VisSched: An Auction based Scheduler for Vision Workloads on Heterogeneous Processors

Diksha Moolchandani, *Graduate Student Member, IEEE,* Anshul Kumar, José F. Martínez, *Senior Member, IEEE,* and Smruti R. Sarangi, *Member, IEEE*

***Abstract*—With the growth of edge computing, application-specific workloads based on computer vision are steadily migrating to edge cloudlets. Scheduling has been identified to be a major problem in these cloudlets. In this paper, we propose a generic architectural solution, VisSched, that leverages the fact that most vision workloads share similar code kernels (such as library code for linear algebra), and as a result they tend to exhibit similar phase behavior. This allows us to create an auction theory based scheduling mechanism, where we give each thread a replenishable virtual wallet, and threads are scheduled based on the amounts that they bid for executing on a free core. We show that in 20-40% of the cases, our scheduling algorithm is theoretically optimal, and in the remaining cases, it reaches a global optimum obtained using Monte Carlo simulations 90-95% of the time. Our results for the MEVBench vision workloads show a 17% higher performance and a 14% lower ED^2 as compared to the nearest competing algorithm in the literature.**

***Index Terms*—scheduling, auction theory, asymmetric multi-cores, hardware scheduler**

I. Introduction

Today's computing landscape is characterized by the following trends: traditional scaling based on Moore's law has stopped, instead of increasing the number of cores on the chip the focus has moved to creating bespoke systems, and because of the proliferation of mobile/IoT/edge-computing applications there is a lot of stress on new types of workloads. Many of these new workload suites [1], [2], [3] are tailored for a specific genre of applications such as computer vision, self driving, IoT data processing, analytics, edge computing, etc.

Unlike traditional benchmark suites such as Spec and Parsec that have very little code in common, these workload suites share code kernels to a much larger extent. For example, analytics applications share sub-routines for performing convolutions, computing dot products, and computing the result of *tanh* or *sigmoid* functions. The conventional approach to increase the performance of such workloads is to design a bespoke accelerator that can accelerate the common parts such as CNN accelerators for speeding up inferencing.

Manuscript received April 18, 2020; revised June 12, 2020; accepted July 6, 2020. This article was presented in the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems 2020 and appears as part of the ESWEEK-TCAD special issue.

Diksha Moolchandani is with the School of IT, Indian Institute of Technology Delhi, New Delhi 110016, India (email: diksha.moolchandani@cse.iitd.ac.in).

Anshul Kumar, and Smruti R. Sarangi are with the Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi 110016, India (e-mail: anshul@cse.iitd.ac.in; srsarangi@cse.iitd.ac.in).

José F. Martínez is with the School of Electrical and Computer Engineering, Cornell University, Ithaca, NY, USA (email: martinez@cornell.edu).

We look at this problem from a different angle. All such suites comprise a multitude of benchmarks that are ideally suited for different platforms: multicores, GPUs, FPGAs, and ASICs. It is envisaged that there is a high level task dispatcher that dispatches tasks to these disparate computing units, and subsequently a dedicated runtime manages the computations. For the sake of experimental evaluation, we focus on a subset of applications in these suites that are known to perform the best on multicore processors, and show that just by creating a better scheduler we can gain 17% in performance vis-a-vis the best state of the art scheduler [4]. Our scheduler requires a few performance counters and registers per core, and the scheduling logic is implemented in hardware (0.078mm^2 additional area).

1) Scope of our Work: In this paper we studied the behavior of many such newly proposed workload suites [1], [2], [3], and concluded that they share library code to a large extent. This gives us a unique opportunity to design a bespoke scheduler and also make theoretical guarantees regarding the quality of the schedule. As an example, we consider popular computer vision benchmarks in this paper that are primarily based on computational geometry, and thus do not have effective CNN implementations. Note that the scope of our work is generic and *is not dependent on a particular platform or a particular benchmark suite*. Our only requirement is that the set of workloads need to share a common set of code kernels (libraries), and their execution should be decomposable into a set of a few distinct types of phases, where the phases have the same meaning across the benchmarks – let us call such workloads *correlated workloads*.

2) Contributions:

Problem: Schedule a set of correlated workloads on a heterogeneous processor with a diversity of computational units – small cores and big cores in our case.

1) We propose a formal clustering based technique to divide the execution of benchmarks into phases. As long as these phases are correlated across the benchmarks our technique is applicable.
2) We give each thread a virtual wallet, and propose an auction theory based technique for threads to bid for cores (computational units). Threads have an incentive to save virtual money for the future, as well as win the bid.
3) We use game theory to show that in roughly 40% of practical scenarios we make optimal decisions. In the rest of the cases, our decisions are very close to optimal. To the best of our knowledge this is the first such approach for correlated workloads.

4) For the MEVBench vision workloads we achieve a 17% higher performance, and a 14% lower ED^2 as compared to the best competing algorithm [5].

II. Auction-Based Scheduling

A. Overview

In any scheduling problem we have t threads and c cores, where typically $t > c$. The threads have different requirements in terms of performance, functional unit usage, and energy usage. From the point of view of the thread it would like to execute on the fastest core that is available. However, in such cases both selfish as well as altruistic decisions are suboptimal, and can lead to starvation. Traditional graph and ILP based approaches also have limited value because they tend to take a global view of the scheduling problem. If we have a large number of tasks and a large number of potential interactions, then such problems are intractable, and cannot be solved at runtime.

Hence, instead of a centralized approach, the distributed approach is more preferable where the threads and the cores take *local decisions*; we need to ensure that in some sense the decisions are *fair* and are close to being *globally optimal*.

Fairness We can define the fairness metric as a ratio of the time that a set of threads take to execute when other applications are running, and the time it takes for them to execute in isolation. This ratio should be bounded by a small value in most cases.

Optimality In the scheduling problem, optimality is hard to establish because competing threads of different applications benefit at the expense of each other. One of the most widely used metrics/objectives is the *Nash equilibrium*.

In our algorithm our main focus is to establish that it reaches a Nash equilibrium. Fairness is *established experimentally*.

B. Nash Equilibria

A *Nash equilibrium* is a game-theoretic concept in a setting where we have multiple players (threads) with strategies and payoffs or utilities. Assume that each thread has some virtual money and the strategy is to bid a part of it to *win* (right to execute for a limited duration) a core. The utility is defined as the benefit the thread would derive by running on the core, which is typically a function of the IPC (instructions per cycle) [6].

The threads are said to be at a Nash equilibrium when given that the strategies of the rest of the threads are fixed, no thread gains by unilaterally changing its strategy. Conceptually, a Nash equilibrium is a stable operating point, which is a constrained local maxima. The equilibrium is said to be *stable*, where if one of the threads slightly changes its strategy, the rest of the threads do not deviate from their Nash equilibrium strategies. This is a local maxima. Note that in any problem we can have multiple Nash equilibria – we need to choose the one that maximizes the aggregate utility. In a game that has a single Nash equilibrium, it is said to be strictly dominant.

Note that Nash equilibria are not always global optima (e.g: the Prisoner's dilemma problem [7]); however, for the purposes of scheduling they have been found to correlate with experimentally observed optima very well [8].

C. Auction Theoretic Principles

Let thread t_i's wallet balance prior to the auction be W_i. If it bids all of it, it will not be left with any money for the future. If it bids too little, it might not win the bid. It thus needs to bid a fraction μ_i (and save the rest). This is known as the *proportional bidding scheme* in the auction theory literature, and μ_i is the *strategy* for thread i. A thread is aware of the strategies of other threads but is not aware of their wallet balance, which varies.

A simple formula for the utility of thread i is $(1-\mu_i)W_i$ if it wins, and 0 if it loses. Here, the utility of the winner is equal to the money that it saves, and the loser's utility is zero. It is obvious that the winner would like to win by bidding the least amount. If we have two threads, where W_i and W_j vary from 0 to 1 uniformly, then the globally optimal strategy is $\mu_i = \mu_j = 0.5$. In a multiplayer scenario with different distributions for each W_i, the optimal set of strategies (μ_is) needs to be computed with Monte Carlo simulations accelerated with AI algorithms.

We complicate this scenario further by adding a winner's bonus (additional utility for the winner, B_i) and an auctioneer's fee (F_i), which needs to be first deducted from the balance of the winner. The resultant expression for the utility ($\mathscr{U}_i$) of the winner is as follows.

$$\mathscr{U}_i(\mu_i) = (1-\mu_i)(W_i - F_i) + B_i \tag{1}$$

The justification for F_i, and B_i will be provided in Section VI. We also give a loser's subsidy L such that the losing thread is better equipped to bid in the future. Given this setting we use the following results from auction theory to establish optimality.

If the distributions of $(W-F)$ and B are uniform distributions and are the same, then a globally optimal Nash equilibrium exists, where the bids are proportional (PB bids) to the wallet balance (20-40% of time in our evaluations). For the rest of the cases, a Nash equilibrium does exist; however, the strategies are not necessarily proportional to the wallet balance. We experimentally show that even with proportional bids our solutions are extremely close to experimentally determined optima 90-95% of the time.

III. Related Work

The broad area of scheduling can be divided into two classes: application specific – exact task durations are known, and hence graph based approaches can be used; and general purpose – task durations can be variable, and thus heuristics are needed. We chose the best-performing prior work from the latter category for comparison.

