handle memory interference in SoC design remains a problem yet to be solved.

This paper proposes a new approach, named processor-centric contention-slowdown modeling (PCCS), to address this important problem. PCCS is based on a systematic analysis of the impact of co-run memory contention, and comprises a new methodology and a new slowdown model.

- *Analysis:* We conduct a series of experiments to observe the influence of co-run memory contention. The observations contradict the proportional distribution of memory bandwidth assumption the prior SoC co-run model [19] builds on. Through an in-depth analysis, we validate that fairness control in memory controllers is an important reason for the observed co-run slowdowns—a factor neglected by prior SoC co-run modeling [19].
- *Methodology:* The new approach leverages a *source-obliviousness insight*, that is, the influence external memory interference has on the performance of an application is determined by the degree of interference, and is largely oblivious to what the sources of the external traffic are. Led by the insight, the new approach employs a *processor-centric* modeling scheme, which uses a set of *calibrators* (controllable memory traffic generators) to assist the empirical measurements in determining the model parameters. The modeling process needs no measurement of co-runs of various combinations of applications, but the produced slowdown models can be applied to arbitrary applications.
- *Slowdown model:* The new slowdown model is a *three region interference-conscious model*, which classifies an application, based on its algebraic computation intensity, into one of three categories, each of which features a distinctive class of slowdown models. The model is processor-centric, characterizing the architecture behavior in the presence of external memory bandwidth demands. Piecewise formulation is used in model formulation to ensure accuracy.

We evaluate the general applicability of our slowdown model and demonstrate that it is precise enough for guiding SoC design explorations. We validate our model on a set of Rodinia benchmarks and some deep-learning operations on two real devices, an NVIDIA's Jetson AGX Xavier autonomous SoC [3] which contains three different PUs (8-core ARM CPU, Volta GPU, and deep learning accelerator (DLA)), and a Qualcomm's SnapDragon 855 mobile SoC [6] equipped with 8-core Kryo CPUs and Adreno™640 GPUs. The new method reduces average prediction errors of the state-of-art model from 30.3% to 8.7% on GPU, from 13.4% to 3.7% on CPU, from 20.6% to 5.6% on DLA. In use case studies, the results help avoid over-provisioning PUs or their frequencies, saving up to 50% area (with reduced cores) or 52.1% power budget (with reduced frequencies) over the suggested configurations by prior models, while maintaining the same level of actual co-running workload performance. The results confirm the effectiveness of the proposed method and models in bridging the gap in the current support of heterogeneous SoC designs.

2 MEMORY INTERFERENCE CHARACTERIZATION

The first step in building our proposed slowdown model is to understand and characterize the contention occurring when kernels with different memory access behaviors are co-located on different PUs.

2.1 Target Architecture

Our slowdown model targets the heterogeneous shared memory (HSM) SoC, which has multiple types of processors, a memory controller interface to shared DRAM memory, and a high-bandwidth on-chip interconnect. An example is NVIDIA's Jetson AGX Xavier architecture [3], shown in Figure 4. It is a recent HSM-SoC targeting autonomous computing applications. Xavier integrates 8-core ARM v8.2 CPU, 512-core NVIDIA Volta with 64 TensorCores, an NVIDIA Deep Learning Accelerator (DLA), a Programmable Vision Accelerator (PVA) and multimedia accelerators on the same die, and they share the same system memory. The memory uses channel interleaving to construct 256-bit width from 8 32-bit channels. The theoretical peak bandwidth is 136.5GB/s.

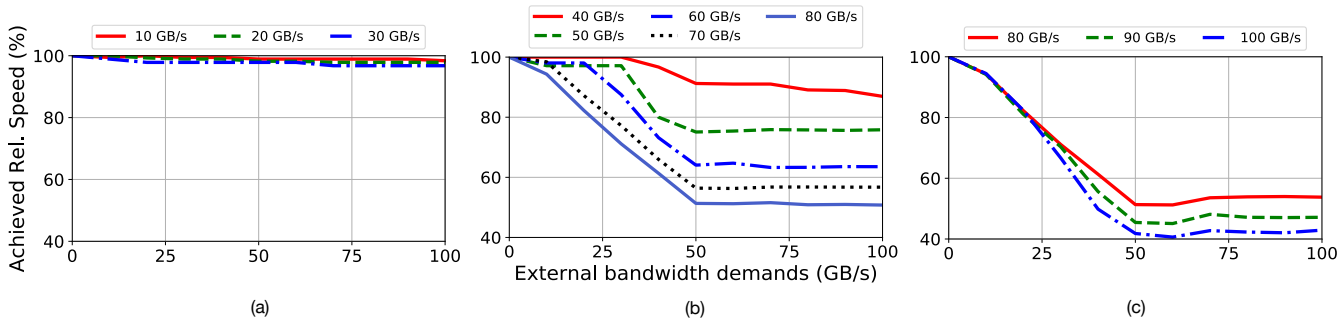

Figure 3: The performance of synthetic programs under different memory pressures. The standalone requested memory BW varies from 10 GB/s to 30 GB/s in (a), from 40 GB/s to 80 GB/s in (b) and from 80 GB/s to 100 GB/s in (c).

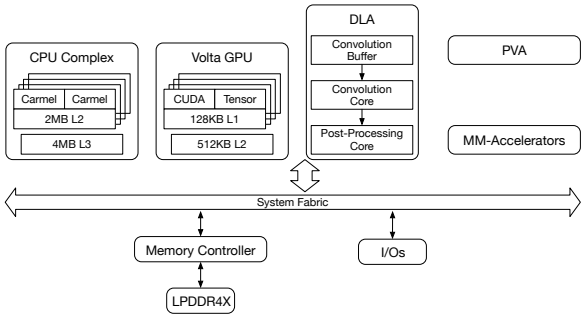

Figure 4: The NVIDIA Jetson AGX Xavier Architecture

The target HSM-SoC has two common properties: (i) PUs operate concurrently. (ii) The available memory bandwidth is shared among PUs. Our analysis focuses on the common scenarios where execution of a kernel spans only one type of PU and a PU runs only one kernel at a given time.

2.2 Observations

We first introduce a term: the *bandwidth demand* of a kernel on a PU is the memory bandwidth requested by that kernel in its *standalone* execution on that PU without kernels running on other PUs. We identify three main factors that affect *co-run slowdown*, that is, the slowdown of a kernel K_i running on PU P_j when other kernels are running on other PUs:

- The maximum standalone achieved speed that a specific PU P_j used by the kernel K_i can achieve.
- The bandwidth demand of kernel K_i on PU P_j.
- The total bandwidth demanded of other kernels running on other PUs.

To understand how an HSM-SoC behaviors under these three factors, we experiment with the kernels in the Roofline Model construction suite [1], which is a collection of synthetic vector add and multiplication kernels with adjustable memory bandwidth (BW) demands and operation intensity. We use the GPU and CPU of the NVIDIA's Xavier. We vary the BW demand of each of the kernels from 10GB/s to 100GB/s with a 10GB/s increase. For each experiment, we also create a synthetic external BW demand that varies from 0 to 100GB/s and co-run it with the experimented kernel.

The results are shown in Figure 3. The y-axis is the percentage of the achieved standalone speed of the kernel of interest. The x-axis is the external memory BW demand. (The actual external BW pressure is equal to or lower than the demand.)

The workloads fall into three categories.

(1) Figure 3 (a) shows the observations on the kernels that request only a small amount of memory BW. Their achieved speed drops slightly as external BW demand increases.
(2) Figure 3 (b) shows the kernels that demand a more significant amount of BW (i.e., 40-80 GBps), and the achieved speed curves exhibit a three-stage trend:
   (a) The curves start with a relatively flat segment, showing the little influence of external pressure on the achieved speed.
   (b) The curves enter a fast near-linear dropping region when the external BW demand increases beyond a certain level.
   (c) The curves then flatten out as the external BW demand exceeds a certain level.
(3) Figure 3 (c) shows the observations of the kernels that request a large amount of BW. Even when there is just a small external BW demand, the achieved speed reduces significantly. But when the external BW demand goes beyond a certain level, the curves flatten out.

