A Coordinated Multi-Agent Reinforcement Learning Approach to Multi-Level Cache Co-partitioning

Rahul Jain
Dept. of Computer Sc. and Engg.
Indian Institute of Technology Delhi
Email: rahuljain@cse.iitd.ac.in

Preeti Ranjan Panda
Dept. of Computer Sc. and Engg.
Indian Institute of Technology Delhi
Email: panda@cse.iitd.ac.in

Sreenivas Subramoney
Microarchitecture Research Lab
Intel Technology India Pvt. Ltd., Bangalore
Email: sreenivas.subramoney@intel.com

***Abstract*—The widening gap between the processor and memory performance has led to the inclusion of multiple levels of caches in the modern multi-core systems. Processors with simultaneous multithreading (SMT) support multiple hardware threads on the same physical core, which results in shared private caches. Any inefficiency in the cache hierarchy can negatively impact the system performance and motivates the need to perform a co-optimization of multiple cache levels by trading off individual application throughput for better system throughput and energy-delay-product (EDP). We propose a novel coordinated multi-agent reinforcement learning technique for performing Dynamic Cache Co-partitioning, called Machine Learned Caches (MLC). MLC has low implementation overhead and does not require any special hardware data profilers. We have validated our proposal with 15 8-core workloads created using Spec2006 benchmarks and found it to be an effective co-partitioning technique. MLC exhibited system throughput and EDP improvements of up to 14% (gmean:9.35%) and 19.2% (gmean: 13.5%) respectively. We believe this is the first attempt at addressing the problem of multi-level cache co-partitioning.**

I. INTRODUCTION

Modern multi-core systems have multi-level caches of different capacities exhibiting varying access latencies, motivating a trade-off between individual application throughput and system throughput by performing efficient cache space allocations at various levels.

The on-chip caches commonly use some variant of the Least Recently Used (LRU) replacement policy which exploits the temporal locality of the applications and implicitly partitions the shared cache based on the application requirements. This implicit partitioning may not be efficient if an application with low data reuse flushes high reuse data of the other competing applications. The cache space is utilized best when the working set of an application fits the allocated space, and there is high data reuse. Since the working set of an application varies over the various execution phases, a dynamic partition methodology would work best.

Effective cache partitioning enables each sharing core to retain a desired working set in the cache, isolated from other cores' cache activity. Since each application has a different working set, a multi-core system can perform better cache management if the cache space can be allocated on multiple cache levels by co-partitioning. For example, on a 2-SMT core, with 128 KB L2 and 1 MB L3, two applications with working sets of 100 KB and 1000 KB, and similar data reuse frequency, are expected to exhibit high system throughput if the first application occupies the 100 KB on L2 and negligible space on L3. This ensures that both the applications are able to fit their respective working sets within the on-chip caches compared to an implicit partitioning which may have resulted in the larger application dominating all the cache levels.

II. RELATED WORK

Cache partitioning (CP) research has focused on the partitioning of the shared last level cache (LLC). However, on SMT processors, partitioning is also relevant for the private caches along with the shared LLC. Static profiling based LLC partitioning [1] and dynamic cache reconfiguration of L1 [2] and LLC [3] have been studied for performance and power. A static profiling based approach may not be feasible for systems with a large number of target applications.

Dynamic Cache Partitioning (DCP) adapts to the multiple application cache requirements during runtime. It consists of two independent steps: (1) finding the cache allocation policy, which computes the optimal cache space allocation for concurrently running applications; and (2) policy enforcement control, which ensures that the cache space allocation is enforced as per the computed policy. Utility based cache partitioning (UCP) [4] is a popular technique for implementing cache allocation policy. Most DCP work has focused on the policy enforcement controllers (step two), while utilizing the UCP technique for step one [5]–[10]. Jain et al. [11] proposed a single agent reinforcement learning model for allocation policy on the LLC. There are proposals for efficient cache management via cache replacement policy by incorporating multiple application and thread priorities [12], [13].

In an SMT, dynamic sharing of intra-core resources (register file, reorder buffers, etc.) [14] and shared LLC [15] have been studied. None of the above work has considered the problem of performing co-partitioning of the multiple cache levels. We focus on the cache allocation policy at L2 and L3 cache levels simultaneously. The proposed dynamic co-partitioning technique, Machine Learned Caches (MLC), can be used with any of the policy enforcement controllers, and does not require any special hardware data profilers.