A. Scheduling on Heterogeneous Cores

CAMP [6] runs those applications on fast cores that get maximum speedup when run on a fast core with respect to a small core. This speedup is defined as the *utility factor*, and this forms the basis of the core allocation. Similar works by Koufaty et al. [9] and Shelepov et al. [10] find the bias of an application towards cores based on its relative speedup

on a big core vis-a-vis a small core, or use profiling to pre-calculate the affinity of applications for cores. Another popular approach is *BIS* [11] that schedules bottlenecks in the code (code responsible for the maximum number of wait cycles) on the fast cores. Joao et al. [5] (*UBA*) extended this work by accelerating the lagging threads in addition to the bottlenecks identified in *BIS* such that the reduction in the total execution time is maximized. COLAB [12] uses a regression model to find the core affinity, then it schedules the threads on the basis of the bottlenecks (similar to BIS). The scheduling quantum is set with the objective of maximizing fairness. AdaMD [4] adopts an adaptive model that periodically assigns free cores to under-performing applications determined using an ML model. These approaches are unfortunately not tailored for correlated workloads.

B. Market Mechanisms for Scheduling of Heterogeneous Tasks

Wang et al. [13] proposed *XChange*, a market mechanism based on supply-demand model, in which the users bid for resources in a dynamic market setting, where the prices vary based on resource contention and utility. The approach works well when we have a choice of multiple resources in every resource allocation round (not true in our case and hence we do not compare with them). In comparison, Guevara et al. [14] provided a market mechanism to allocate a single resource, where the bids are decided based on task deadlines. This approach is only relevant for soft real time workloads.

Another work by Pereira et al. [15] allows cores to bid for the tasks, where the strategy of a core is its frequency setting. The utility is derived from the variation of the energy consumption as a result of changing the frequency setting. In comparison, we have tasks with unknown completion times and in our setting, tasks bid for a core (the reverse problem).

C. Auction Theory for Job-shop and Network Scheduling

In a typical job-shop scheduling problem we have a set of jobs and a set of resources. The jobs can have an execution order between them. The aim is to typically minimize the makespan (total execution time) of the schedule. Tang et al. [16] wrote a seminal paper in this area: Whenever a job bids for a time slot on a machine (resource), its price is decided on the basis of the number of competing bidders. A bidder has some virtual money – it needs to pay some money to run on a machine and pays a penalty if it loses the bid. It tries to maximize the amount of virtual money it holds. Zeng et al. [17] extended this work for a flexible job scheduling problem, which involves multiple sets of jobs that are mutually independent. It thus has a two-level auction process: first decide the set of jobs, and then decide the job itself (from the winning set). Bukchin et al. [18] have specialized this problem for a 2-machine scenario.

Liang et al. [19] applied auction theory for allocating resources to different service types in 5G network slicing. They formulate the problem as an auction where the bidders are the service types and the items to be auctioned are the network resources. The bid for a service type is decided on the basis of its bandwidth requirement. The payment for a resource is decided on the basis of the usage of the resource. Ding et al. [20] applied auction theory to grid computing networks. The grid resources are auctioned and the grid users bid on the basis of the price, memory and speed of the grid resources.

The flavor of all related work is more or less the same – decide a utility function and compute or adjust the bid amount based on it. Our work is however novel and different in many aspects. The differences are as follows.

Our novel contributions:

1) Our first innovation is identifying, understanding, and characterizing phases in correlated workloads. We propose a method to identify such phases at run time and effectively capture their execution characteristics (see Section IV).
2) The novelty lies in the way we create our bidding functions (combination of miss rates, core costs, cache costs), the way in which we create proportional bids, and how we use them to prove the optimality of the auction.
3) We are not aware of other works that propose a loser's subsidy and auctioneer's fee (the way we have defined them).
4) Another novel aspect of our design is storing the pre-computed optimal bid values in a pattern table and using them at run time. No other paper has characterized the search space as we do – we have demonstrated that our search space is mostly convex with localized regions that are concave.

IV. Characterization of Workloads

A. Experimental Setup

Parameter	Value	Parameter	Value
Cores	16 (10 small + 6 big)	Technology	14 nm
Big cores	6	Small cores	10
Processor Cores			
Big core		**Small core**	
Issue Width	4	Issue Width	2
Pipeline Type	Out-of-order	Pipeline Type	In-order
Frequency	3.1 GHz	Frequency	1.55 GHz
L1 i-cache, d-cache (Private caches)			
Write-Mode	Write-Back	Block Size	64 bytes
Associativity	4	Size	32 KB
Latency	4 cycles (for Big core),	2 cycles (for Small core)	
Shared L2			
Write-Mode	Write-Back	Block Size	64 bytes
Associativity	8	Bank Size	256 KB
Latency	20 cycles	#Banks	16
Main Memory and NoC			
Latency	200 cyc.	Memory Controllers	2
NoC	2D torus	Flit Size	16 bytes
Routing	X-Y	Router + Hop latency	3 cycles

TABLE I: Details of the baseline system (source [21])

We used a **full-system cycle-approximate** architectural simulator, *Tejas* [22], for all the simulations. It has been rigorously validated with native multicore hardware (for both in-order and out-of-order processors). It is bundled with the popularly used Orion 2.0 and McPat 1.0 tool sets for simulating power and energy of the NoC and cores respectively. We simulate the MEVBench suite (e.g: feature extraction, corner detection, decision trees, augmented reality) of vision benchmarks (adapted for multicores) [1], and assume an edge-computing setting, where we simulate a bespoke system

running on the edge that receives requests from mobile users, runs vision workloads on them, and returns the result. We collected traces from the Qemu virtual machine to simulate the networking stack of the Linux 4.2 kernel.

Our simulation setup (see Table I) is similar to Intel's QuickIA architecture, where we have two kinds of cores: a large out-of-order core and a small in-order core (typically used for vision benchmarks). The small core's frequency is half of the big core's frequency in our setup (similar to [23]).

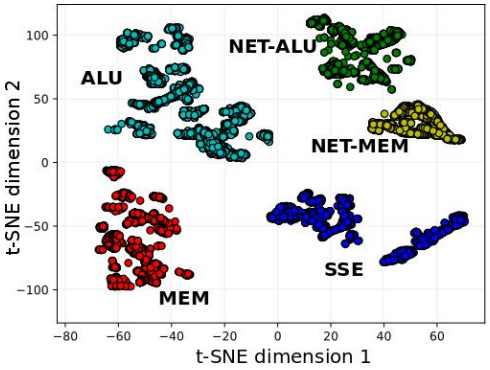

Fig. 1: Clustering of all the phases across the benchmarks

We define **performance** as a quantity that is inversely proportional to the simulated execution time of a single benchmark. For a set of benchmarks, it is inversely proportional to the instruction throughput. We also evaluate fairness that captures latency-sensitivity, contention, and starvation.

B. Characterization of the MEVBench Suite

1) Phase Behavior: The aim here is to establish that in correlated workloads we have a *few distinct types of phases*. We start with defining the following instruction classes: ALU, SSE (vector instructions), memory, instructions in the networking code (ALU and memory). We divided the execution of each workload into chunks of 100k-cycle intervals, and collected 5000 such data points per workload. These data points across the workloads were represented as a multi-dimensional vector, where each dimension contains the percentage of a particular class of instruction. We then classified these vectors using K-Means clustering, and then visualized the clustering results in 2-D by projecting the vectors to a lower dimension using the t-SNE algorithm. Figure 1 shows five distinct clusters. They have been named on the basis of the most frequently occurring instruction class.

Similar experiments on other correlated workload suites such as Cortexsuite [3], self-driving workloads [2] yielded similar results (not shown due to lack of space). We see distinct clusters, where each cluster represents a particular phase.

a) Correlation of Phases across Benchmarks: To validate the quality of the clusters, we plot the Silhouette score for the clusters. The Silhouette score is a standard technique to calculate the efficiency of clustering. A Silhouette score of '1' indicates that the samples are correctly classified into the clusters that they are most related to. It may be observed (see Figure 2(a)) that the silhouette scores are high (roughly

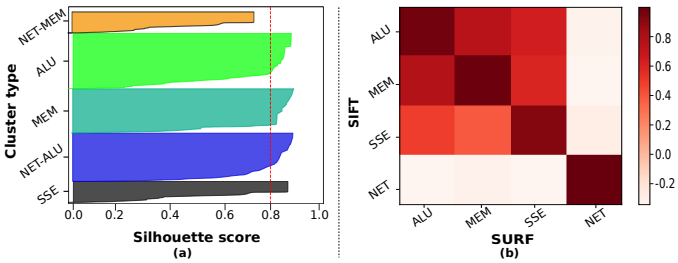

Fig. 2: (a) Silhouette plot of the clusters, (b) Phase-wise correlation of *Sift* and *Surf*

80%) indicating that the data is indeed separable into distinct clusters. Each plot is an irregularly shaped trapezoid because different points in a cluster have different Silhouette scores.