The observations contradict the proportional distribution of memory BW assumption the prior SoC co-run model [19] builds on. For instance, according to that model, there should be no slowdowns at the beginning part of the curves in Figure 3 (b,c) as the total requested BW has not yet reached the peak BW of the system, and there should not be a flat part at the end of curves in Figure 3 (b,c). We further conduct an in-depth analysis of observed co-run slowdown trends to explain those observations, especially the flat part at the end of those curves.

2.3 Validation: Fairness Control

Our study shows that the observed co-run performance trends are the results of the prioritization of row-hit requests in memory controllers (MC) and the fairness control employed in MC. MCs typically prioritize row-hit requests in the MC queues to maximize

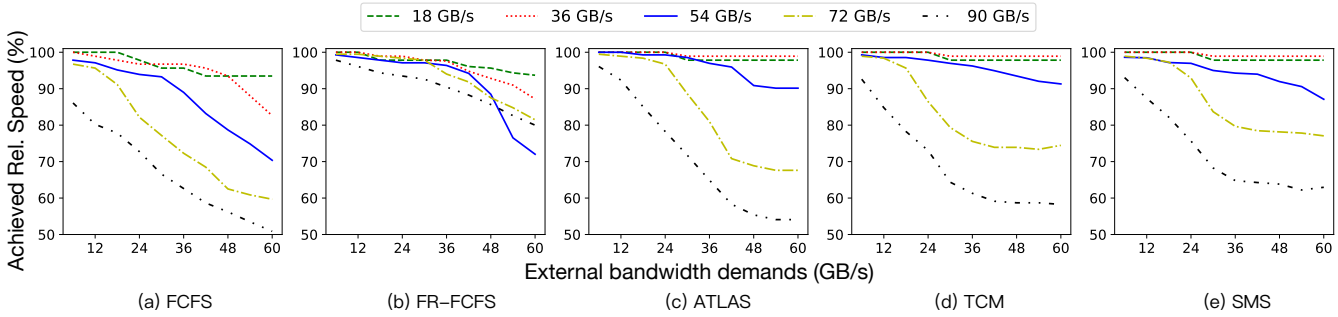

Figure 5: The achieved relative speed (%) of synthetic programs running on high BW group under different memory pressure on different scheduling policies.

total bandwidth [35]. But when more PUs simultaneously access the memory, the high row-buffer hit rate can no longer be maintained. Hence, even when the cumulative BW demand is less than what the memory is capable of, a significant amount of memory access slowdown is observed, hence reducing the speed.

The reason for flattened segments at the end of curves is the fairness control [15, 33, 34] in MCs. As prior studies have shown [8, 24, 25, 32–34], adopting fairness-aware scheduling policies in MCs is essential for multi-core processors and HSM-SoCs. Without fairness control, the shared memory multi-core processors and HSM-SoCs will experience three major problems: 1) Low system throughput; 2) Vulnerability to denial-of-service; 3) Unpredictable slowdown and uncontrollable quality-of-services (QoS). The main reason is that memory-intensive programs would hog most BW in uncontrolled memory interference.

To validate that fairness control is the reason for observed performance trends, we conduct a series of measurements. Because commercial systems do not disclose detailed MC designs, we conduct the study via a cycle-accurate x86 CMP DRAM simulator, Ramulator [26], by using scheduling policies with and without fairness controls. The front end of the simulator is based on Pin [28]. We model the memory system in detail to faithfully capture bandwidth limitation and contention, and enforce bank/channel/bus conflicts. Table 1 shows the major DRAM and processor parameters.

We construct several synthetic vector add and multiplication kernels built from the roofline toolkit [1] with different memory BW demands. To simulate heterogeneous scenarios, we regard the 16 cores as two classes, with cores 0-7 as the low-bandwidth group and cores 8-15 as the high-bandwidth group. In the low BW group, we vary the requested standalone memory BW of the kernels from 6GB/s to 60GB/s with a 6GB/s increase, and from 9GB/s to 90GB/s with a 9GB/s increase in the high BW group.

We evaluate five memory scheduling policies: first-come-first-serve (FCFS), first-ready-FCFS (FR-FCFS) [35], Adaptive per-Thread Least-Attained-Service (ATLAS) [24], Thread Cluster Memory Scheduling (TCM) [25], and Stage Memory Scheduling (SMS) [8]. The last three policies adopt fairness control. The brief description of these policies are shown in Table 2. The default parameters of these policies are used.

The results of these five policies are shown in Fig. 5, where, the y-axis is the percentage of the achieved standalone speed of kernels. Table 3 reports the average row buffer hit rates (RBH) and the effective BW percentage over the theoretical peak BW when the sum of co-located programs' standalone BW is equal to or larger than the theoretical peak BW of these 5 policies.

In the FCFS results, as shown in Fig. 5 (a), the achieved speed is reduced proportionally with external BW demands. Since MC with FCFS deals with memory requests chronologically, each row buffer deals with requests without locality awareness, leading to low RBH and small effective BW (Table 3) in co-location scenarios. FR-FCRS improves BW usage but does not adopt fairness controls. In Fig. 5 (b), the programs suffer from large slow down when they

Table 1: Config. of Memory Controller Simulation

Processor	16-core, 2.2 GHz, 128 entries reorder buffer;
Cache	Private L1D cache, 8-way, 64KB, 4 cycles; Two-core shared L2 cache, 8-way, 1MB, 9 cycles; Shared L3 cache, 16-way, 4MB, 26 cycles.
DRAM Controller	256-entry request buffer, XOR-based address-to-bank mapping
DRAM Chip Parameters	DDR4-3200 timing parameter [26], 8 banks, 4K-byte row buffer per bank, Single rank, 4 channels, 64-bit wide channel, 102.4 GB/s theoretical bandwidth

Table 2: Scheduling policies of memory controllers(MC)

Policy	Description
FCFS	MC schedules memory requests chronologically.
FR-FCFS [35]	MC prioritizes row-hit requests.
ATLAS [24]	Prioritization order: 1) Over threshold request. 2) Requests from the thread that has attained least service. 3) Row-hit requests. 4) Oldest requests.
TCM [25]	Prioritization order: 1) Non-memory-intensive programs 2) Periodically rank shuffle memory-intensive programs 3) Row-hit requests. 4) Oldest requests.
SMS [8]	Steps:1) Group requests to the same row into batches 2) Schedule batches with p probability shortest first and (1-p) probability round-robin

Table 3: Row buffer hits(RBH) and effective BW

Policies	FCFS	FR-FCFS	ATLAS	TCM	SMS	Xavier
RBH (%)	47.7	91.6	74.2	79.6	84.7	-
Effective BW Percentage over Peak BW (%)	65.6	89.7	78.4	80.8	84.3	79.1

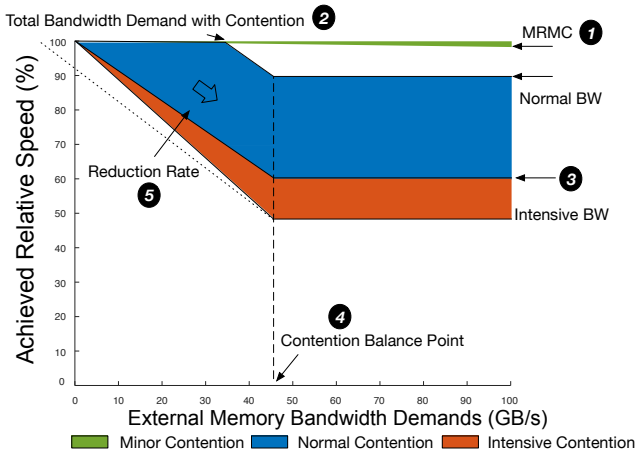

Figure 6: The three-region interference classification model.

