CARMA: Contention-aware Auction-based Resource Management in Architecture

Diman Zad Tootaghaj, *Student Member, IEEE*, Farshid Farhat, *Student Member, IEEE*

***Abstract*—As the number of resources on chip multiprocessors (CMPs) increases, the complexity of how to best allocate these resources increases drastically. Because the higher number of applications makes the interaction and impacts of various resource levels more complex. Also, the selection of the objective function to define what "best" means for all applications is challenging. Memory-level parallelism (MLP)-aware algorithms in CMPs try to maximize the overall system performance or equalize each application's performance degradation due to sharing. However, depending on the selected "performance" metric, these algorithms are not efficiently implemented for all applications, because these centralized approaches mostly need some further information regarding applications' need. In this paper, we propose a contention-aware game-theoretic resource management approach (CARMA) using market auction mechanism to find an optimal strategy for each application in a resource competition game. The applications learn through repeated interactions to choose their action on choosing the shared resources. Specifically, we consider two cases: (i) cache competition game, and (ii) main processor and co-processor congestion game. We enforce costs for each resource and derive bidding strategy. Accurate evaluation of the proposed approach shows that our distributed allocation is scalable and outperforms the traditional and current approaches.**

***Index Terms*—Auction Theory, Computer Architecture, Resource Management.**

I. Introduction

THE number of cores on chip multiprocessors (*CMP*) is increasing each year and it is believed that only many-core architectures can handle the massive parallel applications. Server-side *CMP*s usually have more than 16 cores and potentially more than hundreds of applications can run on each server. These systems are going to be the future generation of the multi-core processor servers. Applications running on these systems share the same resources like last level cache (*LLC*), interconnection network, memory controllers, off-chip memories or co-processors where the higher number of applications makes the interaction and impacts of various resource levels more complex. Along with the rapid growth of core integration, the performance of applications highly depend on the allocation of the resources and especially the *contention* for shared resources [1–11]. In particular, as the number of co-runners running on a shared resource increases, the magnitude of performance degradation increases. Also, the selection of the objective function to define what "best" means for all applications is challenging or even theoretically impossible to improve IPC of one application and memory latency of another application simultaneously in a system. As a result, this new architectural paradigm introduces several new challenges in terms of scalability of resource management and assignment on these large-scale servers. Therefore, a scalable competition method between applications to reach the optimal assignment can significantly improve the performance of co-runners on a shared resource. Figure 1 shows an example of performance degradation for 10 *spec 2006* applications when running on a shared 10MB *LLC* (Shared), or when running on a private 1MB *LLC* (Separate).

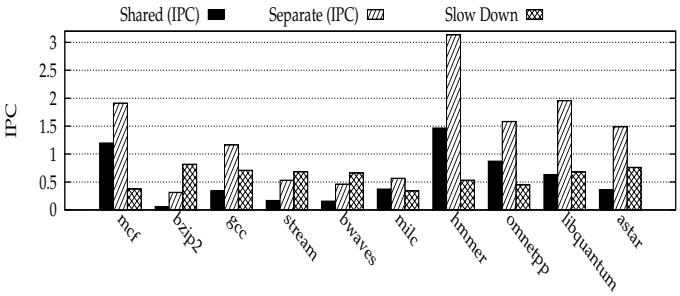


Fig. 1: Performance degradation of 10 different *spec 2006* applications sharing *LLC*.

Among these shared resources, sharing *CPU*s and *LLC*s plays an important role in overall CMP utilization and performance. Modern *CMP*s are moving towards heterogeneous architecture designs where one can get advantage of both small number of high performance *CPU*s or higher number of low performance cores. The advent *Intel Xeon Phi* co-processors is an example of such heterogeneous architectures that during run-time the programmer can decide to run any part of the code on small number of *Xeon* processors or higher number of *Xeon Phi* co-processors. Therefore, the burden of making decisions on getting the shared resources is moving towards the applications. In addition to the shared *CPU*s, shared *LLC* keeps data on chip and reduces off-chip communication costs [12]. Sometimes an application may flood on a cache and occupy a large portion of available memory and hurt performance of another application which rarely loads on memory, but its accesses are usually latency-sensitive. Recently, many proposals target partitioning the cache space between applications such that (1) each application gets the minimum required space, so that per-application performance is guaranteed to be at an acceptable level, (2) system performance is improved by deciding how the remaining space should be allocated to each one.

Prior schemes [3, 12–17] are marching towards these two goals, usually by trading off the system complexity and maximum system utilization. It is shown that neither a pure private *LLC*, nor a pure shared *LLC*, can provide optimal performance for different workloads [6]. In general, cache partitioning techniques can be divided into way partitioning and co-scheduling techniques. In a set-associative cache, partitioning is done by

D. Z. Tootaghaj and F. Farhat are with the School of Electrical Engineering and Computer Science at the Pennsylvania State University, PA, USA. Email: {dxz149, fuf111}@cse.psu.edu). A preliminary version appeared in Proc. IEEE ICCD'17 [1].

per-way allocation. For example, in a 4-way 512KB shared cache allocating 128KB to application *A* means to allow it storing data blocks in only one way per-set, without accessing remaining. Co-scheduling techniques try to co-schedule a set of applications with lowest interference together at the same time such that the magnitude of slow-down for each application is the same or a performance metric is optimized for all applications. However, it is shown that, depending on the objective function for the performance metric, cache allocation can result in totally different allocations [4]. In general Prior schemes have the following three limitations:

1. Scalability: All of the prior schemes suffer from scalability; especially when the approach is tracking the application's dynamism [3, 13, 17]. The reason is that algorithm complexity becomes higher in dynamic approaches. The root cause of this complexity is that all previous techniques make decisions (cache partitioning, co-scheduling) centralized using a central hardware or software. For example, main algorithm of [13] has exponential complexity $O(\binom{N+K-1}{K-1})$ where N is the number of applications sharing *LLC* and K is the number of ways. Table I shows the state of the art cache partitioning algorithms and their complexity of checking performance of different permutations.

2. Static-based: Most of the prior works, use static co-scheduling to degrade slow-down of co-running applications on the same shared cache. However, static-based approaches cannot catch dynamic behavior of applications. Figure 2 shows an example of two applications' IPC (*hmmer* and *mcf*) from *Spec 2006* under different *LLC* sizes. Let us consider a case where we have two cache sizes, a large cache of 1MB which can be shared between applications, and two private caches of 512KB which are not shared. The two applications are competing for the cache space. Suppose that both applications have two phases $(0, T)$ and $(T, 2T)$. If the first application gets the larger cache space its *IPC* increases by 35 percent in the first phase and by 20.6 percent in the second phase. The second application's *IPC* increases by 15 percent in the first phase and by 36.84 percent in the second phase if it gets the larger cache space. In a static-based scheduling approach, the larger *LLC* is always allocated to the first application with higher IPC in the time interval $(0, 2T)$, but in *CARMA*, the applications compete for the shared resources, and in the first phase, the larger *LLC* is allocated to the first application and in the second phase *CARMA* allocates the larger *LLC* to the second application. Therefore, static-based approaches cannot capture the dynamism in application's behavior and ultimately degrade the performance significantly.

3. Fairness: Defining a single parameter for fairness is challenging for multiple applications, since applications have different performance benefits from each resource during each phase. In prior works fairness has been defined as a unique metric (eg. IPC, Power, Weighted Speed-up) for all applications. Therefore, in current approaches, the optimization goal of algorithms is the same for all applications. Consequently, we cannot sum up applications that desire different metrics in the same platform to decide on. However, if one application needs better IPC and another requires lower energy, the previous algorithms are not able to model it. The only way to address diversity of metrics (to be optimized) is to have an appropriate translation between different metrics (eg. IPC to Power) that is not trivial, while not addressed in prior study.

TABLE I: Complexity comparison of state-of-the-art *LLC* partitioning/co-scheduling algorithms.