Figure 2(b) shows a representative example where we plot the pair-wise correlation between different phases of two benchmarks: *Surf* and *Sift*. We observe that intervals corresponding to the same phase behave *very similarly* (correlation above 0.9) across the benchmarks. We also established a correlation (0.7-0.8) between the phase and the IPC of an interval. Hence, the phase information can be used as a proxy for IPC. **Summary:** *The existence of phase behavior and the correlation of phases across benchmarks may help us simplify our scheduling decisions.*

V. Design

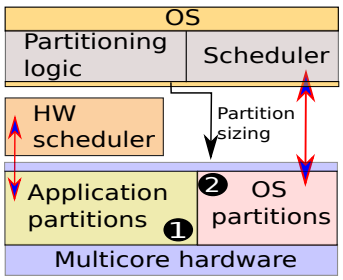

Fig. 3: Overview of the design

Figure 3 shows the overall design of the system. We modify the OS to partition the cores between the hardware scheduler and the OS scheduler (similar to [24], [25], [26]). This OS-level partitioning allows us to create a custom hardware scheduler. The OS partitions the cores into two parts such that the applications execute in the first partition and the OS routines execute in the second partition. In order to allow scheduling of the applications at fine-grained intervals, we deploy a hardware scheduler for the first partition.

We propose to use a hardware scheduler because the typical (worst-case) phase change interval for our applications is $31.25 - 62.5\mu s$, which is far lower than a typical OS time slice (1-10ms) [27]. For such fine-grained intervals, the performance benefits of scheduling are reasonable only if the context switching overheads are insignificant as compared to the granularity of scheduling. This is not the case in our setting and thus software schedulers will not prove to be useful [28], [29], [30]. Specifically, Li et al. [31] analyzed the cost of a context switch in different scenarios and quantified it to be $44.1\mu s$ to $1496.1\mu s$. This overhead is minimal for a typical

OS time slice, however for our fine-grained task scheduling, this overhead is prohibitive. Our hardware implementation is able to perform scheduling and context switching within 67*ns*, which is small enough to maintain the performance benefits of fine-grained scheduling.

Our hardware scheduler maintains two queues: one for running tasks and one for ready tasks. It also maintains the context information of the tasks (PC and register state). The scheduler manages the context, performs context switching, and invokes the auction controller to find the next task that needs to be run on a free core. If the OS wants to run some process on any of the cores in the application partition, it sends a message to the hardware scheduler. This is a standard approach followed in prior work [24], [25], [32], [33].

Our main contribution in this paper is the design of the auction controller and the algorithm it uses. We use a standard hardware scheduler design [25], [33].

VI. Implementation

We created a scheduling scheme, *VisSched*, using the key insights gathered in Section IV on a system with 10 small cores and 6 big cores (determined to be a Pareto-optimal configuration in the energy-performance space in Section VII-E). The high-level flow of the scheduling algorithm is as follows: ❶ Find an initial phase-core mapping. Since the number of phases is lower than the number of cores, each phase is allotted a set of cores (*core set*). ❷ Determine the first assignment of threads to cores. ❸ After a thread finishes its scheduling quantum on a core, predict the thread's upcoming phase, and put it in the ready queue of the mapped core set. ❹ Perform an auction for the currently freed core among the threads in the ready queue of its core set. ❺ Assign the new winner thread to this core. ❻ Go to ❸. ❼ After a certain time, recalculate the phase-core mapping.

Figure 4 shows an overview of the hardware scheduler and the structures used. These will be discussed in detail in the upcoming sections.

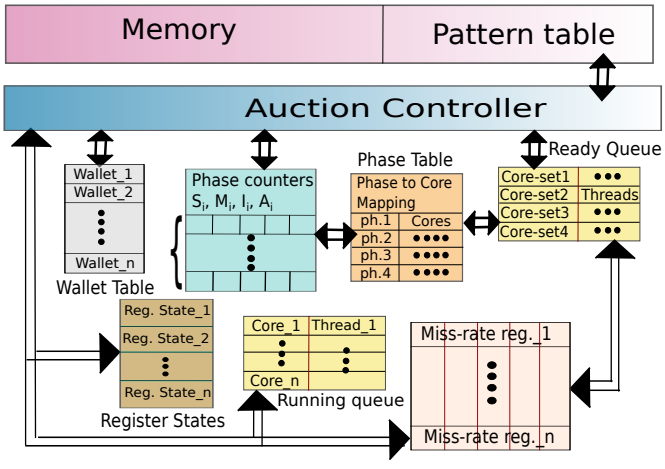

Fig. 4: Hardware Scheduler

A. Mapping the Set of Cores to Phases: Steps ❶, ❷ and ❼

We consider four distinct phases: SSE, memory, compute, and network (networking code). At the outset we map the set of cores (allotted to the application partition) to the phases in the following priority order: 4 big cores for SSE tasks; 1 big and 3 small cores each for ALU and memory tasks; and 4 small cores for network tasks. We define a 100,000 cycle interval, and based on the instruction mix we determine its phase (explained in Section VI-B). Every 100 million cycles (1000 intervals), we recompute this mapping based on the relative proportions of executed phases ($\phi_s : \phi_m : \phi_n : \phi_c$, where ϕ_s is a counter for SSE phases, ϕ_m for memory phases, ϕ_n for network phases, and ϕ_c for compute phases). The mapping information is stored in a *Phase Table* that is a 4-entry structure (one entry for each phase) with each entry being 16 bits, one per core, i.e., the i^{th} bit in this entry indicates if the i^{th} core is mapped to it. Note that *we never let a core go idle; if it does not have enough threads in its corresponding phase, we schedule a thread belonging to another phase.*

B. Predicting the Phase of the Next Interval: Step ❸

It is easy to identify the network phases on the basis of library calls or system call invocations. For the rest of the three phases, we use three instruction counters (ALU A_i, memory M_i, and SSE S_i). The weighted average of these counters over the previous five (determined empirically) intervals is used to decide the phase for the *upcoming interval*. The rules are shown in Table II in decreasing order of their preference. Here, I_i denotes the total number of instructions in the i^{th} interval. The intuition for the equation for the memory phase is that we wish to accord additional priority to memory instructions because they involve multiple operations (memory + addition), and miss rates have a large effect on performance.

Condition	Inference
$\sum_{i=1}^{5} \frac{M_i}{I_i} > \sum_{i=1}^{5} \frac{A_i+S_i}{2I_i}$	Memory phase
$\sum_{i=1}^{5} \frac{S_i}{I_i} > \sum_{i=1}^{5} \frac{A_i}{I_i}$	SSE phase
default	Compute phase

TABLE II: Heuristics for phase prediction

Since the total number of instructions per interval can vary, weighting the counters with the number of instructions in the respective interval increased the accuracy of our results (shown in Table VII in Section VII-D).

C. Auctioning process: : Steps ❹ and ❺

Table III shows the mapping of terms between our equation for utility (Equation 1) and the architectural metrics.

Broad principles: ❶ A memory-bound thread should be discouraged from running on a faster core. We found that in our suite of benchmarks, the marginal utility of running such threads on fast cores is minimal. ❷ A CPU-bound thread should be encouraged to run on a faster core. ❸ There should be no starvation. ❹ Each waiting thread should have an incentive to bid and win. ❺ Threads should not overspend on the bidding process such that they can use the remaining amount in future auctions. These are in accordance with standard auction rules.

We now discuss the intuition behind the mappings: auctioneer's fee (F), bonus (B), wallet balance (W), loser's subsidy

(L), and the bidding strategy ($\mu_1 \ldots \mu_N$) (N is the number of bidders).

Meaning	Example	*VisSched*
Balance	W	W
15 bits, Initialized to $W_{base} = 5000$		
Auctioneer's fee	F	κC
κ is a constant, C is a combination of the core cost and cache cost		
Bid	$\mu(W-F)$	$\mu(W-\kappa C)$
Bonus	B	$\lambda \times IPC/energy(mJ) + W_{base}$
The winner's bonus is IPC/energy (normalized with a constant λ + a constant amount (W_{base}))		
Utility	$(1-\mu)(W-F)+B$	$(1-\mu)(W-\kappa C)$ + $\lambda \times IPC/energy(mJ)$ + W_{base}
Loser's subsidy	L	L
$L = 0.1 * W_{base} \Rightarrow L = 500$		
New wallet balance	$W_{winner} = (1-\mu)(W_{old}-F)+B$ $W_{loser} = W_{old} + L$	
Parameter Values	$\lambda = 100$, $\kappa = 500$, $C_B = 1000$, $C_L = 250$	

TABLE III: Terms and definitions

1) The Auctioneer's Fee (F): The Auctioneer's fee captures the cost of migrating a thread from one core to another core. We model two kinds of costs: the core cost and the cache cost. The core cost captures the overhead of a context switch. If a thread is bidding for the core on which it was previously running, then the core cost is zero. Otherwise it is C_B for the big core, and C_L for the small core, where $C_B = 4 * C_L$ (ratio of IPCs on the two cores observed in our simulations).

The cache cost is zero if a thread is bidding for the same core on which it previously ran, otherwise it is meant to be commensurate with the cycles that are lost in cache misses when the thread is migrated to a new core. We formulate the cache cost as a sum of three components: L1 cost, L2 cost, and I-cache cost. The L1/I/L2 cost is the arithmetic mean of the L1/I/L2 cache miss rates recorded over the last five phases (stored in a shift register). We discourage a thread with high L1/I miss rates from migrating because regardless of its behavior (steady on a core or migrating), it is nevertheless memory-bound at the moment and thus should not move to a new core. Similarly, we can justify the I-cache cost. Analogously, high L2 miss rates indicate that it is memory-bound and thus should be given a chance to benefit from enhanced locality at the L1 level.