Table 4: Notations and Model Parameters

Term	Description
Normal BW	The level of BW demand by the current PU that separates the minor and normal contention regions.
Intensive BW	The level of BW demand by the current PU that separates the normal and intensive contention regions.
Max. Reduction of Minor Contention (MRMC)	The maximum slowdown in the minor contention region at the largest external memory pressure
Contention Balance Point (CBP)	The value of the external memory demand where the speed curve of the current PU starts going flat.
Total Bandwidth Demand with Contention (TBWDC)	The sum of the BW demand by a PU and the total external BW demand where the speed curve of the current PU enters the dropping phase in the normal contention region
$rate^N$	The reduction rate of the achieved speed in the normal contention region
$rate^I$	The reduction rate of the achieved speed in the intensive contention region
Peak Bandwidth (PBW)	The peak bandwidth of the entire HSM-SoC
Achieved Relative Speed (RS)	The percentage of the standalone execution speed that has been achieved

are co-located with memory intensive programs, resulting in low system throughput.

On the other hand, the results on all three scheduling policies with fairness control exhibit trends similar to our observed trends on Xavier (Fig. 3). ATLAS, for instance, tries to maintain fairness according to attained services. In HSM-SoCs, the processor attained the least services will be prioritized, leading to similar attained services across processors. As shown in Fig. 5 (c), when a program needs a small BW and it is co-located with a high BW demand program, its achieved speed remains nearly unchanged, as its bandwidth demand gets satisfied by the MC prioritization scheme. A medium BW demanding program however sees some drops of its achieved speed at the beginning as its demands exceed what the prioritization scheme of the MC offers. When the external BW demand grows, the slowdown becomes larger, as other co-located programs send more requests in their own time slots. When the external BW demand keeps growing, they eventually reach a stable state under the MC scheduling policy, hence the flat segments in the performance curves. For a high BW demanding program, it is similar except that they get into the second phase from the beginning. Similar effects are shown in the curves of TCM and SMS as Figures 5 (d,e) show, although there are some detailed differences due to the differences in the specifics of the schedulers; detailed discussions are omitted for the sake of space.

3 MEMORY INTERFERENCE SLOWDOWN MODEL

Based on the observations in Fig. 3, we propose a *three-region memory interference slowdown model* for a processor in an HSM-SoC. Our approach is statistical via regression-based analysis. An alternative is to construct the model analytically on the detailed memory controller designs. This approach requires detailed knowledge on the memory controller of each target device. As memory controllers in commercial systems are usually undisclosed, the approach is infeasible in general.

In the three-region model, we create a linear function for each of the three regions demonstrated in Fig. 3 (a), (b), and (c), and present them in the unified chart given in Fig. 6:

(1) *Minor Contention Region* is the top-most region in the figure where the workload on the current PU requests memory bandwidth that is low enough so that the effects of the external memory bandwidth demand are minimal.
(2) *Normal Contention Region* is where the current PU's requested memory bandwidth is at a middle level such that the interference performance curve follows the pattern in Fig. 3 (b). The pattern has two points determining the beginning and end of its linear region, and they are marked as *Total Bandwidth Demand with Contention* point and *Contention Balance Point*, respectively.
(3) *Intensive Contention Region* is the bottom-most region in the figure where the current PU's requested bandwidth is so high that the achieved speed is significantly affected by the external bandwidth demand.

An important feature of this classification is that it is *processor-centric* since it characterizes each PU in the system separately rather than creating peer-to-peer contention analysis as many CPU+GPU based studies do. The different PUs on the same HSM-SoC would have different values for contention region boundary, contention balance point and the reduction rate. For example, the GPU has a large number of threads and its standalone performance can hide the memory latency, hence using more bandwidth of the entire HSM-SoC. However, when the GPU is under contention from external traffic caused by another processor in the HSM-SoC, the point of *total bandwidth demand with contention* would be larger. On the other hand, the reduction rate in both normal and intensive contention regions is larger. The achieved relative speed after the contention balance point is also smaller for GPUs. We next explain the exact formulations of the slowdown curves as well as how the model parameters can be determined for a PU.

3.1 Model Formulation

Table 7 summarizes the parameters and concepts used in our model. We begin building our model by defining *region* partitions as shown

in Equation 1. The x represents the bandwidth requested by the kernel running on the current PU. The top six values are PU-specific values that can be obtained using the method explained in Section 3.2.

$$Region = \begin{cases} Minor & 0 \le x \le normal\ BW \\ Normal & normal\ BW \le x \le intensive\ BW \\ Intensive & intensive\ BW \le x \end{cases} \quad (1)$$

To model the contention in the minor contention region, we use the *MRMC*, the constant rate of reduction observed for this region. The achieved relative speed in the minor contention region, RS^M, is as follows:

$$RS^M = 100\% - \frac{MRMC * x}{PBW} \quad (2)$$

For the normal contention region, the achieved relative speed, RS^N, is a piece-wise function, as shown in Equation 3, where y represents the total external memory demand.

- The first piece of Equation 3 represents the case when the summation of the PU BW demand and external demanded $(x+y)$ is smaller than the processor-specific value of *TBWDC* and y is smaller than *CBP*. In this case, the achieved relative speed of the current PU is the same as the minor contention part.
- When $x+y$ is larger than *TBWDC* and y is smaller than *CBP*, the achieved relative speed of the current PU is reduced by a constant $rate^N$ multiplied by $x + y - TBWDC$.
- When y is larger than *CBP*, the reduction of achieved relative speed of the current PU remains constant, which is defined as $100\% - (x + CBP - TBWDC) * rate^N$.

$$RS^N = \begin{cases} 100\% - \frac{MRMC*x}{PBW} & 0 \le x + y \le TBWDC \\ & and\ y \le CBP \\ 100\% - (x + y - TBWDC) * rate^N & TBWDC \le x + y \\ & and\ y \le CBP \\ 100\% - (x + CBP - TBWDC) * rate^N & CBP \le y \le PBW \end{cases} \quad (3)$$

For the intensive contention region, since the requested BW already exceeds the *TBWDC*, the achieved relative speed starts reduction with minimal external pressure demand and the reduction rate R^I is larger. We obtain the rate $rate^I$, shown in Equation 4, by extending the performance reduction curve (dotted lines in Fig. 6) by multiplying with rate $rate^N$ in the normal region and then by dividing it by *CBP*.

$$rate^I = \frac{rate^N * (x + CBP - TBWDC)}{CBP} \quad (4)$$

The achieved relative speed of an intensive region, RS^I, has a reduction stage and a flat stage. The piecewise equation is shown in Equation 5. The two pieces of this equation are identical to the second and third pieces of Equation 3 except that $rate^N$ is replaced with $rate^I$.

$$RS^I = \begin{cases} 100\% - (x_i + y - TBWDC) * rate^I & TBWDC \le y \le CBP \\ 100\% - (x_i + CBP - TBWDC) * rate^I & CBP \le y \le PBW \end{cases} \quad (5)$$

3.2 Model Construction

The slowdown model relies on several PU- and SoC-specific parameters that need to be determined via either standalone or collocated runs. To achieve varying amounts of memory BW requests, we create synthetic kernels with different compute/memory (i.e., operational) intensities, similar to those used in the Roofline model [40]. The page https://github.com/processorcentricmodel/PCCS shows the code and pseudo code. The basic idea is to have the PU load each word in an array of a certain size and perform some operations. The BW of this kernel is the total memory access size (i.e., twice the array size) divided by execution time. We modify the number of operations per word to control the operation intensities to generate different BW kernels. They serve as calibrators of our models. For each variation of operational intensity, we record and report the resulting standalone BW demand on the architecture it runs on.

The construction process of our model uses the calibrators (synthetic traffic generators) based on the *source-oblivious assumption*, that is, the amount rather than the source of external traffic matters. It makes the construction processor-centric, avoiding running many combinations of co-running scenarios. We validate the assumption on NVIDIA Jetson AGX Xavier [3] by creating the external traffic with different sources but with the same total amount. The achieved relative speed was very close.