Algorithm	Search Space
Utility-based main algorithm [13]	$\binom{N+K-1}{N-1}$
Greedy Co-scheduling [17] N applications and N/K caches	$\binom{N}{K}$
Hierarchical perfect matching [17] N applications	N^4
Local optimization [17] N applications and N/K caches	$(N/K)^2\binom{2K}{K}$
CARMA N applications and K resources	$O(NK)$

In this paper, we present a game-theoretic resource assignment method to address all the above shortcomings including scalability, dynamism and fairness, while applications can get their desired performance based on their utility functions.

1. Semi-Decentralized: Dual of each centralized problem is decentralized, if the optimization goal is broken into a smaller meaningful sub-problems. In the context of heterogeneous resource assignment this is straightforward. The profiling, analyzing and evaluating the demands are on application side, but the final decision on assigning the resources to applications based on the applications' bids is easily performed by the OS while they compete with each other for the best assignment. Like a capitalist system, the complexity of the governing transfers to the independent entities, and the government just make the policies and the final decisions. To achieve this, we introduce a novel market-based approach. Roughly speaking, the complexity of our approach in worst case scenario (for each application) is $O(NK)$ where N is the number of the applications and K is the number of available resources. However, on average the auction terminates in less than $N/2$ iterations.

2. Dynamic: In order to confront the scalability problem of previous approaches, we use a market-based approach to move the decision making to the individual applications. Iterative auctions have been designed to solve non-trivial resource allocation problems with low complexity cost in government sale of resources, *eBay*, real estate sales and stock market. Similarly, decentralized computation complexity is lower than centralized (for each application) which provides the opportunity to make the decision revisiting the allocation in small time quantum, or when a new application leaves or comes into the system.

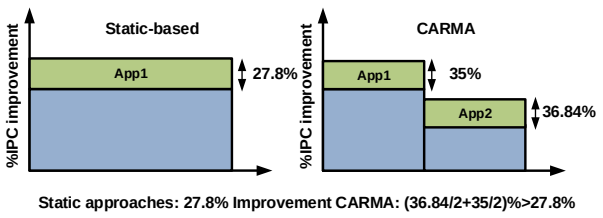


Fig. 2: Performance comparison of static and dynamic scheduling of two applications (*hmmer* and *mcf* from *Spec 2006*) under two different *LLC* sizes.

3. Fair: The proposed method solves the heterogeneous resource assignment problem in the context of marketing. Applications' demand regardless of the global optimization objective (IPC, Power, etc.) translates to the true valuation of their own performance. Resource assignment to the applications with the highest bids is performed by the auctioneer (the OS); making it local optimization objectives. Hence, resource assignment can be performed for different applications with different objectives known as utility functions.

Overall, the proposed approach for cache contention game on average brings in 33.6% improvement in system performance (when running 16 applications) compared to shared *LLC*; while reaching less than 11.1% of the maximum achievable performance in the best dynamic scheme. In the case study of heterogeneous CPU assignment, it brings in 106.6% improvement (when running 16 applications at the same time). Also, the performance improvement increases even more as the number of co-running applications increases in the system.

Other potentials: We introduce an auction-based resource management approach for different applications in large-scale competition games. In short, we as a system owner pay for a high-end CMP system for servers and guarantee that each application/user takes its best from the system by paying us back, or we as an application owner bid/pay the system to get the resources for my best performance. The auctioneer is application-agnostic, and does not interfere with applications' profile to globally optimize the system, but the applications compete for their own improvement. The two case studies of cache partitioning and CPU sharing are examples for resource sharing and the proposed approach can be employed in other resource partitioning algorithms.

The reminder of the paper is organized as follows. Section II discusses the background and motivation behind this work. In section III, we discuss our auction-based game model. Section IV discusses the case study of cache contention game and the case study of main processor and co-processor contention and simulation results. Section V studies related works and Section VI concludes the paper with a summary.

II. Motivation and Background

A. Motivation

Different applications have different resource constraint with respect to CPU, memory, and bandwidth usage. Having a single resource manager for all existing resources and users in the system result in inefficiencies since it is not scalable and the operating system may not have enough information about applications' needs. For example, traditional LRU-based cache strategy uses cache utilization as a metric to give larger cache size to the applications which have higher utilization and lower cache size to the applications with lower cache utilization. However more cache utilization does not always result in better performance. Streaming applications for example have very high cache utilization, but very small cache reuse. In fact, the streaming applications only need a small cache space to buffer the streaming data. With rapid improvements in semiconductor technology, more and more cores are being embedded into a single core and managing large scale application using a single resource manager becomes more challenging.

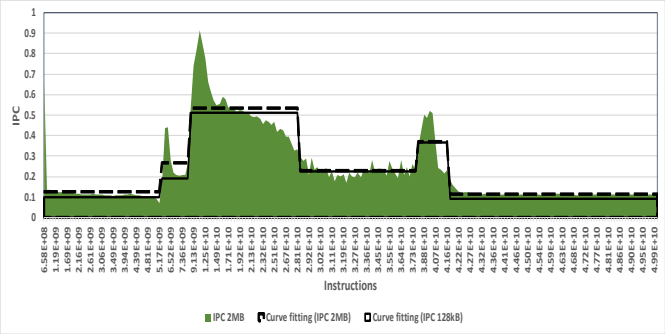


Fig. 3: Phase transition in mcf with different L2 cache sizes.

In addition, defining a single fairness parameter for multiple applications is non-trivial since applications have different bottlenecks and may get different performance benefits from each resources during each phases of their execution time. Defining a single reasonable parameter for fairness is somewhat problematic. For instance, simple assignment algorithms which try to equally distribute the resources between all applications ignores the fact that different applications have different resource constraints. As a consequence, this makes the centralized resource management systems very inefficient in terms of fairness as well as performance needs of applications. We need a decentralized framework, where all applications' performance benefit could be translated into a unique notion of fairness and performance objective (known as utility function in economics) and the algorithm tries to allocate resources based on this translated notion of fairness. This translation has been well defined in economics and marketing, where the diversity of customer needs, makes more economically efficient market [18]. Economists have shown that in an economically efficient market, having diverse resource constraints and letting the customers compete for the resources can make a Nash equilibrium where both the applications and the resource managers can be enriched. Furthermore, applications' demand changes over time. Most resource allocation schemes preallocate the resources without considering the dynamism in applications' need and number of users sharing the same resource over time. Therefore, applications' performance can degrade drastically over time. Figure 3 shows phase transitions for instruction per cycle (IPC) of mcf application from *spec 2006* over 50 billion instructions.

We try to find a game-theoretic distributed resource management approach where the shared hardware resources are exposed to the applications and we show that by running a repeated auction game between different applications which are assumed to be rational, the output of the game converges to a balanced Nash equilibrium allocation. In addition, we compare the convergence time of the proposed algorithm in terms of dynamism in the system. We evaluate our model with two case studies: 1) Private and shared last level cache problem, where the applications have to decide if they would benefit from a larger cache space which can potentially get more congested or a smaller cache space which is potentially less congested. 2) Heterogeneous processors (*Intel Xeon* and *Xeon Phi*) problem,

where we perform experiments to show how congestion affects the performance of different applications running on an *Intel Xeon* or *Xeon Phi* co-processors. Depending on the amount of congestion in the system, the application can offload the most time consuming part of its code on the *Xeon Phi* co-processors or not.

B. Background

Game theory has been used extensively in economics, political and social decision making situations [19–26]. A game is a situation, where the output of each player not only depends on her own action in the game, but also on the action of other players [27]. Auction games are a class of games which has been used to formulate real-world problems of assigning different resources between n users. Auction game framework can model resource competition, where the payoff (cost) of each application in the system is a function of the contention level (number of applications) in the game.

Inspired by market-based interactions in real life games, there exists a repeated interaction between competitors in a resource sharing game. Assuming large number of applications, we show that the system gets to a Nash equilibrium where all applications are happy with their resource assignment and don't want to change their state. Furthermore, we show that the auction model is *strategy-proof*, such that no application can get more utilization by bidding more or less than the true value of the resource. In this paper we propose a distributed market based approach to enforce cost on each resource in the system and remove the complexity of resource assignment from the central decision maker.