The final cost, C, is a sum of the core and cache costs (appropriately normalized using the constant κ).

2) Bonus: IPC vs IPC/Energy: The bonus (defined in Section II) can be made proportional to the thread's IPC (as in previous works) because intuitively it is the advantage that a thread will derive if it gets access to a core. However, this would lead to unfair schedules in our setting of heterogeneous multicores because the scheduler will only prefer big cores. Note that the average energy usage in a small core is 60-70% lower than a big core (results not shown due to a lack of space).

We thus make the bonus proportional to $IPC/energy$, which is not decidedly better for any particular type of core in our simulations (also observed by [34], [35], [36]). It still captures our original intention because we find a positive correlation between IPC and $IPC/energy$ values for 10/11 benchmarks (values range from 0.8 to 0.99). *Note that if the correlation between two sets of values is between 0.75 and 1, the sets are considered to be highly correlated.* The only exception is *Sift* because of the preponderance of SSE instructions.

3) Wallet balance (W), and Loser's subsidy (L): We initially assign a wallet balance to every thread, say W_{base}, to bid for their first auction. For the subsequent auctions, this wallet balance varies depending on whether the thread has won or lost the last auction. For a winning thread, the new wallet balance is equal to its utility in the last auction round. Recall that the utility in a round is the sum of the remaining wallet balance (savings) and the bonus in that round (see Equation 1). Thus to maximize the utility, each bidder will want to maximize its savings to bid higher in the next round (ensuring ❺), and will also be tempted to get the bonus amount (ensuring ❹). The wallet balance of the loser is updated by adding a loser's subsidy (L) to its old balance. This prevents starvation in the long run (ensuring ❸). We formally prove that our algorithm is theoretically starvation-free in Section VI-E.

D. Auctioning process: The working

We propose to use a hardware structure called an *auction controller* and a small region in memory called the *pattern table (PT)* to perform the tasks explained in Section VI-C.

1) Auction Controller: An auction is triggered whenever there is a phase change in some thread or when the thread at the head of the ready queue waits for more than 10 ms (10 $\times$ length of a scheduling quantum in Linux). A dedicated unit, the auction controller performs the auction, and finds a new task to run on the free core, and sends this information to the hardware scheduler to schedule the task on the core.

When a thread changes its phase, the core sends a message to the auction controller via the NoC. This contains the id of the core (4 bits), the thread id (5 bits), IPC in the last phase (1 byte in fixed point format), the energy consumed in μJ (4 bytes) in the last phase, and the costs ($3*40$ bits). The auction controller maintains a wallet table that is a 32-entry table that contains the wallet balance for each thread (starting balance: W_{base}). It gets the list of competing threads from the ready queue of the core-set. After deducting the scaled cost κC from the wallet balance W of each competing thread, it reads the strategy ($\mu_1 \ldots \mu_N$) from the pattern table (PT), computes the bids for each bidding thread, and performs the auction. The winning thread's wallet balance is updated as per the rules in Table III. Note that the bonus is credited to the winner's account when it goes for auction the next time because of its dependence on IPC and energy in the previous phase.

2) Pattern Table (PT) for Storing Strategies: Consider the general problem where we have k threads bidding for a single core. Each thread would like to maximize its expected utility (see Equation 1) when $(W-F)$ and B vary according to predefined distributions (see Section II). Given k pairs of these distributions for k threads, the aim is to compute the strategy $\mu_1 \ldots \mu_k$ such that the system reaches a Nash equilibrium. If there are many equilibria we need to choose the one that maximizes the aggregate utility.

Since this problem is computationally hard, a practical approach to solve this problem is to store pre-computed strategies in a lookup table. Indexing this lookup table is non-trivial because we need to find a way to compactly represent a pair of distributions. To solve this problem, let us leverage the fact that we have 4 types of well-defined phases in our program, and each pair of distributions ($W-F$ and B) is expected to be very well correlated with the type of the phase (also established experimentally). We can thus use the 2-bit phase id as a proxy for the pair of distributions. Hence, in an example system with 8 bidders, we create a 16-bit (2 bits per bidder) vector that stores the phases of all the bidding threads.

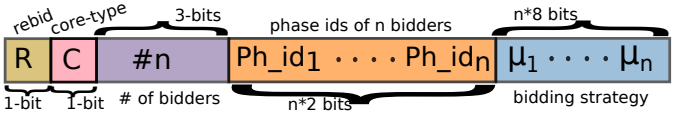

Fig. 5: A row of the PT (pattern table)

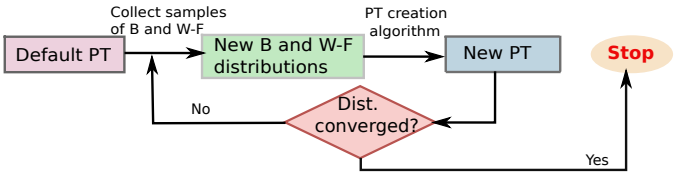

Fig. 6: Method of generating the PT

We can further generalize this representation by storing one additional bit (*core-type* bit) to represent the core type: big or small. If the thread that was running on a core decides to rebid immediately, let us store its phase id as the first phase id. To compute the migration cost correctly we store one more bit to indicate if the thread that was just running is rebidding or not (*rebid* bit). We also need 3 bits to indicate the number of competing bidders (between 1 to 8). We thus have a 21-bit vector, which fully describes an 8-bidder system (see Figure 5). For this system we need to compute the Nash equilibrium strategy: $\mu_1 \ldots \mu_8$. We thus create a table that stores the strategy for each unique combination of inputs. Even though we have 21 bits we need not have 2^{21} rows in the PT. After taking the unique combinations into account, we calculated the size of the table to be 32 KB.

a) Populating the Pattern Table: To populate each row of the pattern table, we need to know the distributions of $(W-F)$ and B for each phase-$\langle$core type$\rangle$ combination. Then we can compute the strategy: $\mu_1 \ldots \mu_8$. This is done *periodically at run-time* (see Figure 6). We first start with a default PT ($\mu = 0.5$ for all the combinations) and then iteratively arrive at a stable PT. The steps are as follows: ❶ Collect 1000 samples of B and $(W-F)$ for each phase-$\langle$core type$\rangle$ combination. ❷ Run the PT creation algorithm (say once a day when the servers are undergoing routine maintenance) ❸ Check if the distributions for B and $(W-F)$ converge, else go back to step ❶. The process stops after the distributions converge.

For step ❷, we need to find a solution that is at the Nash equilibrium for each entry of the PT – the computations for each entry can be done concurrently. We use a variant of a simple hill climbing based approach where we greedily move towards a local minima. We start at a random solution S $(\mu_1 \ldots \mu_k)$, compute its divergence and then keep randomly perturbing a μ value (keeping other μ values unperturbed) by 0.01 till the divergence stops reducing (local minima). The *divergence* $\mathscr{V}$ is defined as follows. First, let us define the utility $\mathscr{U}_j$ for the j^{th} thread in solution S as per Equation 1. Let us replace μ_j in S with a set of values in $(0,1)$ in steps of 0.01, and find the largest utility $\mathscr{U}'_j$ (for solution S'). If $\mathscr{U}'_j > \mathscr{U}_j$ we set the deviation $D_j = \mu'_j - \mu_j$, else we set it to 0, where μ'_j is the value of μ for the j^{th} bidder in the solution S'. The divergence $\mathscr{V}(S) = \sum_j D_j$; the lower the divergence, the better it is; it is 0 at a Nash equilibrium. We define the *aggregate utility* as the sum of the individual utilities.

We again start at a new random point and repeat the same process. The solution converges to the Nash equilibrium after 150-200 iterations. Populating the entire PT will take 16 minutes on the simulated asymmetric multicore system, and collecting all the samples of $W-F$ and B takes less than a second. We typically require 3 iterations to reach a convergent distribution. These iterations can be distributed over time to introduce minimal disruption in a running system.

In most cases, these distributions were well-approximated as uniform distributions (verified using the KS test [37], our distribution shows a D-value of 0.15-0.20). Generally, as long as these distributions can be weakly approximated ($0 < D < 0.5$) as a uniform distribution, our algorithm gives near-optimal results (true for most workloads).

E. Starvation

The new wallet balance of the winner thread is as follows.

$$W_{new} = (1-\mu)(W_{old} - F) + B$$

If $W_{old} \gg F$ and $W_{old} \gg B$, then $W_{new} < W_{old}$ because $0 < \mu < 1$. This means that there is a wallet threshold τ such that if $W_{old} > \tau$, the wallet balance decreases after a thread wins.

We claim that the sum S of wallet balances in a k-thread system is bounded. If this is true, then we cannot have a perpetual loser (starvation) because its wallet balance will approach infinity due to the continuous accumulation of the loser's subsidy, L. Hence, assume our hypothesis is false. Consider a point where $S \gg F$, $S \gg B$, $S \gg L$ and $\forall i, S \gg \tau_i$. Here, τ_i is the wallet threshold for thread i.