The rest of the paper is organized as follows: Section III provides the motivation for multi-level cache co-partitioning.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

Fig. 1. Throughput sensitivity to varying L3, L2, L1D cache ways normalized to maximum cache ways

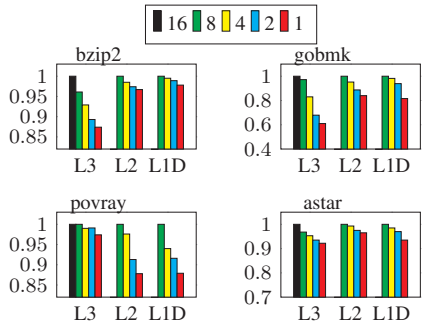
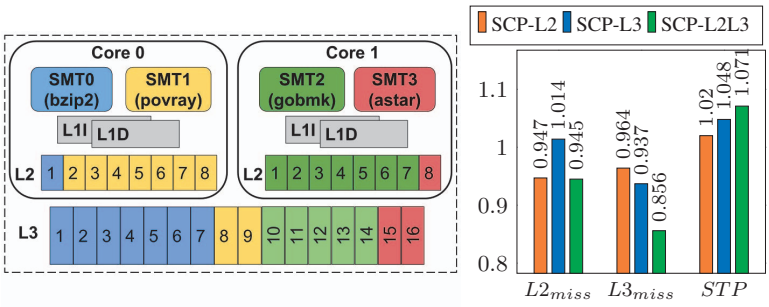


Fig. 2. Co-partitioning for Multiple Applications (bzip2, povray, gobmk, astar)



Section IV introduces the coordinated multi-agent reinforcement learning model. Section V discusses our novel proposal to infer additional learnings called Smart Updates. Section VI discusses the experimental setup, results, and the hardware overheads, comparing our co-partitioning technique (MLC) with the state-of-the-art cache partitioning research. Any reference to core implies an SMT-core throughout the paper.

III. MOTIVATION FOR CO-PARTITIONING

A. Single Application Cache Sensitivity

An application's working set determines its sensitivity to different cache levels. To understand the cache way allocation sensitivity of different applications, we simulated the Spec2006 benchmarks for different cache sizes with fixed number of cache sets and varying number of cache ways. Figure 1 shows a few Spec2006 benchmarks with their respective sensitivities at different cache levels. As shown in Figure 1, not all applications are sensitive to all cache levels. For example, *bzip2* showed a high sensitivity to available L3 cache ways but very low sensitivity to L1D and L2 cache ways, while *povray* showed high sensitivity to reducing L1D and L2 cache ways, but very low sensitivity to L3 cache ways. This analysis motivates the problem of efficiently partitioning across cache levels among multiple applications.

B. Multiple Application Cache Hierarchy Partitioning

When multiple applications are running on an SMT system, the applications compete for space on all the shared caches. The L1D cache shows a very high degree of data reuse due to temporal and spatial locality, which filters out the high locality requests to L2 and L3. Cache partitioning reduces the available cache associativity to an application. To keep the L1D simple, we have focused on co-partitioning of L2 and L3 only. We experimented with a 4-benchmark workload of *<bzip2, povray, gobmk, astar>* on a system with a total of 4 SMTs (2 physical cores with two SMT per core) as shown in Figure 2. Based on the cache sensitivity information from the single application runs of Figure 1, the different cache levels were statically partitioned at the start of the experiment run as shown in Figure 2. *bzip2* has low L2 sensitivity while *povray* is L2 sensitive, hence more space on L2 is assigned to SMT1. Since *bzip2* has high L3 sensitivity, SMT0 is assigned more space on L3. On the other physical core *astar* shows very low sensitivity to both L2 and L3 caches and hence is allotted less space on L2 and L3. Figure 2 shows the cache misses at L2 and L3 levels for the static cache partitioning (SCP) performed on the workload, normalized to the cache misses with the LRU based implicit cache allocation. SCP-L2 and SCP-L3 correspond to the runs where only L2 and L3 caches were partitioned respectively. SCP-L2L3 is the run when both L2 and L3 have been partitioned. The SCP-L2L3 achieves a System Throughput (STP) improvement of 7.1% and outperforms both SCP-L2 (2.0%) and SCP-L3 (4.8%).