The traditional resource assignment is performed by the operating system or a central hardware to assign fair amount of resources to different applications. However, fair scheduling is not always optimal and solving the optimization problem of assigning m resources between n users in the system is an integer programming which is an NP-hard problem and finding the best assignment problem becomes computationally infeasible. Prior works focus on designing a fair scheduling function that maximizes all application's benefit [28–32], while applications might have completely different demands and it is not possible to use the same fairness function for all. By shifting decision making to the individual applications, the system becomes scalable and the burden of establishing fairness is removed from the centralized decision maker, since individual applications have to compete for the resources they need. Applications start by profiling the utility function for each resource and bid for the most profitable resource. During the course of execution time they can update their belief based on the observed performance metrics at each round of the auction. Updating the utility functions at each round of the auction is based on the history of the observed performance metrics which shows the state of the game. This state indicates the contention on the current acquired resources. The payoff function in each round depends on the state of the system and on the action of other applications in the system.

1) Sequential Auction: Auction-based algorithms are used for maximum weighted perfect matching in a bipartite graph $G = (U, V, E)$ [33–35]. A vertex $U_i \in U$ is the application in the auction and a vertex $V_j \in V$ is interpreted as a resource. The weight of each edge from U_i to V_j shows the utility of getting that particular resource by U_i. The prices are initially set to zero and will be updated during each iteration of the auction. In sequential auctions, each resource is taken out by the auctioneer and is sequentially auctioned to the applications, until all the resources are sold out.

2) Parallel Auction: In a parallel auction, the applications submit their bids for the first most profitable item. The value of the bid at each iteration is computed based on the difference of the highest profitable object and the second highest profitable object. The auctioneer would assign the resources based on the current bids. At each iteration, the valuation of each resource is updated based on the observed information during run-time which shows the contention on that particular resource.

III. The Method

Consider n applications and i instances of m different resources. Applications arrive in the system one at a time. The applications have to choose among m resources. There exists a bipartite graph between the matching of the applications and the resources.

In general, there can be more than one application to get a shared resource. However, each application cannot get more than one of the available heterogeneous resources. For example, if we have two cache spaces of size 128kB (one way) and 256kB (two ways), each application can either get the 128kB, or the 256kB cache space and can't get both of them at the same time. Furthermore, each resource R_k has a cost p_k which is defined by the applications' bid in the auction.

Figure 4 shows our auction-based framework to support *CARMA* between N applications that execute together competing for M different resources. Each application has a utility table that shows how much performance it gets from each M resources at each time slot. Based on the utility tables, applications submit bids for the most profitable resource. Based on the submitted bids, the auctioneer decides about the resource assignment for each resource, and updates the prices. Next, the applications who did not get any assignment compete for the next most profitable resource based on the updated prices repeatedly until all applications are assigned. Figure 4 shows an example of a resource assignment and the corresponding bipartite graph.

A. Problem Defenition

We formulate our problem as an auction to enforce cost/value updates for each resource as follows:

- **Valuation** $\mathbf{v_i}(\mathbf{t}, \tilde{\mathbf{m}})$: Application i has a valuation function which shows how much it benefits from the resource vector $\vec{m}$ at time t. The valuation function at time $t = 0$ for cache contention case study is derived from the IPC (instruction per cycle) curves using profiling, and for processor and co-processor contention case study is derived from the profiling of separate cache performance

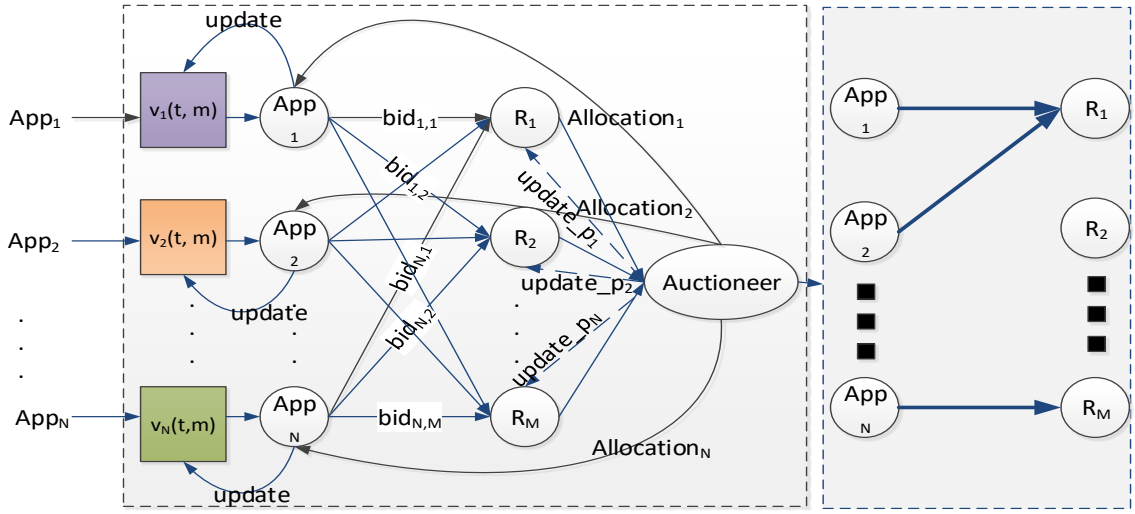


Fig. 4: Framework for auction-based resource assignment (CARMA).

TABLE II: Notation used in our formulations.

N	Number of players or applications (from App_1 to App_N).
M	Number of resources (from R_1 to R_M).
$\vec{m}$	A positive $M \times 1$ vector in the resource space that shows how much each application gets from each resource.
T	Time intervals where the bidding is held
$t_{i,j}$	j-th phase time for i-th application during its course of execution time.
T_i	Last phase time for application i.
$v_i(t, \vec{m})$	The valuation function of application i for the resource assignment $\vec{m}$ at time t.
$v_{i,j}(t, \vec{m}, r)$	The valuation function of (application i,resource j), if we replace the j-th resource in the resource vector m by r
δ	dynamic factor that shows how much we can rely on the past iterations.
$G = (U, V, E)$	A bipartite graph showing the resource allocation between the applications and the set of resources.
U	The set of applications which shows the left set of nodes in the bipartite graph $G = (U, V, E)$.
V	The set of resources which shows the right set of nodes in the bipartite graph $G = (U, V, E)$.
E	The edges in the bipartite graph.
$b_{i,k}$	User i's bid for k-th resource.
F_i	The total budget (summation of bids) a user have.
C_k	The total capacity of each resource.
p_k	The price of resource $k \in V$ in the auction.
K	Number of cache levels

of the application. However, in general, each application can choose its own utility function.

- **Observed information**: The observed information at each time step is the performance value of the selected action in the game. Therefore, the applications repeatedly update the history of their valuation function over time.
- **Belief updating**: Let T be the time intervals where the bidding is held. At each iteration step of the auction, the applications update their valuation of each resource based on the observed performance on the resource vector. The update at time W is derived using the following formula:

$$v_i(W, \vec{m}) = \frac{\sum_{0 \le n \le W/T} \delta^{W/T-n} \cdot v_i(nT, \vec{m})}{\sum_{0 \le n \le W/T} \delta^{W/T-n}}, \quad (1)$$

where $v_i(W, \vec{m})$ shows the observed valuation of resource vector $\vec{m}$ at time W by application i in the system; δ shows the discount factor between 0 and 1 which shows how much a user relies on its past observations in the system. The discount factor is chosen to show the dynamics of the system. If the observed information in the system changes fast, the discount factor is close to zero, i.e. the application cannot rely on the past observations very much. However, if the system is more stable and the observed information does not change fast, the discount factor is closer to 1. If a user fails in an auction, its payoff and corresponding observed valuation at the current time is equal to zero. So, it won't probably bid for the same resource vector again, since its valuation decreases for next round. We choose the discount factor to be the absolute value of the correlation coefficient of the observed values of the valuations at each iteration step which is calculated as follows:

$$\delta = \frac{E(v_i(W, \vec{m}))^2}{\sigma_{v_i(W,\vec{m})}{}^2} \quad (2)$$

- **Action**: At each time step, the applications decide which resource to bid and how much to bid for each resource.

Table II shows important notation used throughout the paper. In the following sections, we describe our distributed optimization scheme to solve the problem.