The model construction for a target PU includes two main steps. The first step is to run each of the BW kernels on the target PU under various external BW demands. The external BW demand is generated by running the BW kernels on other PUs. We record the achieved relative speeds of the kernels on the target PU under the various external BW demands into a two-dimensional matrix. The element $rela[i][j]$ in that matrix is the achieved relative speed of the i-th smallest kernel running on the target PU under the j-th smallest external BW demands.

The second step is to analyze that matrix to determine model parameters in Fig. 6. The algorithm takes this matrix (rela[n][m]), standalone BW (stdBW[n] and external BW (extBW[m]) as the input. The analysis includes 5 steps to determine model parameters.

[1] To find the *normalBW* and *MRMC*, ❶ in Fig. 6, the algorithm examines the last column of array rela[:][m-1]. The first value from the top (row 0) that is two times larger than rela[0][m-1] defines the boundary, $k^{boundary}$, for the normal region, and the BW demand corresponding to this row is the *normal BW*. The element on the previous row and last column defines the *MRMC* value.

[2] For the boundary program row, the algorithm examines the value from left to right. The first column with notable reduction (2**MRMC*) defines the total bandwidth demand with contention by summing the total BW demand, *TBWDC*, as shown with ❷ in Fig. 6.

[3] The algorithm examines the first column from the top (row 0). The first element that has a notable (2**MRMC*) reduction defines the intensive bandwidth boundary ❸ in Fig. 6.

[4] Among the normal region rows, we calculate the relative speed changes in each row from left to right to find the turning points (columns) that start the flat region. The average external BW of these points defines the contention balance point ❹ in Fig. 6.

[5] The average achieved performance reduction rate within the normal region and the contention balance point is the reduction rate of the normal contention region ❺.

After these steps, the PCCS model of a PU is constructed for this HSM-SoC.

Handling multi-phase programs: To apply PCCS to mult-phase programs, we can divide the program into different phases and apply the prediction on each phase to obtain slowdown under contention. The total slowdown is obtained by aggregating each phase according to the execution time percentage in standalone running. We use `cfd` as a multi-phase example in Section 4.1 to demonstrate PCCS's applicability.

3.3 Model Scaling with Memory Bandwidth

In architecture design, memory and PUs are usually designed together. Fortunately, memory changes across the generations of SoCs are often incremental, consisting mostly frequency changes; the main memory technology changes (e.g., from DDR4 to DDR5) are less common [2]. For example, between 2013 and 2020, the same single-channel 32-bit low-power DDR3 (LPDDR3) memory architecture was deployed in over 20 different SoCs designed by Qualcomm. The only major memory-related change across these designs is the I/O bus clock frequency which varies between 533 and 933 MHZ [5].

While the PCCS model construction process outlined in Section 3.2 requires runs on a physical system, our slowdown model can well adapt to incremental clock frequency changes in the shared memory subsystem via *linear bandwidth scaling*. To demonstrate the feasibility of such an approach, (1) we first construct the original PCSS model on Xavier AGX when the memory clock is set to the highest possible value of 2133HMz and find the five key bandwidth-related parameters using the methodology prescribed in Section 4.1. (2) We then linearly scale down these five PCCS parameters to reflect the proportional frequency and channel-count changes if the memory clock were set to be 1066MHz, 1333MMz, and 1600MHz. (3) Finally, we under-clock the Xavier memory clock frequency to these three values and re-obtain the five PCCS parameters by following the empirical model construction process for each frequency.

Table 5 shows the difference between the scaled-down and the constructed values (i.e., the values which are empirically obtained by underclocking the target SoC) of the five parameters that determine a PCCS model. The results show that the average error due to the linear scaling of these parameters is lower than 3%. In summary, in the case where there is a major memory technology change, one would need to update the model through profiling on devices with the new technology; after that, the model can be used via simple scaling for the upcoming generations of the SoC just as described earlier in this section.

Table 5: Linear Parameter Scaling in PCCS

Parameters	Scaling method	Avg. error of scaled PCCS over constructed PCCS (%)
Normal BW (GB/s)	Linearly scaled parameters w.r.t to the ratio between the original and the target mem. freq. and # of channels.	1.5%
Intensive BW (GB/s)		2.1%
MRMC (%)		2.2%
CBP (GB/s)		1.7%
TBWC (GB/s)		2.2%
$Rate^N$ (% per (GB/s))	Calculated based on the scaled values of the parameters above	1.8%
$Rate^I$ (% per (GB/s))		2.5%

3.4 Uses in SoC Designs

The objective of hardware design space exploration is to predict how potential workloads would perform and minimize the hardware cost accordingly. This step typically needs many rounds to fine tune. PCCS helps this exploration process by providing how much slowdown the tasks will experience when running in a collocated manner. The accuracy of the slowdown predicted by PCCS is only as good as the accuracy of the standalone performance predictions or profilings fed into the PCCS model.

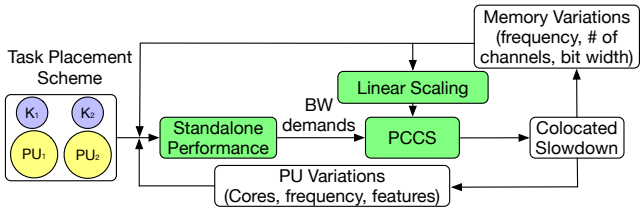

Figure 7: PCCS workflow.

The workflow of using PCCS is shown in Figure 7. A task placement scheme for an application indicates a mapping of kernels K_1 and K_2 to *PU*s in a system. For a given placement, we can obtain the standalone memory BW requirements of K_1, K_2, profiling results, and bandwidth model of each PU. We use the total external demand BW and the demand BW of the PUs as inputs for PCCS to obtain collocated execution slowdown so that designers can explore the effects of architectural features on the contention-based slowdown.

PCCS can help in the exploration of two critical sets of design parameters:

PU-related architectural changes: The effects of the number of cores in a PU (e.g., SMs or CPU cores), the PU clock frequency, and hardware features, such as SIMD, on contention-caused slowdown can be explored via PCCS. The architects can either utilize an existing architecture by reconfiguring it in software to obtain new standalone profiling results or analytically scale existing standalone performance predictions for BW. Then, the new standalone BW demand values can be fed into PCCS to obtain a slowdown prediction that can be used to decide whether this PU variation will meet the requirements. This process is explored in detail in Section 4.3.

Memory sub-system parameters: When designing new SoCs, depending on the target domain, architects often adjust memory subsystem parameters such as I/O clock frequency, number of channels, and bit-width. Then, the designer can either modify, if possible, the software configurable memory parameters of the existing architecture to obtain new standalone profiling results or they can scale their standalone performance models to reflect memory-related design changes. The PCCS model construction process does not require the collection of collocated execution related parameters for the new memory subsystem being considered. Instead, as detailed in Section 3.3, PCCS model parameters can be linearly scaled to match the resulting theoretical BW change ratio implied by the considered memory-related design changes. Then, the scaled BW parameters can be used to obtain a slowdown prediction that can be used to decide whether this memory variation is feasible.

Table 6: Two Experiment Platforms

NVIDIA Jetson AGX Xavier	
CPU	8-core Carmel 64-bit ARMv8.2 @ 2265MHz
GPU	512-core NVIDIA Volta@1377MHz with 64 Tensor Cores
DLA	NVIDIA DLA @1395.2 MHz, 512KB Convolutional buffer
Memory	16GB 256-bit LPDDR4x @ 2133MHz | 137GB/s
Qualcomm Snapdragon 855	
CPU	8-core Kryo 485 64-bit ARMv8.2 @1.8GHz
GPU	Qualcomm® Adreno™ 640 GPU
Memory	16GB 64-bit LPDDR4x @ 2133MHz | 34GB/s

4 EVALUATION

4.1 Empirical Validation of The Slow-down Model

Our slowdown model explained in Section 3 is built via synthetic benchmarks. In this section, we validate the prediction accuracy of our model using several widely used benchmark kernels and neural networks.