There has to be one thread i whose wallet balance is at least S/k. Either it wins the next bid or another thread, j, wins the next bid whose balance satisfies the inequality $(1-\mu_j)W_j > (1-\mu_i)S/k$. We ignore F and B here because they are insignificant. In both cases, the wallet balance of the winning thread is far greater than the thresholds τ_i and τ_j. Hence, the sum of balances will decrease by $max(\mu_i W_i, \mu_j W_j) - (k-1)L$. This is strictly positive because of our assumptions. Thus, we can choose a value of S above which the sum of balances will decrease in every round *regardless of the winner*. Let this be S_τ. The sum can only exceed this value because of the bonus and loser's subsidy in the previous round, which is bounded by $B + (k-1)L$. In the current round, the sum will again decrease by an amount that is far great than this number. We can thus claim that the sum of wallet balances is bounded by $S_\tau + B + (k-1)L$. **This proves that we have no perpetual loser**.

VII. Evaluation

We evaluate the workloads on a 16-core system with moderate loading of 1.25X (25% more threads than cores). Over-subscription has been used to evaluate the efficacy of scheduling in prior work [21], [38], because if we have an excess of resources, there is no need for efficient scheduling. For all our evaluations, the optimal number of threads for each multithreaded workload is taken from reference [1]. We have 7 multithreaded benchmarks and 4 single-threaded benchmarks (*Boost*, *ObjRec*, *FaceDet*, *AugRel*). The latter are run only as a part of a bag.

To ensure that all the schemes under consideration are on a level playing field, we ensure the following: ❶ all the schemes are scheduled with the same granularity/time quantum, ❷ the baseline multicore architecture is the same, ❸ all the schemes are evaluated with over-subscription, ❹ we simulate *all the time and energy overheads*, ❺ wherever possible, we augment the existing schemes to make them better and more suitable for the vision workloads, ❻ when we use the term *energy*, it refers to the energy consumed by the entire system.

A. Performance and ED^2

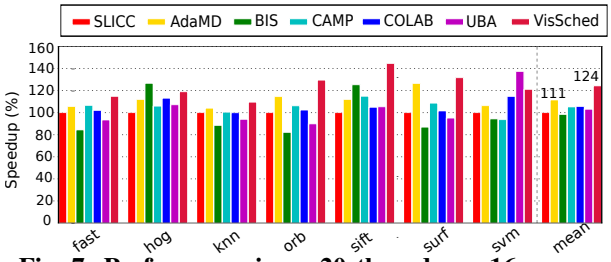

Fig. 7: Perf. comparison: 20 threads on 16 cores

1) Homogeneous Workloads: We compare the performance (defined in Section IV-A) of *VisSched* with six state-of-the-art scheduling schemes (see Figure 7): *SLICC* [38], *CAMP* [6], *BIS* [11], *UBA* [5], *AdaMD* [4], and *COLAB* [12]. Table IV shows the details of competing work. We have also highlighted (in bold) the augmentations that were done for *UBA* and *BIS* (also see the third column). Note that in general, comparing any two scheduling schemes is hard because there are multiple parameters at play. Nevertheless, the following reasons make our scheme perform well: ❶ we recognize the existence of distinct phases and their homogeneous behavior across the workloads in the suite, ❷ we leverage this behavior to propose an auction theoretic framework, ❸ in most cases our schedule is near-optimal, ❹ our suite has no clearly identifiable bottlenecks such as barriers, which are exploited in the competing works [5], [11], ❺ using a replenishable virtual wallet allows us to ensure better fairness and get productive mappings.

On an average, we perform 13% better than the best of the other schemes for a load of 1.25X oversubscription. Figure 7 shows the speedup when 4 *instances* of the same multi-threaded workload (4 instances, 5 threads each) are scheduled on a 16-core system. We need 5 threads to accommodate 4 child and 1 master thread [1]. Let us analyze the two exceptions: *HoG* and *SVM*. The auction results (see Table VII) for both the benchmarks show that they have the least migration to auction ratios. This is an artefact of our scheduling algorithm, which never prefers to keep a core *idle*, and retains the previously executing thread even if it has changed its phase – this leads to an unproductive mapping even though the core utilization is improved. We see this effect in *HoG* and *SVM*, because the distribution of phases across the threads is very non-uniform at any given point in time.

Technique	Functionality	Remarks
SLICC [38]	Exploits the similarity of code based on the i-cache footprints.	No utility metric to exploit the asymmetry of cores
CAMP [6]	Calculates the utility of every application to schedule on asymmetric cores.	Utility is not aware of the thread behavior
BIS [11]	Accelerates bottlenecks. Our augmentations include accelerating the longest waiting **phases***.	Our applications do not have clearly identifiable bottlenecks
UBA [5]	Accelerates bottlenecks, and lagging threads with a high utility metric. Our augmentations include the acceleration of **phases*** with high utility metrics.	
AdaMD [4]	Accelerates the under-performing threads found using an ML model	Not aware of the thread behavior
COLAB [12]	Enqueues the tasks on the basis of their core affinity but schedules them on the basis of bottlenecks	Our applications do not have clearly identifiable bottlenecks
** Phases in BIS and UBA refer to the phases that we identified in Section IV-B1*		

TABLE IV: Description of the competing techniques

2) Bag-of-task Workloads:

	fast	hog	knn	orb	sift	surf	svm	boost	objrec	facedet	augrel
Bag1		✓	✓		✓		✓				
Bag2	✓		✓	✓			✓				
Bag3	✓		✓			✓	✓				
Bag4	✓			✓	✓	✓					
Bag5	✓			✓		✓	✓				
Bag6		✓				✓	✓	✓	✓		
Bag7		✓				✓		✓	✓	✓	✓

The first five configurations contain multi-threaded workloads (5 threads each) and the last two configurations contain a mix of multi-threaded and single-threaded workloads in the bag. Figure 8 shows the performance of a heterogeneous bag-of-tasks based workload. We outperform the next best technique (*AdaMD*) by 17%. Let us analyze a few representative benchmark specific trends. The only difference in Bag4 and Bag5 is that *Sift* is swapped with *SVM*. However, Bag5 has a higher speedup (4%) because of the presence of *SVM* that has a smaller working set and hence a lesser migration cost. This gets accounted for in its penalty and hence it migrates at a faster rate between cores with lesser degradation in IPC as compared to *Sift*.

Table V shows the L2 MPKI and L1 hit-rates for all the bags under different scheduling schemes. The numbers in Table V and Figure 8 are very well correlated. A high performance is a consequence of low L2 MPKI and high L1 hit-rate. The only exception is Bag7. In Bag7, *UBA* has the lowest L2 MPKI and is expected to perform well, however it is not able to capture the locality of data in the L1 cache and has a low L1 hit-rate. Thus, it performs sub-optimally.

Finally, we compare the ED^2 of all the scheduling algorithms for the bag-of-task workloads in Figure 9 and observe a very good correlation with Figure 8. We achieve 14% lower ED^2 than the closest competing scheme, *AdaMD*. This is primarily because we optimize for both energy and performance.

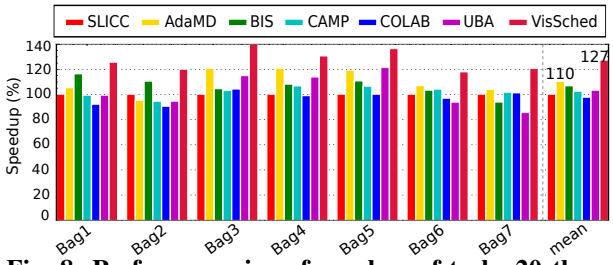

Fig. 8: Perf. comparison for a bag-of-tasks:20 threads on 16 cores

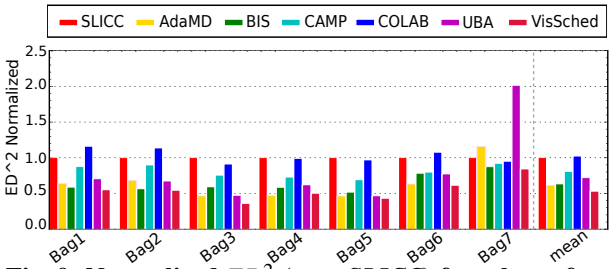

Fig. 9: Normalized ED^2 (w.r.t *SLICC*) for a bag-of-tasks:20 threads on 16 cores

Proposal	Bag1		Bag2		Bag3		Bag4		Bag5		Bag6		Bag7	
	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate	L2 MPKI	L1 hit rate
SLICC [38]	10.5	89.5	6.86	89	8.32	92.5	9.22	93.46	8.23	94.11	9.44	94.07	9.72	91.86
AdaMD [4]	8.4	91.6	5.23	92.4	4.6	92.8	6.2	94	5.1	95.5	6.49	96	8.12	91.3
BIS [11]	9.55	91.8	4.94	92.6	5.77	94.7	6.43	94.2	5.5	96.07	8.95	94.4	8.89	91.25
CAMP [6]	10.67	90.41	7.27	90.76	7.59	93.23	8.61	94.07	6.95	94.32	8.81	94.14	8.89	91.25
COLAB [12]	9.85	90.13	6.16	90.5	5.67	91.4	7.47	93.3	6.23	94	8.39	93.9	7.83	91.9
UBA [5]	9.52	90.28	5.70	91.64	5.04	92.74	6.32	94.92	5.61	95.89	8.05	94.45	4.78	88.29
VisSched	**7.76**	**92.08**	**4.5**	**91.05**	**4.44**	**94.14**	**6.94**	**95.17**	**4.95**	**96.08**	**7.4**	**95.21**	**7.67**	**91.34**

TABLE V: Cache statistics for the bag-of-task workloads

B. Fairness

We use the same fairness metric as used in XChange [13]. First, let us define the slowdown ratio (SR) – ratio of IPCs of the threads of the same application when run in a bag, and when run individually. The fairness is defined as SR_{min}/SR_{max} for the threads in a bag (should ideally be 1). Closer is the fairness to 1, the better it is. As shown in Figure 10, *VisSched* has the highest fairness (mean: 0.6); all the other schemes saturate at 0.49. The primary reason for a good fairness is a better scheme for managing the bidding capacity (B and L).