B. Distributed Optimization Scheme

The goal is to design a repeated auction mechanism which runs by the operating system to guide the applications to choose their best resource allocation strategy. The applications' goal is to maximize their own performance and the operating system wants to maximize the total utility gain from the applications. Each application can use its own utility function and evaluates the resources based on the desired value of the resources.

Applications' approach: The application i wants to maximize the expected utility (pay-off) with respect to a limited budget (F_i) during all phases of its execution time. We have:

$$\forall i \in U \quad maximize \sum_{0<t<T_i} v_i(t,\vec{m}) - b_i(t,\vec{m}),$$
$$subject\ to \sum_{0<t<T_i} b_i(t,\vec{m}) \leq F_i, \quad \forall i \in U. \quad (3)$$

OS's approach: The operating system wants to maximize the social welfare function which is translated into submitted bids from the applications in a limited resource constraints.

$$maximize \sum_{i=1}^{N} \sum_{0<t<T_i} b_i(t,\vec{m}) \cdot A_i(t,\vec{m}),$$
$$subject\ to \sum_{i=1}^{N} A_i(t,\vec{m}) \leq A_{max}, \quad \forall t : 0 \leq t \leq T,$$
$$A_i(t,\vec{m}) \in \{0,1\}, \ \forall i \in U, \ \forall \vec{m} \subseteq \mathcal{V}, \ \forall t : 0 \leq t \leq T, \quad (4)$$

where the binary variable $A_i(t,\vec{m})$ represents the decision to assign resource vector $\vec{m}$ to application i at time t (when $A_i(t,\vec{m}) = 1$) or not (when $A_i(t,\vec{m}) = 0$); and $\mathcal{V}$ is the vector space of the all resource vectors ($\forall \vec{m}$); and A_{max} shows the maximum number of the applications which can share the resource vector $\vec{m}$.

Illustrative example: As an illustrative example shown in Figure 2, let us consider a case where we have two different resources, a large cache of 1MB which can be shared between applications, and two private caches of 512KB which are not shared. The first application participates in the auctions with 35 bid at the first phase and 21 bid at the next phase. The second application participates in the auctions with 15 bid at the first phase and 37 bid at the next phase. The auctioneer (OS) decides to allocate larger cache in the auctions at first phase to the first application and at next phase to the second application.

C. Analysis

The distributed optimization problem is hard to solve. However, in reality, the problem can split into simpler subproblems since each application knows its bottleneck resource and would first bid for the first bottleneck resource to maximize its utility.

We suppose all applications in the system are risk-neutral which means they have a linear valuation of the utility function. Each risk-neutral agent wants to maximize its expected revenue. Risk attitude behaviors are defined in [36] where the agents can broadly be divided into risk-averse, risk-seeking and risk neutral. Risk-averse agents prefer deterministic values rather than risky value profits and risk-seeking applications have a super-linear utility function and prefer risky utilities than sure utilities. Next, we derive the Bayes Nash equilibrium strategy profile for all agents in the system assuming risk neutrality.

Definition 1: A strategy profile a is a pure Nash equilibrium if for every application i and every strategy $a'_i \neq a_i \in A$ we have $u_i(a_i, a_{-i}) \geq u_i(a'_i, a_{-i})$

Algorithm 1: Parallel Auction for Heterogeneous Resource Assignment

Input: A bipartite Graph (U, V, E).
Output: The allocation of resources to applications.

1 The initial resource vector for each application is the average amount across various resources. At time $t = nT$, the valuation of each application for each resource vector is updated using Eq. 1.

2 For application $U_i \in U$, the first bottleneck resource is

$$V_{i,j_1^{max}} = \max_{1 \leq j_1 \leq M} \Delta v_{i,j_1}(t,\vec{m},r_{j_1}) - p_{j_1}$$

where the differential valuation function is $\Delta v_{i,j}(t,\vec{m},r_j) = v_{i,j}(t,\vec{m},r_j) - v_i(t,\vec{m})$.

3 Find the second bottleneck resource for application $U_i \in U$ in the system:

$$V_{i,j_2^{max}} = \max_{1 \leq j_2 \leq M; j_2 \neq j_1} \Delta v_{i,j_2}(t,\vec{m},r_{j_2}) - p_{j_2}$$

4 Each application calculates the partial bid for its first bottleneck resource using the following formula:

$$b_{i,j_1^{max}}(t) = V_{i,j_1^{max}} - V_{i,j_2^{max}} + p_{j_1^{max}} + \epsilon$$

5 Each resource $r_j \in V$ which can be shared between L applications, is assigned to the L highest bidding applications $Winner_{i,j} = \{i_1, i_2, ..., i_L\}$ and the price for that resource is updated as follows:

$$p_j = \max_{l \in \{1,...,L\}} b_{i_l,j}$$

6 The $minBid$ for each resource is updated as the minimum bid of l applications who acquired j-th resource. That is:

$$B_j^{min} = \min_{i_l \in Winner_{i,j}} b_{i_l,j}$$

7 Goto step 2 until all partial bids for all resources are determined.

8 The total bid of the application i is as follows:

$$b_i(t,\vec{m}) = b_i(t,[r_{j_1}; r_{j_2}; ...; r_{j_M}]) = \sum_{j=1}^{M} b_{i,j}(t)$$

where iteratively $\vec{m} = [r_{j_1}; r_{j_2}; ...; r_{j_M}]$.

9 Find the estimated investment $I_i(t)$ using Algorithm 2 to plan the upper bound of the investment with respect to the budget F_i. If $I_i(t) \geq b_i(t,\vec{m})$, application i will participate in this auction at time t, otherwise it quits and other applications do the steps 2 and 3.

Theorem 1: Suppose n risk-neutral applications whose valuations are derived uniformly and independently from the interval $[0,1]$ compete for one resource which can be assigned to m applications who have the highest bid in the auction. We will show that Bayes Nash equilibrium bidding strategy for each application in the system is to bid $\frac{n-m}{n-m+1} v_i$ where v_i is the profit of application i for getting the specified resource.

Algorithm 2: Budget Planning

Input: A bipartite Graph (U, V, E).

Output: Participation (YES) in an auction or Quit (NO).

1 At time $t = nT$, assume that we have the same state in terms of resources.

2 For application $U_i \in U$, we similarly run the steps 2, 3, 4, 5 and 6 of Algorithm 1 to find all estimated bids in next rounds based on its various phases. We have:

$$b_i(t_{i,j}, \vec{m}) = \sum_{j=1}^{M} b_{i,j}(t); \forall t_{i,j} > t$$

Also, we have the previous bids of the application i:

$$b_i(t_{i,j}, \vec{m}); \forall t_{i,j} < t$$

3 If $F_i \geq \sum_{\forall t_{i,j} \neq t} b_i(t_{i,j}, \vec{m})$, then YES and the application will participate in the auction. Otherwise NO, and the application will update the zero valuation for current round using Eq. 1.

Theorem 1, states that whenever there is a single resource that users compete to get it with different valuation functions, the Nash equilibrium strategy profile for risk-neutral users is to bid $\frac{n-m}{n-m+1}v_i$. This term tends to the true value of the object when n is a large number.

In case of more than one resource competition we derive Algorithm 1 for heterogeneous resource assignment and will prove that it has a Nash equilibrium in the game. Algorithm 1 uses Algorithm 2 to do budget planning for our purpose. In the first step, all valuations are set to the solo-run of application's performance. Next, each application submits a partial bid for its first bottleneck resource. The partial bid should be larger than the price of the object which is initialized to zero at the beginning of the program. The applications only have the incentive to bid a value that is no more than the difference between the first and second bottleneck resource. Otherwise, it submits a smaller bid to the second bottleneck and gets the same revenue as paying more for the first bottleneck resource. In order to break the equal valuation function between two different applications, we use ϵ scaling such that at each iteration of the auction the prices should increase by a small number. The OS will set the resources' price with these partial bids, and find the minimum of the highest partial bids for each resource. The applications recurse for all the resources, and the total bid is the summation of the partial bids for each application. Then, the applications execute Algorithm 2 to participate in the auction or not. Finally the participated applications with the bids higher than B_j^{min} will get j-th resource.