4.1.1 Setup and Methodology. Target architectures: We use two models of real heterogeneous SoC for experiments. One is NVIDIA Jetson Xavier autonomous system [3] which consists of CPU, GPU and DLA. The other is Qualcomm Snapdragon 855 [4] mobile platform consisting of CPU and GPU. Table 6 shows the architecture details.

Model construction: We follow the methodology detailed in Section 3.2 to build the models. For CPU and GPU, we employ vector-add kernels with different operational intensities; for DLA, we use MNIST neural network and control its operational intensities by varying convolution filter sizes.

Application selection: We evaluate our CPU and GPU slowdown model on Rodinia benchmarks [11], and our DLA model on ImageNet [13] with ResNet-50 and VGG19 models. We select 10 Rodinia benchmarks: three of them, `hotspot` (HS), `leukocyte` (LC) and `heartwall` (HW), are compute intensive and 7 of them, `streamcluster` (SC), `pathfinder` (PF), `srad`, `k-means` (KM), `b+tree` (BT), `CFD` and `BFS`, are memory intensive.

Bandwidth characterization: To find requested memory bandwidths of applications and kernels, we need only the standalone BW rates which can be obtained through NVperf, perf or Valgrind.

External memory pressure: To create contention, we run synthetic kernels on other PUs. For the CPU model, we create the external pressure using the GPU; for the GPU and DLA models, we create the external pressure using the CPU. For every benchmark, we vary the external pressure by changing the BW request of the other processor from 10% to 100% of the peak DRAM BW on this PU with a 10% peak BW as the stride.

Baseline: Gables [19] is the closest and most recent work that proposes a sharing-aware analytical model for different types of PUs that run on HSM-SoCs. The memory contention model proposed by *Gables* assumes that the effective bandwidth of a processor under contention is not reduced as long as the total BW requested is smaller than the SoC peak BW. Otherwise, the effective BW is calculated by pro-rating the requested BW to the available BW.

4.1.2 Validation Results. Fig. 8 shows actual slowdown of 10 Rodinia benchmarks running on the Xavier GPU as well as the predicted slowdowns by *PCCS* and *Gables*, under varying amounts of external memory contention. The average error of our model for GPU is 6.3%. For the benchmarks with small requested BWs, their achieved speed is close to the standalone one under memory contention. For other benchmarks that need medium BW in standalone runs, the slowdown for lower external memory pressure is minimal. However, as the pressure increases to exceed a certain level, their achieved relative speed drops significantly with the external pressure, and eventually the slowdown flattens, as predicted by our three-region contention categorization. Our model consistently and accurately classifies and predicts these trends. The highest prediction error occurs with the `BFS` kernel, which has a poor locality that is affecting the row buffer hit rates significantly.

Fig. 9 compares the predicted (*PCCS* and *Gables*) and the actual achieved relative speeds of 5 Rodinia benchmarks that are run on the Xavier CPU under external memory contention. Overall, the average error of our model for CPU runs is 2.6%. Since `hotspot` is computation intensive, it belongs to the minor contention region. The other four benchmarks belong to the normal contention region. The accuracy slightly decreases during the flat regions of `streamcluster`, `pathfinder` and `k-means`, however the upper bounds for those regions still hold and overall has significantly less errors when compared to Gables.

Fig. 10 compares the predicted (*PCCS* and *Gables*) and the actual achieved relative speed of 10 Rodinia benchmarks running on the Snapdragon 855 GPU, under varying amounts of external memory contention. In this experiment, the average error of our model for GPU is 5.9%. The results on the Snapdragon CPU are more accurate, as shown in Figure 11 with only a 3.1% average error in the predicted slowdown. As Snapdragon architecture uses different memory controllers and different PU designs from Xavier, programs show different standalone bandwidth demands. `Hotspot` for instance, runs with a lower requested throughput on Kyro cores of the Snapdragon due to reduced frequencies in the core and memory. Our model hence moves it into the minor contention category. Despite the differences between the two architectures, our PCCS model gives accurate predictions of co-run slowdown on both of them.

The achieved relative speeds for DLA runs on Xavier are shown in Fig. 12. The average error of *PCCS* slowdown prediction for this PU is 5.3%. Since the DLA is a specialized processor for inference, we observe that the DLA can only achieve 20-30GB/s bandwidth in most standalone runs. Therefore, the DLA's slowdown under external pressure only falls under the normal contention region, while the kernels running on DLA are still sensitive to external memory pressure. As shown in Fig. 12, the achieved relative speed keeps reducing until ~70 GBpS of external pressure and there is only a small flat region at the higher end of external pressure demand.

In comparison, the average prediction errors of the *Gables* model on the GPU, CPU and DLA on Xavier are 39%, 10.3% and 26.7%, and on the CPU and GPU on Snapdragon are 8.1% and 37.6% respectively. The reasons for the low accuracy are (i) it assumes the peak bandwidth is always achievable, and ignores the effects of contention; (ii) it assumes that the effective bandwidth can be proportionally divided down when the requested BW exceeds the DRAM capacity.

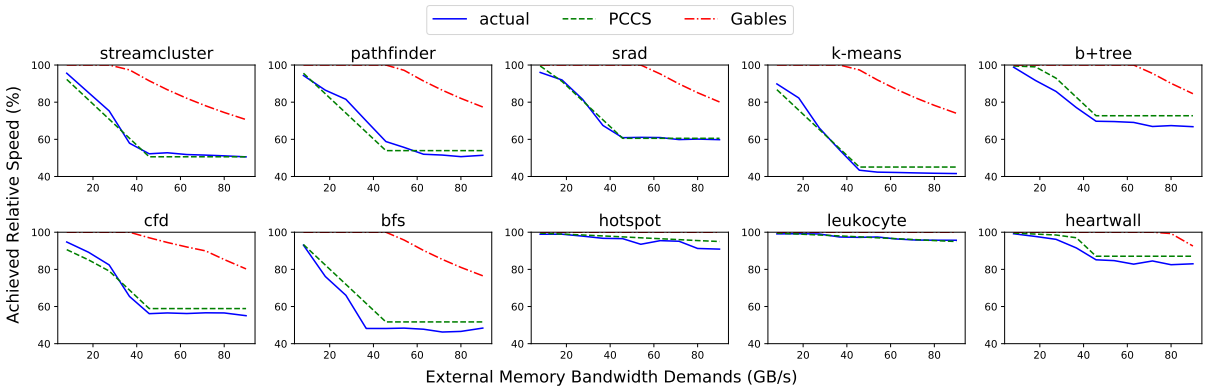

Figure 8: The predicted and actual slowdowns of 10 Rodinia benchmarks on Xavier GPU

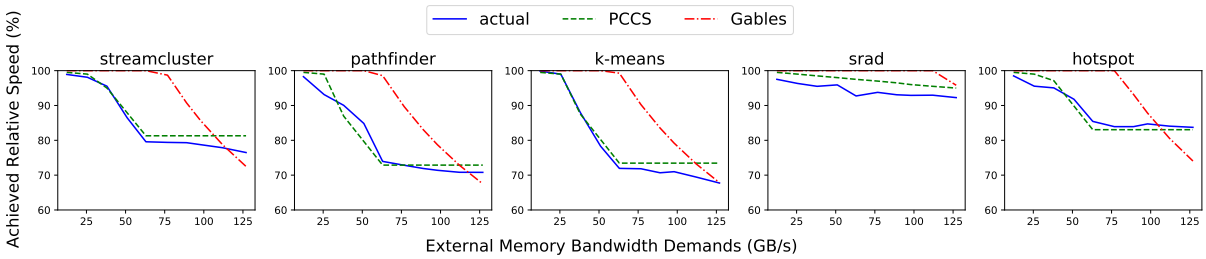

Figure 9: The predicted and actual slowdowns of 5 Rodinia benchmarks on Xavier CPU