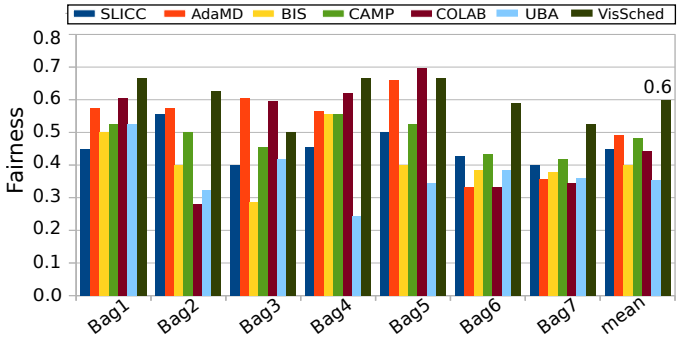

Fig. 10: Fairness for the bag-of-task workloads

C. Setting the Parameters

We use six constants in our algorithm: W_{base}, λ, κ, C_B, C_L ($C_B : C_L$ ratio has already been fixed to 4 based on our observations from the experiments), and L. One of the constants has to be set arbitrarily, and the rest have to be calibrated with respect to it, because in our formulation only the relative values matter. We use the concept of Nash equilibria to set these parameters.

We consider these parameters as players and the payoff is defined in the same way for all the players – the net performance of the system. The Nash equilibrium point represents a stable point where unilaterally changing any parameter does not lead us to a better performance. In fact we found that this parameter combination is very stable and deviations in the values of the parameters gave us a solution where we were strictly worse off.

D. Other Statistics

1) VHDL Synthesis Results and Area Overheads: The area overheads for the additional hardware structures are shown in Table VI. In a typical $200-300mm^2$ die, the area overhead of the hardware scheduler is very little: $0.078mm^2$. The overheads of the rest of the registers and performance counters are negligible.

Mapping counters	4*10 bits (4 for $\phi_s, \phi_m, \phi_c, \phi_n$)
Phase counters (phase prediction)	4*5*20 bits per application (4 for S_i, M_i, A_i, I_i, 5 for history)
Miss-rate shift register	3*40 bits per application (3 for I,L1,L2)
Phase table	4*16 bits (for 16 cores)
Wallet table	32*15 bits (for 32 applications)
ahr	5 bits
Hardware Scheduler	$0.078mm^2$
Tool	Cadence RTL Compiler UMC 14 nm (scaled using [39])

TABLE VI: Area overheads of the hardware structures

Bench-mark (1)	Phase pred. rate(%) (2)	Same phase auctions for 1.25X(%) (3)	# bids lost (avg,max) (4)	# core migr--ations (5)	# auctions (6)
Orb	85	25	(4,13)	987	2545
KNN	91	40	(4,21)	1024	3514
Fast	93	26	(4,55)	797	3144
Sift	90	22	(5,33)	322	2633
Surf	92	28	(4,12)	610	2427
SVM	87	12	(2,6)	12	1506
HoG	88	22	(5,37)	161	2564

TABLE VII: Phase prediction, and the auction process for 200M instruction run

2) Phase Characterization and Distributions: The second column of Table VII shows the phase prediction accuracy for all the simulated intervals in each benchmark. We predict the phase using the heuristics shown in Table II, and then compare it with the phase obtained by finding the cluster that an interval belongs to (based on its instruction mix). Recall

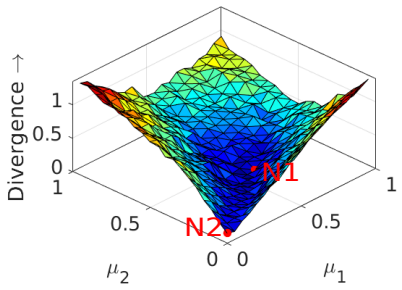

Fig. 11: Search space for 2 bidders

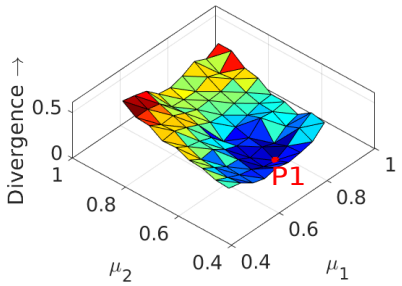

Fig. 12: Constrained search space for 4 bidders

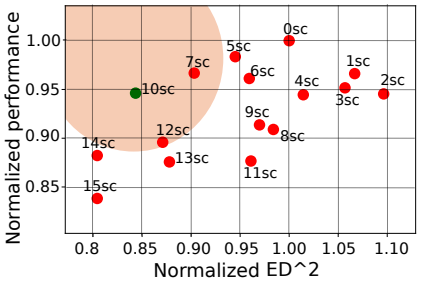

Fig. 13: Sensitivity of VisSched w.r.t/ bid and small core counts

that we characterized clusters based on the instruction mix, and finding the membership of an interval is tantamount to finding the cluster that has the closest instruction mix. The phase prediction accuracy ranges from 85-93%.

3) Results for Hill Climbing: Now as discussed in Section II, for the case of k symmetric bidders (same phase) a unique Nash equilibrium exists, and the bids are a linear function (PB bids) of the wallet balance. We find that in roughly 20-40% of the cases (column (3) in Table VII) we have this case. In these cases, we can compute the optimal solution analytically ($\mu = (k-1)/k$), and hill climbing need not be used. For the general case of asymmetric bidders, let us characterize the search space.

A greedy algorithm like hill climbing always reaches the global minima if the surface of the search space is *convex*. A convex surface has a positive second derivative. Alternatively a function f is convex if for all points x_1 and x_2 in its domain, and $\forall \lambda \in (0,1)$: $f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2)$. Let the LHS be η_1 and the RHS be η_2; we can treat the ratio $\omega = (\eta_1 - \eta_2)/\eta_1$ as a measure of the non-convexity of a point. For example, the inner surface of a cup is a convex surface; however, if there is a small bump, then the surface is non-convex at that point.

Figure 11 shows a plot of the divergence (defined in Section VI-D2) for different values of μ_1 and μ_2 (2 asymm. bidders). We can quickly conclude the following: ❶There exists a Nash equilibrium at N1(0.52,0.37), ❷there is one more local minima at N2 (0.02,0.02), which is trivial and should be ignored, and ❸the surface is mostly convex with some bumps (zoomed in for the case of 4 bidders in Figure 12, each point contains the best configuration for all values of (μ_3, μ_4)).

We performed Monte Carlo simulations with all our distributions and never found the non-convexity ratio(ω) to be more than 5.6%, and the bumps are localized, which means that even the most randomized hill climbing searches can route around such points, and are expected to reach the global minima. We compared our hill climbing results with values obtained from exhaustive simulations (4-bidders), we reached the global minima 90-95% of the time.

4) Auction Process: Columns (4), (5), and (6) in Table VII show the number of bids lost, core migrations and auctions in a sample 200 million-instruction run. An auction happens roughly once every 100k cycles, the probability of a core migration is around 25% per auction. Whenever an auction happens, we have on average 2-5 bidders (in 1.25X threads to core ratio), and because we do not have starvation, the number of times a thread can lose an auction is bounded (6-55 times).

E. Justification for the Mix of Cores

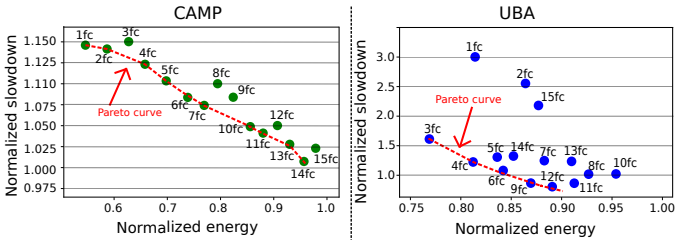

Fig. 14: Energy vs slowdown for different combinations of fast (fc) and slow cores (sc) (normalized to 16fc)

Figure 14 shows a slowdown vs energy plot for all possible mix of cores (*fc*→ fast core, and *sc*→ slow core) in a 16-core asymmetric multicore system with 1.25X oversubscription for two scheduling schemes across the spectrum: *UBA* and *CAMP*. Note that in the figure we only show the number of fast cores to improve readability, e.g, *5fc* means that we have 11 slow cores as well. The dashed line is a Pareto-optimal front. A configuration is said to be *Pareto optimal* if there is no other configuration that uses lower energy for the same slowdown, or has a lower slowdown for the same energy, or both. For both the plots we computed an intersection between the Pareto optimal sets and found two configurations: *(6fc,10sc)* and *(4fc,12sc)*. We chose *(6fc,10sc)* because of its lower slowdown.