The overhead of the auction for the auctioneer (the OS) is very negligible. The OS during the auction only sets the prices of the resources based on the received bids from the applications and gives the resources to the highest bids. So every T seconds, the OS runs these two jobs, which adds a negligible overhead with respect to other tasks of the OS. Our approach also satisfies the following properties: 1) Individual rationality (IR): Applications' expected utility is non-negative because the amount of the bid cannot be beyond the sum of the difference of the valuations which is at most the highest valuation of the application. 2) Truthfulness: Applications cannot benefit from bidding other than their true valuation. By contradiction, if an application bids lower than the true value, there may be another application with a higher bid to take the resource. But we cannot guarantee the truthfulness in the case of collusion among applications. 3) Budget-balance: The whole payments from the applications are less than the OS revenue, which is trivial as we have only one seller which is the OS. 4) Economic efficiency: It has been shown in [33] that this assignment is optimal, but it doesn't mean it is economically efficient since we know that it depends on the applications' valuation which is sub-optimal.

TABLE III: The comparison of *Intel Xeon* processor and *Intel Xeon Phi* processor.

Processors	Xeon E5-2680	Xeon Phi SE10P
Cores/Sockets	8/2	61/1
Clock Frequency	2.7 GHz	1.1 GHz
Memory	32GB 8x4G 4-channels DDR3-1600MHz	8GB GDDR5
L1 cache	32 KB	32 KB
L2 cache	256 KB	512 KB
L3 cache	20 MB	-

IV. Case Studies

A. CPU Scale-up Scale-out Game

The emerging high-performance computing applications lead to the advent of *Intel Xeon Phi* co-processor, that when their highly parallel architecture is fully utilized, can run in order of magnitude more performance than the existing processor architectures. The *Xeon Phi* co-processors are the first commercial product of Intel *MIC* processors where the hardware architecture is exposed to the programmer to choose running the code on either *Xeon* processor or *Xeon Phi* co-processors. It is possible that, during the course of execution, either the processor or the co-processor get congested and the performance of the application degrades a lot. Therefore, making a decision to offload the most time-consuming part of the program on *Xeon* or *Xeon Phi* should be made online, based on the contention level. In this section, we look at the case study of our auction-based model on decision making of running the application on the main or co-processor in a highly congested environment.

The experimental results of this section are run on *Stampede* cluster of *Texas Advanced Computing Center*. Table III shows the comparison of *Intel Xeon* and *Xeon Phi* architectures which is used in this section. It is observed that congestion has a significant impact on the performance of running the application on *Xeon* and *Xeon Phi* machines. Since most cloud computing machines are shared between thousands of users, the programmer not only should get the benefit of parallelism by offloading the most time-consuming part of the code to the larger number of low-performance cores (*Xeon Phi*) but also

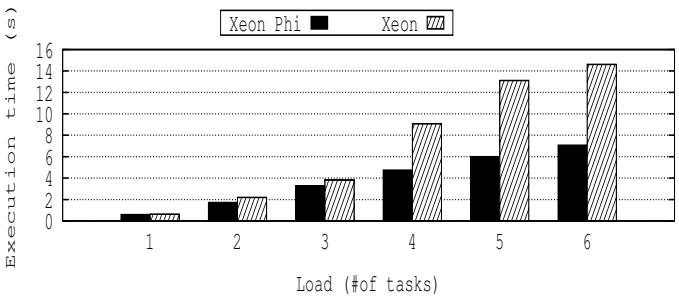


Fig. 5: Congestion effect on *Xeon* and *Xeon Phi* machines.

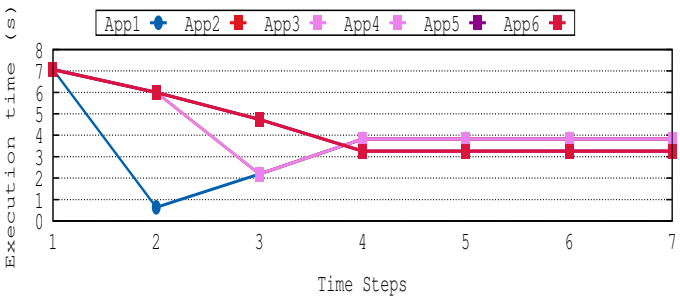


Fig. 6: Performance of 6 instances of applications during the execution time for our proposed game model.

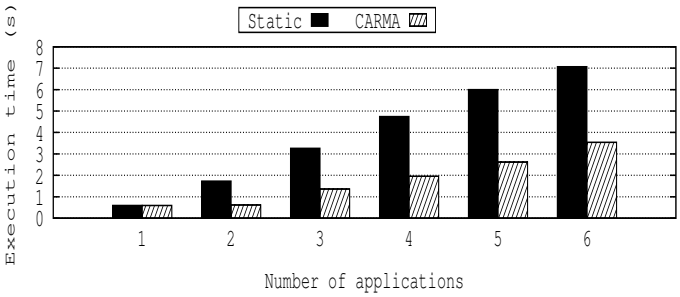


Fig. 7: Performance comparison of congestion-aware schedule versus static schedule.

should consider the congestion level (number of co-runners) in the system. To this end, we performed experiments on *Stampede* clusters. We executed *MiniGhost* application which is a part of *Mantevo* project [37] which uses difference stencils to solve partial differential equations using numerical methods. The applications use the profiling utility functions at $t = 0$ and during the course of execution update the utility function based on the observed performance on each core using Equation 1. Then, they can revisit their previous action on running the code on either the processor or co-processor during run-time.

Figure 5 shows the total execution time with respect to congestion we made in *Xeon* and *Xeon Phi*. In this experiment we ran the same problem size on a *Xeon* and *Xeon Phi* machine multiple times so that we could see the effect of load on the total execution time of our application. It was observed that with the same number of threads *Xeon*'s performance degrades more than *Xeon phi*. Next, we tried to change the application behavior using a congestion-aware game theoretic algorithm to offload the most time-consuming part of the application based on the performance behavior of applications. Figure 6 shows the result of our game-theoretic model during the execution time. It is observed that during the course of execution, the applications change their strategy on either choosing the main processor or the co-processor and all applications' performance converge to an equilibrium point where applications don't want to change their strategy.

Furthermore, it is shown that CARMA can bring in up to 106.6% improvement in total execution time of applications compared to static approach when the number of co-runners is six. The performance improvement would be significant when the number of co-runners increase. Figure 7 shows the performance comparison of CARMA and static approach which does not consider the congestion dynamism in the system and the decision is only made based on the parallelism level in the code.

B. A Case Study of Private and shared cache game

One of the challenging problems in *CMP* resource management systems is whether applications benefit from a shared large last level cache or an isolated private cache. We evaluated CARMA's performance, on a 10MB LLC shown in Figure 8, where 2MB, 1MB, 512kB, 256kB and 128kB levels of LLC can potentially be shared between 16, 8, 4, 2 and 1 applications respectively, the cache levels have 16, 8, 4, 2, and 1 ways. Table V summarizes the studies workloads and their characteristics, including miss per kilo instructions (*MPKI*), memory bandwidth usage, and IPC. We use applications from *Spec 2006* benchmark suite [38]. We use *Gem5* full system simulator in our experiment [39, 40]. Table IV shows the experimental setup in our experiments.

To evaluate the performance of our proposed approach we use utility functions for different number of cache ways shown in Figure 9. These utility functions at the start of the execution can be found using either profiling techniques or stack distance profile [5, 41, 42] of applications assuming there are no co-runners in the system. Next, during run-time, the applications can update their utility functions based on Equation 1. Therefore, there is a learning phase where applications learn about the state of the system and update the utilities accordingly. The stack distance profile indicates how many more cache misses will be added if the application has less number of ways in the cache. Based on the stack distance profile, the applications can update their utility function and bid for the next iteration of the auction if they like to change their allocation. Next, we bring an example of the auction for one time step of the game. This time step can be repeated once an application arrives or leaves the system or when an application's phase changes during run-time. However, in case of one application's phase change or arriving or leaving the system, the algorithm reaches the optimal assignment in much fewer iterations since all other assignments are fixed and a few applications would be affected.