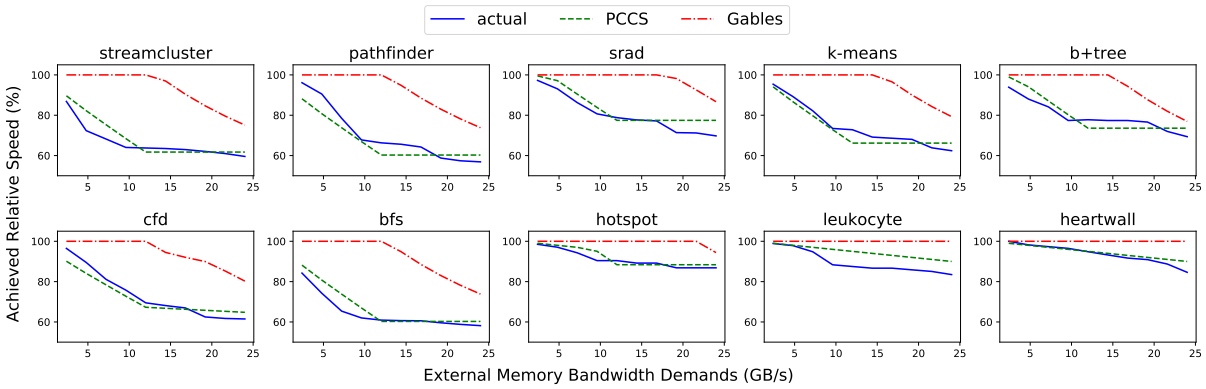

Figure 10: The predicted and actual slowdowns of 10 Rodinia benchmarks on Snapdragon 855 GPU

Table 7: Model Parameters

Parameters	Xavier			Snapdragon	
	CPU	GPU	DLA	CPU	GPU
Normal BW (GB/s)	37.6	38.1	0	6.8	9.1
Intensive BW (GB/s)	65.7	96.2	27.9	19.1	24.1
MRMC (%)	3.7	4.9	NA	5.7	9.8
CBP (GB/s)	46.6	45.3	71.1	16.1	12.8
TBWC (GB/s)	82.8	87.2	22.1	14.1	13.4
$Rate^I$ (% per (GB/s))	0.57	1.11	0.35	1.22	2.27

Model parameters: The model parameters are shown in Table 7. Different types of PUs on the same SoC differ in their model parameters. GPUs are more sensitive to external memory demand and they have a higher reduction rate than CPUs have. DLA is a special case in that it doesn't contain a minor contention region; as a result, even small external memory demands would trigger slowdowns of DLA. It is likely due to the lack of thread-level parallelism in DLA to hide memory latency.

Programs with phase shifts: We also test the prediction accuracy of our model with applications that involve shifts of phases and

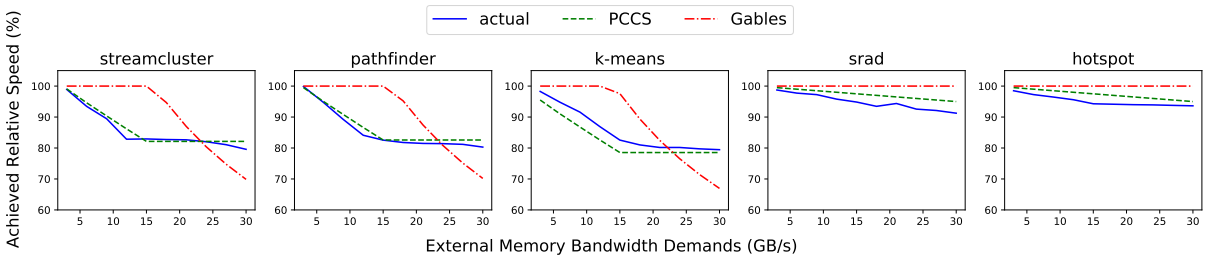

Figure 11: The predicted and actual slowdowns of 5 Rodinia benchmarks on Snapdragon 855 CPU

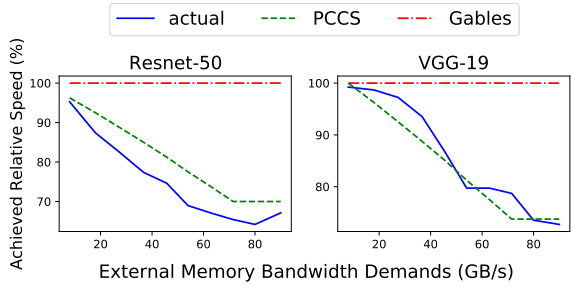

Figure 12: The predicted and actual slowdowns of VGG19 and Resnet50 on the DLA

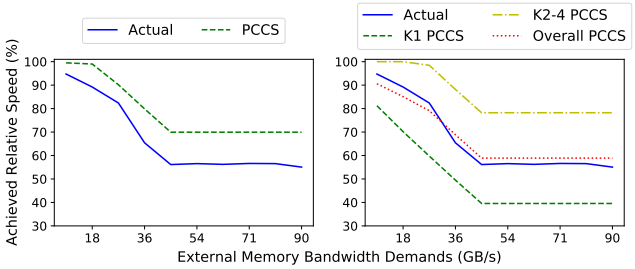

Figure 13: The predicted slowdowns of CFD with (a) average BW and (b) piece-wise BW

exhibit obvious changes in memory bandwidth demands. `CFD` is such a program, which embeds 4 different kernels where one of them (K1) is a high BW kernel and the other three (K2-4) are medium BW kernels. When we use the average BW of the 4 kernels to characterize the interference behavior of CFD, as shown in the results given Fig. 13 (a), our model's prediction has an error rate of 19.4%. It is because high BW demanding kernels suffer a larger slowdown, but using the average BW as the input causes our model to underestimate the slowdown. When we use the BW demand of each kernel as the input to our model and combine the predictions of different kernels by using the execution time percentage of each kernel as the weight, as shown in Fig. 13 (b), the prediction error drops to 4.6%. It indicates the usefulness of our model on code with phase shifts; phase detection [20, 36, 37] is a well-studied topic and is orthogonal to this work.

4.2 Results of Co-locations of Real Programs

To further demonstrate the accuracy of PCCS on real scenarios, we conduct the following experiments. We construct 3-PU co-run workloads. For each workload, every PU runs a benchmark from the Rodinia benchmark suite or an ML model. Table 8 lists 11 of the representative workloads that we show results. We measure and predict the achieved relative speed of the workloads (co-run speed over the standalone speed) until one of PUs finishes its program.

The results of the 11 workloads on CPU, GPU and DLA are shown in Fig. 14, where each bar shows either the actual or the predicted achieved relative speed, and the numbers on the prediction bars are the prediction errors. The average errors of PCCS on CPU, GPU and DLA are 3.7%, 8.7% and 5.6%, while the errors of Gables are 13.4%, 30.3%, 20.6% respectively. Different PUs show different actual achieved relative speed results. The programs with small BW running on CPU and GPU suffer from small memory contention even though the external BW demand is large. Meanwhile, the programs on the CPU have a smaller speed reduction than the programs on the GPU. DLA is more sensitive to external BW on different machine learning models. PCCS is able to capture PU-specific features. The errors on bfs, k-means and b+tree benchmarks are a bit larger than on other programs as those benchmarks create more BW pressure (e.g., row buffer hit rate).

4.3 Demonstrations of the Use in SoC Designs

This section demonstrates the use of PCCS in helping with the pre-silicon SoC design process. By comparing the PU configurations yielded from the use of PCCS and Gables models, we demonstrate how more accurate models could lead to more effective hardware designs.

Setting: The assumed setting is as follows. The architects would like to design an SoC similar to Xavier. Among the many optional designs, one design includes an 8-core ARMv8.2 CPUs, one GPU of a certain model, and one DLA of a certain model on the SoC. This is pre-silicon time, so the SoC is not available yet. The specific design task in this case study is to determine the appropriate frequencies to clock the GPU. The architects have a crucial workload used in guiding their hardware designs, and the kernel in the workload that is supposed to run on the GPU is a clustering kernel, the `streamcluster`.

Objectives: The architects would like to find out the appropriate frequencies to clock the GPU. They consider two cases, the clustering kernel suffers a corun slowdown of no more than 5% or no more than 20%.