F. Sensitivity to the Big and Small Core Counts

We have shown the sensitivity results for big/small core counts in Figure 13. A small core is denoted by *sc* and a big core is denoted by *fc*. The Pareto-optimal front is shown in the figure. Any point towards the top-left is a "good point". It has a high performance and a low ED^2. The point closest to the top-left is the *(10sc,6fc)* combination that we have chosen.

G. Extending VisSched to include DVFS

We extended our scheme to incorporate basic DVFS mechanism with two V-f levels (similar to [40]) – (1.5 GHz, 0.9 V) and (3.1 GHz, 1.25V) [41]. Our scheme can be very easily extended to also incorporate other DVFS techniques and more DVFS settings.

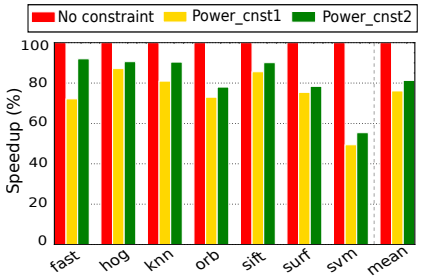

Fig. 15: Perf. comparison of VisSched with DVFS under different power constraints

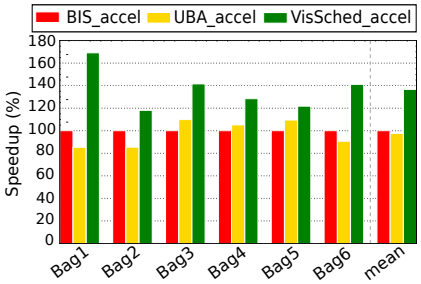

Fig. 16: Perf. comparison of VisSched with state-of-the-art (accelerator included in all the schemes)

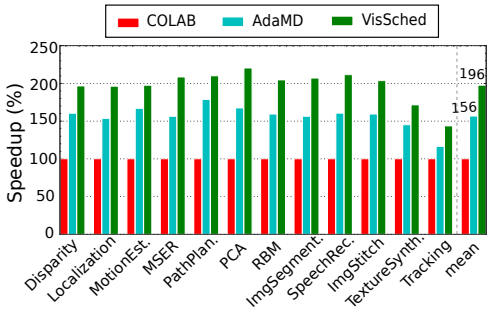

Fig. 17: Performance of Cortexsuite [3] and self-driving applications: 20 threads on 16 cores

The quintessential use of DVFS is as follows. We define a **power constraint for each thread**. The threads are regularly monitored (every 100,000 cycles): if a thread's power consumption is 33% lower than the power constraint and there exists a higher voltage-frequency level, the underlying core may switch to that level. Similarly, if a thread's power consumption violates (exceeds) the power constraint, it is switched to a lower DVFS setting. This is the conventional mechanism; it needs to be modified in our case. The aim is to maximize performance subject to a power constraint.

Since the scheduling granularity in VisSched is 100,000 cycles ($\approx 30\mu s$), which is similar to the time required to switch between two DVFS levels, it is inefficient to perform a voltage-frequency change on every context switch. Hence, we add a constraint that the thread should have executed on the same core for at least three consecutive phases before we change the V-f level (DVFS setting). Second, while bidding we incorporate a cost that is proportional to the difference in the two frequencies – current frequency of the core that we are bidding for, and the last safe frequency (at which the power constraint was not violated) for the thread for the given core type. If the cost is 0, then it means that we do not need to change the DVFS settings (this is desirable).

In Figure 15, the first bar corresponds to the default case (no power constraint). We consider two power constraints – *Power_cnst1* and *Power_cnst2* – per thread. All the workloads run at the higher setting to begin with. *Power_cnst1* corresponds to roughly 2W per thread and is a tighter constraint than *Power_cnst2* that corresponds to roughly 4W per thread. We achieve a performance degradation of 24% at 2W and 19% at 4W as compared to an unconstrained system. Table VIII shows the percentage of the number of times the power constraints are *violated* (averaged across all the threads). In all such cases we transition to the lower setting. This ensures that the overall power consumption of a thread across any substantial window of time ($>$ million cycles) is within the power constraint (also experimentally verified). In almost all DVFS schemes such *transient violations* in the power constraint do occur and that is why we need to transition to a lower DVFS setting.

H. Extending VisSched to include a LACore accelerator

The easiest way to incorporate a custom hardware in VisSched is to allow the SSE phases to run on custom accelerators.

Benchmark	% Violation at 2W	% Violation at 4W	Linear algebra kernel
Fast	24	8.7	convolution
HoG	20	14	vector add.
KNN	15	4.6	vector add.
Orb	24.1	9.27	convolution
Sift	12	7.52	vector mult.
Surf	25	13	vector operations
SVM	6.38	2.9	dot product

TABLE VIII: Average number of violations of the power constraint and the linear algebra kernels

This is a valid approach because the SSE phases capture the dominance of vector instructions, which are typically a part of linear algebra kernels. We assume an accelerator similar to LACore [42] (linear algebra core) that is reported to achieve a 3.43X speedup over optimized x86 code for general purpose linear algebra kernels. We incorporate such an accelerator in our system (parameters taken from the original paper and appropriately scaled for the 14 nm technology node). We map the linear algebra kernels (shown in fourth column of Table VIII) to the accelerator.

Figure 16 shows the performance improvement of VisSched with the accelerator over UBA and BIS (with the accelerators). We did not have to modify the bidding scheme in VisSched to incorporate such accelerators. We modeled the accelerator as a specialized core in UBA and BIS with its unique features. We achieved a 36% improvement over the next best scheme, BIS (with the accelerator). The reasons for outperforming UBA and BIS in this case are the same as the scenarios in which we did not have such an accelerator – more productive mapping of tasks, and a holistic view.

I. Generality of the Scheme

We show that our contributions are not simply limited to the MEVBench suite of benchmarks. We compare our scheme with two recent schemes, AdaMD and COLAB, for Cortexsuite [3] and self-driving applications [2]. Figure 17 shows that we achieve nearly 40% performance improvement over the next best scheme, AdaMD. We also achieve 38% ED^2 improvement over AdaMD (not shown due to lack of space)". This improvement is more than MEVBench because of the presence of more same-phase auctions (40-60%) in the Cortexsuite workloads.

VIII. Conclusion

In this paper we propose a generic architectural solution called *VisSched* that leverages the phase behavior of vision workloads to create an auction theory based scheduling mechanism. For 20-40% of the cases, all the bidders are in the same phase, and thus the PB bids are optimal. For the rest of the cases, which involve asymmetric bidders, we show that the search space is nearly convex and that a non-trivial Nash equilibrium exists. Nevertheless, the Nash equilibrium in that scenario is at least a locally optimal solution, which we are able to find 90-95% of the time, and our experiments show that they yield good results. Our auction framework provides a higher degree of fairness than competing mechanisms, and at the same time delivers both an increase in performance (17%) and a decrease in ED^2 (14%). We also show that DVFS schemes and the accelerators can be incorporated trivially in our framework without any significant changes to the auction process.

Acknowledgement

The authors would like to thank the anonymous reviewers for their valuable comments. They thank Sudhanshu Gupta of University of Rochester for helping with the characterization experiments. They thank Dr. Rajshekar Kalayappan of IIT Dharwad and Priyanka Singla of IIT Delhi for helping with the Tejas simulator and providing feedback on various parts of the paper.

References

[1] J. Clemons, H. Zhu, S. Savarese, and T. Austin, "Mevbench: A mobile computer vision benchmarking suite," in *Proc. IEEE Int. Symp. on Workload Characterization (IISWC)*, 2011, pp. 91–102.

[2] Y. Wang, S. Liu, X. Wu, and W. Shi, "Cavbench: A benchmark suite for connected and autonomous vehicles," in *Proc. IEEE/ACM Symp. on Edge Computing (SEC)*, 2018, pp. 30–42.

[3] S. Thomas, C. Gohkale, E. Tanuwidjaja, T. Chong, D. Lau, S. Garcia, and M. B. Taylor, "Cortexsuite: A synthetic brain benchmark suite," in *Proc. IEEE Int. Symp. on Workload Characterization (IISWC)*, 2014, pp. 76–79.

[4] K. R. Basireddy, A. K. Singh, B. M. Al-Hashimi, and G. V. Merrett, "Adamd: Adaptive mapping and dvfs for energy-efficient heterogeneous multi-cores," *Proc. IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[5] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 154–165, 2013.

[6] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010.

[7] M. J. Osborne and A. Rubinstein, *A course in game theory*. MIT press, 1994.

[8] A.-H. Mohsenian-Rad, V. W. Wong, J. Jatskevich, and R. Schober, "Optimal and autonomous incentive-based energy consumption scheduling algorithm for smart grid," *Innovative smart grid technologies (ISGT)*, pp. 1–6, 2010.

[9] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. ACM 5th European conference on Computer systems*, 2010, pp. 125–138.

[10] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[11] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 223–234, 2012.