Example: As an example, suppose we have 5 different applications and 5 different cache levels with different capacities of 128KB, 256KB, 512KB, 1MB and 2MB. In addition, suppose the 128kB cache level can not accomodate more than one application and 256kB cache can accomodate

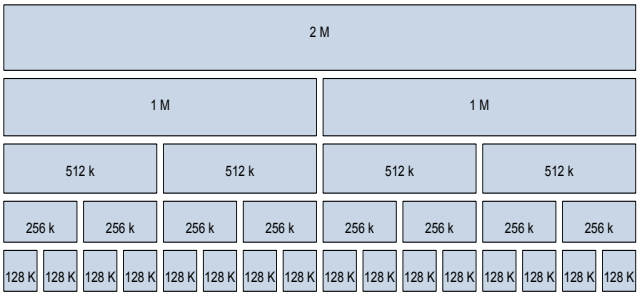


Fig. 8: The proposed last level cache hierarchy model.

TABLE IV: Experimental Setup.

Processors	Single threaded with private L1 instruction and data caches
Frequency	1GHz
L1 Private ICache	32 kB, 64-byte lines, 4-way associative
L1 Private DCache	32 kB, 64-byte lines, 4-way associative
L2 Shared Cache	128 kb-2 MB, 64-byte lines, 16-way associative
RAM	12 GB

TABLE V: Evaluated workloads.

#	**Benchmark**	MPKI	Memory BW	IPC
1	astar	1.319	373 MB/s	2.057
2	bwaves	10.47	1715 MB/s	0.661
3	bzip2	3.557	1194 MB/s	1.367
4	dealII	0.935	307 MB/s	2.107
5	GemsFDTD	0.004	2.19 MB/s	2.023
6	hmmer	2.113	1547 MB/s	2.861
7	lbm	19.287	3954 MB/s	0.533
8	leslie3d	8.469	1942 MB/s	1.297
9	libquantum	10.388	1589 MB/s	0.531
10	mcf	16.93	820 MB/s	0.073
11	namd	0.051	20.32 MB/s	2.362
12	omnetpp	10.34	1147 MB/s	0.504
13	sjeng	0.375	139.2 MB/s	1.403
14	soplex	4.672	390.8 MB/s	0.513
15	sphinx3	0.349	202.8 MB/s	2.223
16	streamL	31.682	3619 MB/s	0.581
17	tonto	0.260	107 MB/s	2.036
18	xalancbmk	12.703	1200 MB/s	0.558

2 applications, 512kB level can have 4 applications, 1MB cache can have 8 applications and 2MB cache can have at most 16 applications. Let's assume the following matrix be the utility function of each application on each cache level.

Some applications may get better utility from smaller cache space since they are less congested and since these applications have low data locality, moving to larger cache spaces not only does not increase their performance but also degrades the performance by evicting other applications from the cache and making contention on the memory bandwidth which is a more vital resource for them [1].

[1] *libquantum*, *streamL*, *sphinx3*, *lbm* and *mcf* are examples of such applications.

$$M = \begin{array}{c} \\ App1 \\ App2 \\ App3 \\ App4 \\ App5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1way & 2way & 4way & 8way & 16way \end{array} \\ \begin{pmatrix} 1.9 & 1.7 & 1.5 & 1 & 0.9 \\ 1.6 & 1.3 & 1.1 & 0.8 & 0.7 \\ 1.4 & 1.0 & 0.6 & 0.5 & 0.4 \\ 0.3 & 0.6 & 0.9 & 1.2 & 1.4 \\ 0.7 & 0.8 & 1.1 & 1.4 & 1.7 \end{pmatrix} \end{array} \quad (5)$$

In the first iteration of the bidding, the first 3 applications bid for the most profitable resource which is 128kB cache and they submit a bid equal to the difference of profit between the first and the second most profitable resource. Therefore, the first application, submits 0.2 bid to 128kb and the second application submits 0.3 and the third application submits 0.4. Since only one of the players can acquire the 128kB cache space, the first application will get it. The 4th and 5th application compete for 2MB cache space and they both get it with the sum bid of both which is 0.5. In the next round, the prices will be updated and since applications 2 and 3 don't have any cache assignment compete for the 256kB cache space and each bid 0.2 which is the difference between 1.7 and 1.5 and 1.3 and 1.1 in the performance matrix accordingly. Since the second level cache can accommodate both applications the price will be updated and the minimum bidding price for someone to get this cache level is updated to the minimum bid of both which is 0.2. Therefore, if some application bid more than 0.2 it can acquire the resource and the application with the smallest bid has to resubmit the bid to acquire the resource. Figure 10a, 10b, and 10c show the bidding steps and the prices and minimum price of bidding accordingly. As seen from the figures, the auction terminates in three iterations when there exist five applications.

C. A Case Study for Hybrid Cache Game

In hybrid cache game, each cache partition can have a cluster of applications. We use different mixes of 4 to 16 applications from *Spec 2006* to evaluate the performance of our proposed approach compared to others. To evaluate our approach, we selected the state-of-the-art centralized cache partitioner [43] (KPart) as a competitor which aims at maximizing the global IPC speedup. CARMA uses multi-resource valuations, so each application can have any criteria to maximize its payoff. In order to provide a fair comparison with our approach, we use IPC speedup as the optimization goal for all applications.

Figure 11 shows the normalized throughput of 10 different mix of applications [44], using CARMA, KPart [43], equal separate cache partitioning and completely shared cache space after convergence. Furthermore, Figure 12 shows the scalability of our proposed algorithm. When the number of co-runners increases from 2 to 16, the performance improves without any need to track each applications' performance in a central module. Having full information about applications' profiles, CARMA outperforms the other centralized competitors, when the number of the applications increases.

Since KPart is a centralized (not an auction-based) approach, we assume that it has an unlimited budget. The budget