Table 8: Workloads Settings

Workload	A	B	C	D	E	F	G	H	I	J	K
CPU	streamcluster	streamcluster	streamcluster	streamcluster	pathfinder	pathfinder	k-means	k-means	hotspot	srad	srad
GPU	pathfinder	pathfinder	leukocyte	srad	streamcluster	heartwall	b+tree	srad	bfs	pathfinder	leukocyte
DLA	Resnet-50	VGG-19	Alexnet	Resnet-50	VGG-19	Alexnet	Resnet-50	VGG-19	Alexnet	Resnet-50	VGG-19

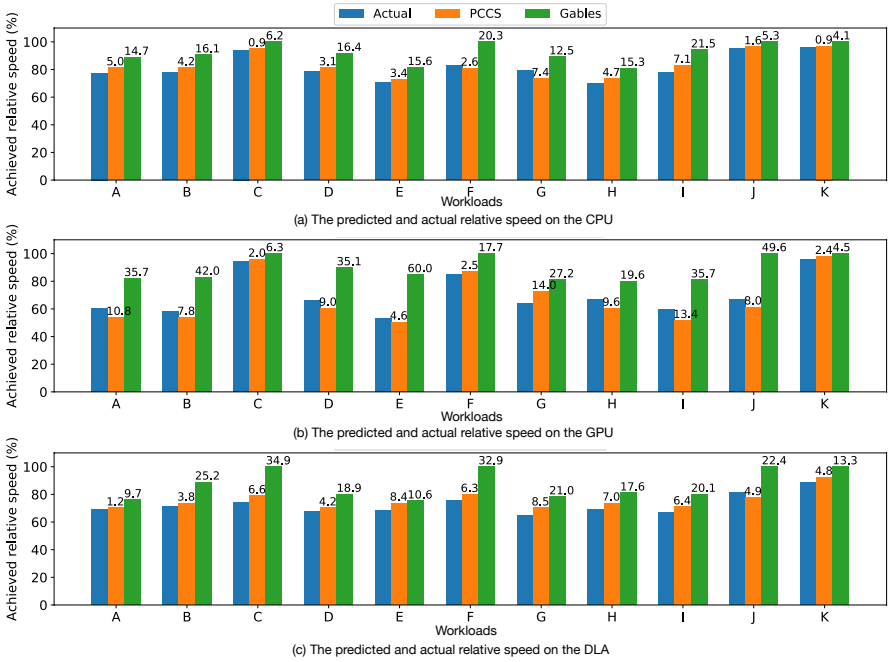

Figure 14: The predicted and actual achieved relative speed of 11 workloads on CPU (a), GPU (b) and DLA (c)

Table 9: PU frequencies selected by PCCS and Gables

External BW demand (GB/s)		20	40	60	20	40	60	20	40	60	20	40	60	Avg.	20	40	60	Avg.
		Selected PU frequencies (MHz) on different external BW demands									Erros (%)							
		Ground truth			PCCS			Gables			PCCS				Gables			
Maximum allowed	5%	840	650	620	860	670	630	880	880	880	2.4	3.1	1.6	2.4	4.8	35.4	41.9	27.4
Slowdown	20%	790	600	550	800	610	570	820	820	820	1.3	1.7	3.6	2.2	3.8	36.7	49.1	29.9

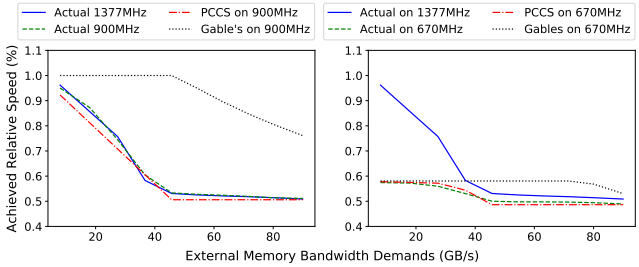

Figure 15: PU frequency exploration and validation for GPU using `streamcluster`.

Given: (i) the standalone performance models of each kernel in the workload, from which one can get the standalone performance and memory bandwidth demand of each kernel on any of the processors at a given frequency; (ii) the PCCS slowdown model of each of the processors; (iii) the Gables' performance model.

Method: The ways to use the two models are similar. The architects can use either of the two models, along with other given conditions, to get the co-run slowdown curves of the clustering kernel under each GPU frequency. From the curves, they can easily tell how much slowdown the kernel would suffer at a given GPU frequency and external memory BW contention. They can then pick the frequencies that meet the requirements in the two cases.

Validations: To quantitatively examine the quality of the outcomes from the use of each of the two models, we check how well the method works on the real NVIDIA Jetson AGX Xavier. We build both the PCCS and the Gables on this processor. For the validation purpose, we get the true standalone performance of the clustering kernel through profiling, and collect the true co-run slowdown curves of the clustering kernel at each frequency. Table 9 reports the frequencies picked by using the two models and the ground

truth at either the 5% or 20% allowed co-run slowdown case. In each case, it shows the results at one of three external memory BW contention levels, 20GB/s, 40GB/s, and 60GB/s.

The frequencies picked with PCCS are 1.3-3.6% off, while Gables 3.8-49.1% off. The large differences are the result of the different accuracies of the two models in predicting co-run slowdowns. Figure 15 shows the co-run relative speed curves predicted by the two models and the ground truth, at each of several frequencies. From the graphs, we can see that for `streamcluster`, its true co-run performance curve at the top frequency, 1377MHz, is almost identical to the curve at frequency 900MHz. It is because the standalone performance of `streamcluster` shows no drop until the frequency goes below 900MHz; there is hence no change in its memory bandwidth demands. Even though such information is given to both PCCS and Gables models, they get very different conclusions. PCCS accurately predicts the co-run curve, while Gable's model fails to do so. It mistakenly predicts that there is no memory contention at 900MHz when there is 20GB/s or 40GB/s external BW demand because the total BW of the SoC is sufficient to accommodate all demands, and hence gives the curve significantly different from the ground truth. The curves at frequencies 670MHz shown in the other graph in Figure 15 show the predictions of the two models when the reduction of frequency causes reductions of the standalone performance. Gable's result again shows a much larger discrepancy from the true curve than PCCS does.

This use case demonstrates that the more accurate slowdown model of PCCS can indeed lead to meaningful benefits in hardware design choices. Similarly, it can help decide the appropriate number of cores, types of PUs, or other hardware settings to use for an SoC.

5 DISCUSSIONS

We briefly discuss two complexities.

Synchronization: Our assumptions do not rule out inter-PU synchronizations as long as they occur after the end or before the beginning of a kernel. Synchronizations with external devices in the middle of the kernel is not common, consideration of which is left for the future.

Address mapping and multi-MC:. Recent HSM-SoCs (Xaiver series, Tegra X1, Tegra X2, and Snapdragon 855 series) usually use channel interleaving mapping and one MC to construct a wider bus width (256-bit), such that applications can use peak BW without considering address mapping. For the case where SoC uses multi-MC and maps different channels to each MC, our model can be extended to support that by considering specific address mappings and coordinations between MCs.

Power budget: Our current model does not explicitly model power or thermal throttling. The experiments on the two real-world SoCs already show much better accuracy than the state of the art; a deeper integration of those other factors could potentially further improve the accuracy. In SoC design, our current model could potentially work with power budgeting by predicting the co-run performance under each given power budget.

6 RELATED WORK

There is a rich set of studies on performance modeling with memory interference awareness. Table 10 lists some studies on memory interference modeling.