[12] T. Yu, P. Petoumenos, V. Janjic, H. Leather, and J. Thomson, "Colab: a collaborative multi-factor scheduler for asymmetric multicore processors," in *Proc. ACM/IEEE 18th Int. Symp. on Code Generation and Optimization*, 2020, pp. 268–279.

[13] X. Wang and J. F. Martínez, "Xchange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures," in *Proc. IEEE 21st Int. Symp. on High Performance Computer Architecture (HPCA)*, 2015.

[14] M. Guevara, B. Lubin, and B. C. Lee, "Navigating heterogeneous processors with market mechanisms," in *Proc. IEEE Int. Symp. on High Perf. Computer Architecture (HPCA)*, 2013, pp. 95–106.

[15] D. Pereira, A. Ilic, and L. Sousa, "On boosting energy-efficiency of heterogeneous embedded systems via game theory," in *Proc. 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, 2017, pp. 19–24.

[16] J. Tang, C. Zeng, and Z. Pan, "Auction-based cooperation mechanism to parts scheduling for flexible job shop with inter-cells," *Applied Soft Computing*, vol. 49, pp. 590–602, 2016.

[17] C. Zeng, J. Tang, Z. Fan, and C. Yan, "Auction-based approach for a flexible job-shop scheduling problem with multiple process plans," *Engineering Optimization*, vol. 51, no. 11, pp. 1902–1919, 2019.

[18] Y. Bukchin and E. Hanany, "Decentralization cost in two-machine job-shop scheduling with minimum flow-time objective," *IISE Transactions*, vol. 52, pp. 1–17, 2020.

[19] L. Liang, Y. Wu, G. Feng, X. Jian, and Y. Jia, "Online auction-based resource allocation for service-oriented network slicing," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 8, pp. 8063–8074, 2019.

[20] L. Ding, L. Chang, and L. Wang, "Online auction-based resource scheduling in grid computing networks," *Int. Journal of Distributed Sensor Networks*, vol. 12, no. 10, pp. 1–12, 2016.

[21] P. Kallurkar and S. R. Sarangi, "Schedtask: a hardware-assisted task scheduler," in *Proc. 50th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2017, pp. 612–624.

[22] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter, "Tejas: A java based versatile micro-architectural simulator," in *Proc. 25th Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2015, pp. 47–54.

[23] D. Shelepov and A. Fedorova, "Scheduling on heterogeneous multicore processors using architectural signatures," in *Proc. Workshop on the Interaction between Operating Systems and Computer Architecture*, 2008, pp. 21–25.

[24] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, "Tessellation: Space-time partitioning in a manycore client os," in *Proc. 1st USENIX conference on Hot topics in parallelism*, 2009, pp. 10–10.

[25] D. Gregorek, J. Rust, and A. Garcia-Ortiz, "Dracon: A dedicated hardware infrastructure for scalable run-time management on many-core systems," *IEEE Access*, vol. 7, pp. 121 931–121 948, 2019.

[26] M. Vetromille, L. Ost, C. A. Marcon, C. Reif, and F. Hessel, "Rtos scheduler implementation in hardware and software for real time applications," in *Proc. IEEE 17th Int. Workshop on Rapid System Prototyping (RSP'06)*, 2006, pp. 163–168.

[27] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *Proc. 39th Annual Int. Symp. on Computer Architecture (ISCA)*, 2012, pp. 213–224.

[28] X. Tan, J. Bosch, D. Jiménez-González, C. Álvarez-Martínez, E. Ayguadé, and M. Valero, "Performance analysis of a hardware accelerator of dependence management for task-based dataflow programming models," in *Proc. IEEE Int. Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2016, pp. 225–234.

[29] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: architectural support for fine-grained parallelism on chip multiprocessors," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 162–173, 2007.

[30] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 311–322, 2010.

[31] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Proc. Workshop on Experimental Computer Science*, 2007.

[32] J. Lee, C. Nicopoulos, H. G. Lee, S. Panth, S. K. Lim, and J. Kim, "Isonet: Hardware-based job queue management for many-core architectures," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 21, no. 6, pp. 1080–1093, 2012.

[33] D. Gregorek, C. Osewold, and A. Garcia-Ortiz, "A scalable hardware implementation of a best-effort scheduler for multicore processors," in

Proc. IEEE Euromicro Conference on Digital System Design, 2013, pp. 721–727.
[34] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar, "Using asymmetric single-isa cmps to save energy on operating systems," *IEEE Micro*, vol. 28, no. 3, pp. 26–41, 2008.
[35] F. Afram, *Dynamic core splitting for improving energy efficiency*. State University of New York at Binghamton, 2010.
[36] M. Malik and H. Homayoun, "Big data on low power cores: Are low power embedded processors a good fit for the big data workloads?" in *Proc. 33rd IEEE Int. Conf. on computer design (ICCD)*, 2015, pp. 379–382.
[37] H. Hassani and E. S. Silva, "A kolmogorov-smirnov based test for comparing the predictive accuracy of two sets of forecasts," *Econometrics*, vol. 3, no. 3, pp. 590–609, 2015.
[38] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos, "Slicc: Self-assembly of instruction cache collectives for oltp workloads," in *Proc. 45th Annual IEEE/ACM Int. Symp. on Microarchitecture*, 2012, pp. 188–198.
[39] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *IEEE Micro*, vol. 31, no. 4, pp. 16–29, 2011.
[40] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguade *et al.*, "Cata: criticality aware task acceleration for multicore processors," in *Proc. IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2016, pp. 413–422.
[41] M. Nejat, M. Manivannan, M. Pericàs, and P. Stenström, "Coordinated management of dvfs and cache partitioning under qos constraints to save energy in multi-core systems," *Journal of Parallel and Distributed Computing*, 2020.
[42] S. Steffl and S. Reda, "Lacore: A supercomputing-like linear algebra accelerator for soc-based designs," in *Proc. IEEE Int. Conf. on Computer Design (ICCD)*, 2017, pp. 137–144.

Diksha Moolchandani received a bachelor's degree in electronics and communication engineering from Indian Institute of Information Technology Jabalpur, Jabalpur, Madhya Pradesh, India.

She is currently a Research Scholar with the School of IT, Indian Institute of Technology Delhi, New Delhi, India. Her current research interests include architectures for computer vision, hardware design, and neural network accelerators. She is a student member of the IEEE.

Anshul Kumar received the Ph.D. degree in Computer Aided Design of Digital Systems from Indian Institute of Technology Delhi, New Delhi, India and is currently an Emeritus Professor with the Computer Science and Engineering Department at IIT Delhi. He has held visiting appointments at USC, University of Edinburgh, KTH Stockholm, and EPFL.

Prof. Kumar's current research interests are VLSI synthesis, embedded systems design methodology and high performance computer architectures and he has published more than 100 research papers in reputed journals and proceedings of refereed international conferences. He has been a consultant to Gateway Design Automation (now Cadence Design Systems), Technology Parks Ltd, ST Microelectronics and Poseidon Design Systems. Prof. Kumar has been associated with the annual International Conference on VLSI Design since its inception in 1985 and has served as its General Co-Chair in 2009.

Prof. Kumar co-founded the start-up company called Kritikal Solutions and served as its Hon. Chairman, Hon. Director and Mentor. He serves as the Technical Advisory Board Member of a start-up company VirtuQ. He received the ACM Transaction on Design Automation of Electronic Systems (TODAES) 2007 Best Paper Award.

José F. Martínez received a bachelor's degree from the Universidad Politécnica de Valencia, and MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign. He is currently Professor of Electrical and Computer Engineering and Associate Dean for Diversity and Academic Affairs in the College of Engineering at Cornell University, Assistant Director for the DARPA/SRC Center for Research in Intelligent Storage and Processing in Memory (CRISP).

Prof. Martínez was a recipient of two IEEE Micro Top Picks papers a HPCA Best Paper award, MICRO and HPCA Best Paper nominations, an NSF CAREER award, two IBM Faculty awards, and one of the inaugural Distinguished Educator awards by the Computer Science Department of the University of Illinois at Urbana-Champaign.

Prof. Martínez currently serves as an elected member of ACM SIGARCH's Board of Directors, and as General Co-chair of ISCA 2021. Previously, he served as Chair of the IEEE Computer Society's Transactions Operating Committee (2017), as Editor in Chief of IEEE Computer Architecture Letters (2013-2016), as Program Co-chair of MICRO 2009, Program Chair of HPCA 2016, and General Co-chair of ISCA 2020. He is a Senior Member of the IEEE.

Smruti R. Sarangi (M'16) received the B.Tech. degree in computer science from the Indian Institute of Technology Kharagpur, Kharagpur, India, and the M.S. and Ph.D. degrees from the University of Illinois at UrbanaChampaign, Champaign, IL, USA.

He is currently an Usha Hasteer Chair Professor with the Computer Science and Engineering Department, Indian Institute of Technology Delhi, New Delhi, India, where he holds a joint appointment with the Department of ELectrical Engineering and the School of Information Technology. He has published extensively in peer reviewed conferences and journals, holds 5 U.S. patents, and has filed 3 Indian patents. He has authored the popular undergraduate textbook on computer architecture entitled Computer Organisation and Architecture (McGraw-Hill). His current research interests include processor reliability, architectural support for operating systems, and processors for the Internet of Things. Dr. Sarangi is a member of the ACM and IEEE.