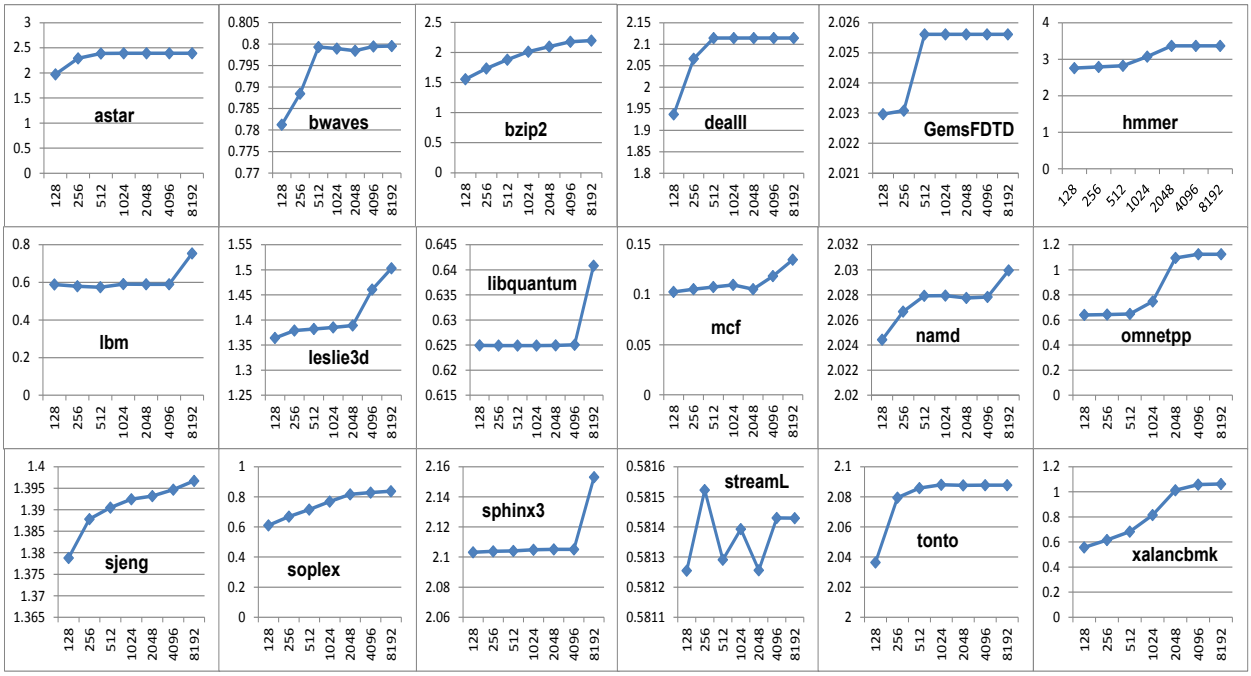
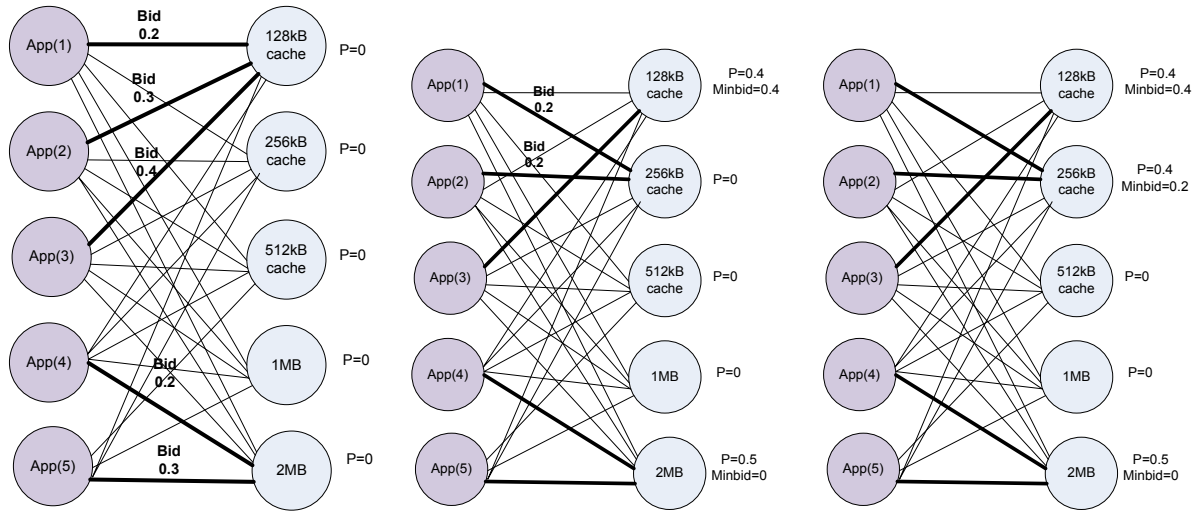


Fig. 9: IPC for different size of LLC.



(a) first round. (b) second round. (c) third round.

Fig. 10: Cache allocation, a) first round, b) second round and c) third round of bidding.

matters in CARMA. We setup another experiment to track the variations of the normalized throughput versus the normalized budget for a mix of 16 applications. Figure 13 shows that the throughput of CARMA, is very sensitive to the budget. The throughput changes dramatically at some inflection point, and at the end it is saturated but higher than KPart.

V. Related Work

With rapid improvement in computer technology, more and more cores are embedded in a single chip and applications competing for a shared resource is becoming common. On the one hand, managing scheduling of shared resources for a

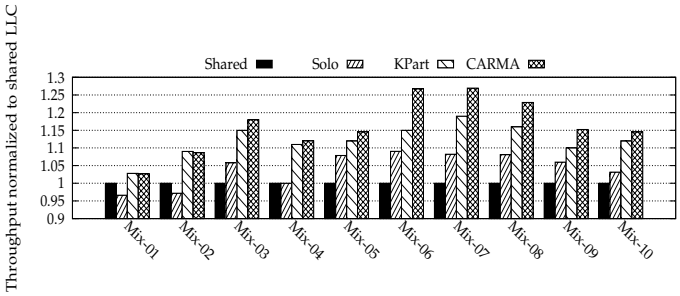


Fig. 11: Throughput of a shared, solo, CARMA and KPart cache allocation schemes.

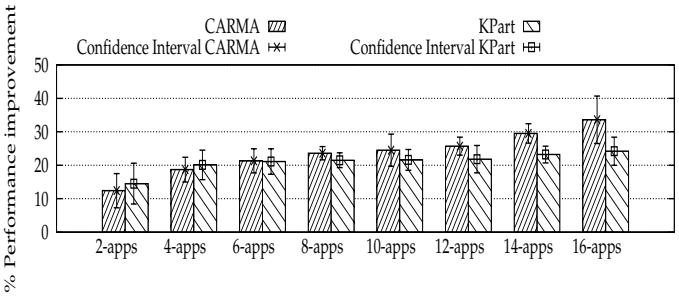


Fig. 12: Performance improvement of CARMA and KPart for different number of applications with respect to shared LLC for the case study of cache congestion game.

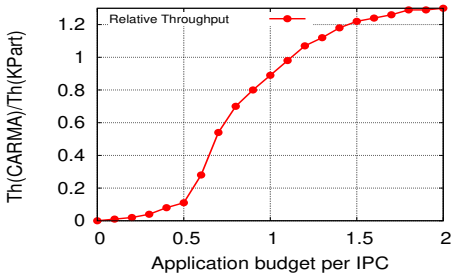


Fig. 13: Relative throughput w.r.t applications' normalized budget.

large number of applications is challenging in a sense that the operating system doesn't know what is the performance metric for each application. But on the other hand, the operating system has a global view of the whole state of the system and can guide applications on choosing the shared resources.

There have been several works, for managing the shared cache in multi-core systems. Qureshi et al. [13] showed that assigning more cache space to applications with more cache utility does not always lead to better performance since there exist applications with very low cache reuse which may have very high cache utilization.

Several software and hardware approaches have been proposed to find the optimal partitioning of cache space for different applications [3]. However, most of these approaches use brute force search of all possible combinations to find the best cache partitioning in runtime or introduce a lot of overhead. There have been some approaches which use binary search to reduce searching all possible combinations [5, 14, 45]. But none of these methods are scalable for the future many-core processor designs.

There exists prior game-theoretic approaches designing a centralized scheduling framework that aims at a fair optimization of applications' utility [28–32]. Zahedi et al. in REF [28, 31] use the Cobb-Douglas production function as a fair allocator for cache and memory bandwidth. They show that the Cobb-Douglas function provides game-theoretic properties such as sharing incentives, envy-freedom, and Pareto efficiency. But their approach is still centralized and spatially divides the shared resources to enforce a fair near-optimal policy sacrificing the performance. In their approach, the centralized scheduler assumes all applications have the same priority for cache and memory bandwidth, while we do not have any assumption on this. Further, our auction-based resource allocation can be used for any number of resources and any priority for each application and the centralized scheduler does not need to have a global knowledge of these priorities.

Ghodsi et al. in DRF [30] use another centralized fair policy to maximize the dominant resource utilization. But in practice, it is not possible to clone any number of instances of each resource. Cooper [29] enhances REF to capture colocated applications fairly, but it only addresses the special case of having two sets of applications with matched resources. Fan et al. [32] exploits computational sprinting architecture to improve task throughput assuming a class of applications where boosting their performance by increasing the power.

While all prior works use a centralized scheduling that provides fairness and assumes the same utility function for all, co-runners might have completely diverse needs and it is not efficient to use the same fairness/performance policy across them. Our auction-based resource scheduling provides scalability since individual applications compete for the shared resources based on their utility and the burden of decision making is removed from the central scheduler. We believe that future CMPs should move toward a more decentralized approach which is more scalable and provides a fair allocation of resources based on the applications' needs.

Auction theory which is a subfield of economics has recently been used as a tool to solve large-scale resource assignment in cloud computing [46, 47]. In an auction process, the buyers submit bids to get the commodities and sellers want to sell their commodities with the maximum price as possible. Also auction-based allocators [48, 49] are multi-buyers with multi-seller but there is only one resource to bid. So, they cannot be used for our purpose, since we have only one seller with multiple bundled resources. That is why we choose a simpler related scheme for a computer architecture to get higher performance with lower transactions and auctions.