Table 10: Related Work Comparison

Related Work	Memory Interference Model	Accuracy	Arch. Design Exploration
Bubble-up [29]	Empirical	High	×
GDP [22]	Dynamic	High	×
Co-run [47]	Lookup Table	High	×
ESP [31]	Linear Regression	Medium	×
Gables [19]	Analytical	Low	✓
PCCS	Empirical & Analytical	High	✓

Bubble-up [29] proposes to empirically measure the memory interference sensitive curve over different other processor memory pressures for each application. Each application has its own sensitive curve which has a high accuracy in predicting the memory interference over memory pressure. Graph-based Dynamic Performance (GDP) [22] estimates the number of load-related stall cycles by multiplying critical path length with the estimated average private mode memory latency, whereby estimates interference-caused slowdown. Another work on co-run scheduling [47] proposes a memory interference performance lookup table to predict co-run performance. This lookup table is measured by profiling different co-run combinations. Another study [31] uses different co-run combinations to train a linear regression model. During the prediction, they use this linear regression model to match maximum likelihood co-run application characteristics. Through a black-box method, their accuracy is lower than the previous three studies. The above four studies provide an accurate memory interference performance for runtime usage purposes, but are for post-silicon runtime uses rather than SoC designs.

A variety of slowdown models have been proposed to improve hardware utilization or QoS. In CPUs, Du Bois *et al* [14] proposed hardware-efficient per-thread cycle accounting architecture for multicore processors to detect inter-thread cache/memory interference and predict slowdown percentage. Application Slowdown Model (ASM) [38] proposed to use a runtime monitor to periodically give the highest priority to one processor's memory requests to estimate standalone performance. In GPUs, Hybrid Slowdown Model [45] proposed to use a hybrid of white-box [21] and black-box [43, 44] methods to predict slowdown for multitasking GPUs.

7 CONCLUSION

Heterogeneous SoC design has been facing the dilemma of accuracy and coverage. Accurate simulations are time-consuming, difficult to cover a large design space; rough estimations by experience or analytical models face high risks of inaccuracy. The new approach proposed in this work strives to bridge the gap. We propose a novel *processor-centric performance modeling* methodology, and a new *three region interference-conscious performance model*. The modeling process needs no measurements of co-runs of various combinations of applications. The new method reduces average prediction errors of the state-of-art model from 24.8% to 8.7% on GPU, and from 13.0%

to 3.3% on CPU, and has demonstrated much-improved efficacy in guiding SoC designs in several use-case studies.

ACKNOWLEDGMENTS

We thank all the anonymous reviewers whose feedback is helpful for improving the final version of the paper. We also thank Seyong Lee, Narasinga Rao Miniskar and Mohammad Alaul Haque Monil, Steve A Moulton for their feedback and support. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CCF-2124010 and CNS-1717425, the US Department of Energy (DOE) Office of Science under contract no. DE-AC05-00OR22725, the Defense Advanced Research Projects Agency Microsystems Technology Office Domain-Specific System-on-Chip Program and DOE Office of Science, Office of Advanced Scientific Computing Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE.

REFERENCES

[1] [n. d.]. CS Roofline Toolkit. https://bitbucket.org/berkeleylab/cs-roofline-toolkit/src/master/. Accessed July, 2020.

[2] [n. d.]. DDR5 vs DDR4 All the Design Challenges and Advantages. https://www.rambus.com/blogs/get-ready-for-ddr5-dimm-chipsets. Accessed Feb, 2021.

[3] [n. d.]. NVIDIA TENSOR CORES. https://devblogs.nvidia.com/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/. Accessed Nov, 2020.

[4] [n. d.]. Qualcomm Snapdragon 855 Mobile Platform. https://www.qualcomm.com/products/snapdragon-855-mobile-platform/. Accessed Nov, 2020.

[5] [n. d.]. Qualcomm Snapdragon Processors. https://www.qualcomm.com/snapdragon/processors/comparison. Accessed Nov, 2020.

[6] [n. d.]. Snapdragon 855 Mobile Platform. https://www.qualcomm.com/products/snapdragon-855-mobile-platform. Accessed Sep, 2019.

[7] Gene M Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*. 483–485.

[8] Rachata Ausavarungnirun, Kevin Kai-Wei Chang, Lavanya Subramanian, Gabriel H Loh, and Onur Mutlu. 2012. Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems. In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 416–427.

[9] Rajkishore Barik, Naila Farooqui, Brian T Lewis, Chunling Hu, and Tatiana Shpeisman. 2016. A black-box approach to energy-aware scheduling on integrated CPU-GPU systems. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 70–81.

[10] David Black-Schaffer, Nikos Nikoleris, Erik Hagersten, and David Eklov. 2013. Bandwidth bandit: Quantitative characterization of memory contention. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 1–10.

[11] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 44–54.

[12] Younghyun Cho, Florian Negele, Seohong Park, Bernhard Egger, and Thomas R Gross. 2018. On-the-fly workload partitioning for integrated CPU/GPU architectures.. In *PACT*. 21–1.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

[14] Kristof Du Bois, Stijn Eyerman, and Lieven Eeckhout. 2013. Per-thread cycle accounting in multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013), 1–22.

[15] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices* 45, 3 (2010), 335–346.

[16] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. 2012. Fairness via source throttling: A configurable and high-performance fairness substrate for multicore memory systems. *ACM Transactions on Computer Systems (TOCS)* 30, 2 (2012), 7.

[17] Linley Gwennap. 2010. Two-headed snapdragon takes flight. *Microprocessor Report* 323 (2010), 1–6.

[18] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 37–47.

[19] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 317–330.

[20] M. Hind, V. T. Rajan, and P. F. Sweeney. 2003. *Phase shift detection: a problem classification*. Technical Report Report 22887. IBM Research.

[21] Qingda Hu, Jiwu Shu, Jie Fan, and Youyou Lu. 2016. Run-time performance estimation and fairness-oriented scheduling policy for concurrent GPGPU applications. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 57–66.

[22] Magnus Jahre and Lieven Eeckhout. 2018. Gdp: Using dataflow properties to accurately estimate interference-free performance at runtime. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 296–309.

[23] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. 2012. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 850–855.

[24] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE, 1–12.

[25] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 65–76.

[26] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices* 40, 6 (2005), 190–200.

[29] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 248–259.

[30] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. 2010. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 257–265.

[31] Nikita Mishra, John D Lafferty, and Henry Hoffmann. 2017. Esp: A machine learning approach to predicting application interference. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 125–134.

[32] Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE, 146–160.

[33] Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *2008 International Symposium on Computer Architecture*. IEEE, 63–74.

[34] Thomas Moscibroda Onur Mutlu. 2007. Memory performance attacks: Denial of memory service in multi-core systems. In *USENIX security*.

[35] Scott Rixner, William J Dally, Ujval J Kapasi, Peter Mattson, and John D Owens. 2000. Memory access scheduling. *ACM SIGARCH Computer Architecture News* 28, 2 (2000), 128–138.

[36] X. Shen, Y. Zhong, and C. Ding. 2004. Locality Phase Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 165–176.

[37] T. Sherwood, S. Sair, and B. Calder. 2003. Phase Tracking and Prediction. In *Proceedings of International Symposium on Computer Architecture*. San Diego, CA, 336–349.

[38] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 62–75.

[39] Wikichip. [n. d.]. Apple A13 Bionic. https://en.wikichip.org/wiki/apple/ax/a13. Accessed Jan. 2020.

[40] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[41] Yuejian Xie and Gabriel Loh. 2008. Dynamic classification of program memory behaviors in CMPs. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*.

[42] Shizhen Xu, Yuanchao Xu, Wei Xue, Xipeng Shen, Fang Zheng, Xiaomeng Huang, and Guangwen Yang. 2018. Taming the" Monster": Overcoming program optimization challenges on SW26010 through precise performance modeling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 763–773.

[43] Wenyi Zhao, Quan Chen, and Minyi Guo. 2018. KSM: Online Application-Level Performance Slowdown Prediction for Spatial Multitasking GPGPU. *IEEE Computer Architecture Letters* 17, 2 (2018), 187–191.

[44] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and reining in application-level slowdown on spatial multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 653–663.

[45] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1371–1385.

[46] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News* 44, 2 (2016), 33–47.

[47] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. 2017. Co-run scheduling with power cap on integrated cpu-gpu systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 967–977.

[48] Tsahee Zidenberg, Isaac Keslassy, and Uri Weiser. 2012. Multiamdahl: How should i divide my heterogenous chip? *IEEE Computer Architecture Letters* 11, 2 (2012), 65–68.