Our auction-based algorithm is inspired by work of Bertsekas [33] that uses an auction-based approach for network flow problems. Our algorithm is an extension of local assignment problem proposed by Bertsekas et al. that has been shown to converge to the global assignment within a linear approximation.

VI. Conclusion

This paper proposes a distributed resource allocation approach for large-scale servers. The traditional resource management system is not scalable, especially when tracking the application's dynamic behavior. The main cause of this complexity is the centralized decision making which leads to higher time and space complexity. With increasing number of cores per chip, the scalability of assigning different resources to different applications becomes more challenging in future generation CMP systems. In addition, diversity in application's need makes a single objective function inefficient to get an optimal and fair performance metric. We introduce a framework to map the allocation problem to the known auction economy model where the applications compete for the shared resources based on their utility metrics of interest.

REFERENCES

[1] D. Z. Tootaghaj and F. Farhat. Cage: A contention-aware game-theoretic model for heterogenous resource assignment. In *The 35th IEEE International Conference on Computer Design (ICCD)*, 2017.

[2] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, 2011.

[3] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[4] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *PACT*. ACM, 2006.

[5] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[6] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *MICRO*, 2006.

[7] F. Farhat, D. Z. Tootaghaj, Y. He, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. Stochastic modeling and optimization of stragglers. *IEEE Transactions on Cloud Computing*, 2016.

[8] D. Z. Tootaghaj and F. Farhat. Optimal placement of cores, caches and memory controllers in network on-chip. *arXiv preprint arXiv:1607.04298*, 2016.

[9] D. Z. Tootaghaj, F. Farhat, M. Arjomand, P. Faraboschi, M.t T. Kandemir, A. Sivasubramaniam, and C. R. Das. Evaluating the combined impact of node architecture and cloud workload characteristics on network traffic and performance/cost. In *IEEE international symposium on Workload characterization (IISWC)*. IEEE, 2015.

[10] F. Farhat, D. Z. Tootaghaj, and M. Arjomand. Towards stochastically optimizing data computing flows. *arXiv preprint arXiv:1607.04334*, 2016.

[11] D. Z. Tootaghaj. *Evaluating Cloud Workload Characteristics*. PhD thesis, Pennsylvania State University, 2015.

[12] C. Liu, A. Sivasubramaniam, and M. T. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *IEEE Proceedings on Software*, 2004.

[13] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*. IEEE Computer Society, 2006.

[14] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *IEEE 14th International Symposium on High Performance Computer Architecture (HPCA)*, pages 367–378, 2008.

[15] R. Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *ICS*. ACM, 2004.

[16] N. Rafique, W. T. Lim, and M. Thottethodi. Architectural support for operating system-driven cmp cache management. In *PACT*. ACM, 2006.

[17] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *PACT*. ACM, 2008.

[18] Y. Zhou and D. Wentzlaff. The sharing architecture: sub-core configurability for iaas clouds. In *ACM SIGARCH Computer Architecture News*, 2014.

[19] D. Z. Tootaghaj, F. Farhat, M. R. Pakravan, and M. R. Aref. Game-theoretic approach to mitigate packet dropping in wireless ad-hoc networks. In *IEEE CCNC*, 2011.

[20] D. Z. Tootaghaj, F. Farhat, M. R. Pakravan, and M. R. Aref. Risk of attack coefficient effect on availability of ad-hoc networks. In *IEEE CCNC*, 2011.

[21] K. Kotobi and S. G. Bilen. Spectrum sharing via hybrid cognitive players evaluated by an m/d/1 queuing model. *EURASIP Journal on Wireless Communications and Networking*, 2017.

[22] K. Kotobi and S. G. Bilen. Introduction of vigilante players in cognitive networks with moving greedy players. In *Vehicular Technology Conference (VTC Fall)*. IEEE, 2015.

[23] G. Kesidis, K. Kotobi, and C. Griffin. Distributed aloha game with partially rule-based cooperative, greedy, and vigilante players. *Department of Computer Science and Engineering, Penn State University, Tech. Rep. CSE*, 2013.

[24] A. Kurve, K. Kotobi, and G. Kesidis. An agent-based framework for performance modeling of an optimistic parallel discrete event simulator. *Complex Adaptive Systems Modeling*, 2013.

[25] N. Nasiriani, C. Wang, and B. Kesidis, G. Urgaonkar. Using burstable instances in the public cloud: Why, when and how? *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2017.

[26] C. Wang, N. Nasiriani, G. Kesidis, B. Urgaonkar, Q. Wang, L. Y. Chen, A. Gupta, and R. Birke. Recouping energy costs from cloud tenants: Tenant demand response aware pricing design. In *Proceedings of the 2015 ACM Sixth International Conference on Future Energy Systems*. ACM, 2015.

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

[28] S. M. Zahedi and B. C. Lee. Ref: Resource elasticity fairness with sharing incentives for multiprocessors. *ACM SIGARCH Computer Architecture News*, 2014.

[29] Q. Llull, S. Fan, S. M. Zahedi, and B. C. Lee. Cooper: Task colocation with cooperative games. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017.

[30] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, 2011.

[31] S. M. Zahedi and B. C. Lee. Sharing incentives and fair division for multiprocessors. *IEEE Micro*, 2015.

[32] S. Fan, S. M. Zahedi, and B. C. Lee. The computational sprinting game. In *ACM SIGOPS Operating Systems Review*. ACM, 2016.

[33] D. P. Bertsekas. *Network Optimization: continuous and discrete methods*. Athena Scientific, 1998.

[34] A. S. Kyle. Continuous auctions and insider trading. *Econometrica: Journal of the Econometric Society*, 1985.

[35] Cristina Nader Vasconcelos and Bodo Rosenhahn. Bipartite graph matching computation on gpu. In *EMMCVPR*. Springer, 2009.

[36] J. Ferber. *Multi-agent systems: an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.

[37] `http:/manetovo.org`.

[38] `http://www.spec.org/spec2006`.

[39] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[40] `http://gem5.org/`.

[41] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium on*. IEEE, 2002.

[42] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 2004.

[43] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez. Kpart: A hybrid cache partitioning-sharing technique for commodity multicores. In *HPCA*, 2018.

[44] S. Srikantaiah, E. Kultursay, T. Zhang, M. T. Kandemir, M. J. Irwin, and Y. Xie. Morphcache: A reconfigurable adaptive multi-level cache hierarchy. In *HPCA*. IEEE, 2011.

[45] D. K Tam, R. Azimi, L. B. Soares, and M. Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ACM SIGARCH Computer Architecture News*, 2009.

[46] V. Krishna. *Auction theory*. Academic press, 2009.

[47] S. Parsons, J. A. Rodriguez-Aguilar, and M. Klein. Auctions and bidding: A guide for computer scientists. *ACM Computing Surveys (CSUR)*, 2011.

[48] D. Zhang, Z. Chang, T. Hamalainen, and F. R. Yu. Double auction based multi-flow transmission in software-defined and virtualized wireless networks. *Trans. Wireless. Comm.*, 2017.

[49] S. Zou, , Z. Ma, and X. Liu. Resource allocation game under double-sided auction mechanism: Efficiency and convergence. *IEEE Transactions on Automatic Control*, 2018.

Diman Zad Tootaghaj received the Ph.D. degree in computer science and engineering from the Pennsylvania State University in 2018. She received B.S. and M.S. degrees in Electrical Engineering from Sharif University of Technology, Iran in 2008 and 2011, and M.S. and Ph.D. degrees in Computer Science and Engineering from the Pennsylvania State University in 2015 and 2018. Her current research interests include computer network, recovery approaches, distributed systems, and stochastic analysis.

Farshid Farhat is a PhD candidate at the School of Electrical Engineering and Computer Science, The Pennsylvania State University. He obtained his B.Sc., M.Sc., and Ph.D. degrees in Electrical Engineering from Sharif University of Technology, Tehran, Iran. His current research interests include resource allocation in parallel, distributed systems, computer vision and image processing. He is working on the modeling and analysis of image composition and aesthetics using deep learning and image retrieval on different platforms ranging from smart phones to high-performance computing clusters.