The above experiment motivates the co-partitioning of multiple cache levels for better system throughput. A dynamic co-partitioning which can adapt to the changing application execution phases should result in better improvements. We propose to use Reinforcement Learning (RL) [16] technique, since it does not require any offline training or cache utility model, and has the capability to learn the good co-partitions by interacting with the system at runtime. Additionally, RL technique has low overheads as discussed in Section VI-C.

IV. MULTI-AGENT REINFORCEMENT LEARNING

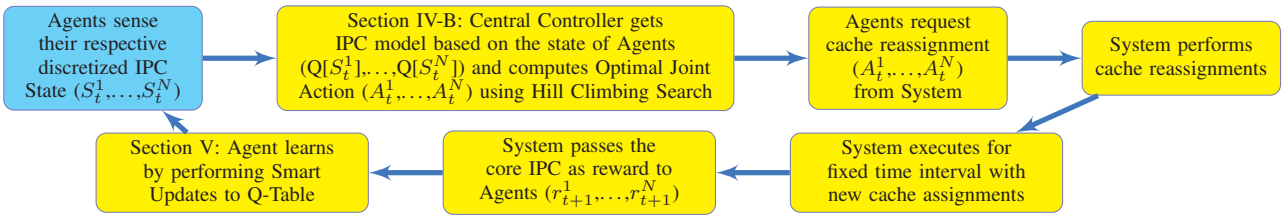
Reinforcement Learning [16] is a computational model for reward based learning implemented using the Markov Decision Process (MDP) framework. It allows an autonomous agent to interact with a dynamic environment and learn the optimal control policy based on the rewards earned for the various actions by trial and error. Q-Learning is one of the most popular RL algorithms, and applications based on this technique have achieved impressive results in recent times [17]. A one-step Q-Learning can be represented by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \times max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

where an agent performs an action a_t in state s_t at the start of an interval t, to receive a reward r_{t+1} at the end of t. Q is the Action-Utility function being learned online, α is the learning rate, and γ is the reward discount factor.

A single MDP agent model for cache co-partitioning would have a large number of state-action pairs to be learned, resulting in an unfeasible RL model. We propose to execute one MDP agent per core, requesting for cache allocation at different levels based on the observed reward. The coordinated

Fig. 3. Multiple Agent Coordination and Interaction with the Architectural Environment



multi-agent approach requires a central cache reconfiguration controller which searches for globally optimal joint actions for the various agents. At the start of each fixed-time interval, an agent senses its state and requests a cache reconfiguration (action), and receives a reward at the end of the interval. Based on the rewards the agent learns the good states and reassignment actions as a Q function. Figure 3 shows the work-flow for multi-agent coordination and interaction with the architectural environment.

A. Agent Model for Multi-Level Cache Co-partitioning

The effectiveness of an RL technique depends on the fidelity of the MDP model. The MDP should capture the various system metric sensitivities in the state model enabling the agent to identify the good and bad states. The action results in reconfiguration and its effect must be captured well in the reward function to enable the agent to identify the good and the bad actions from a state. We propose to execute one MDP per core which maintains its own learnings in a Q-Table and enables the central cache reconfiguration system to search for the globally optimal joint actions.

1) State Model: To capture the cache system states, our MDP model observes the Instructions-Per-Cycle (IPC) on the core. An application experiencing low cache misses would have a high IPC and vice versa. The change in IPC is a consequence of cache reassignment actions of the agent and the application phase changes. The MDP model is capable of handling such uncertain behavior and hence its learning towards an optimal policy is not impacted. The IPC values are quantized into 10 different levels resulting in 10 states.

2) Action Space: The actions of the MDP model represents the cache hierarchy reconfiguration. A co-partitioning action is a vector *Action* representing the allocation and deallocation requests for all the cache levels. For two cache levels (L2 and L3), a reconfiguration action has two elements, and is represented by *Action* = $<i, j>$, where integers $i, j \in [-w_n, +w_n]$, referring to the requested changes in the number of ways for L2 and L3 respectively, and w_n is the maximum number of cache ways that can be reassigned in a single step. Positive and negative numbers in the above actions represent allocation and deallocation respectively. At each step, the agent can request for multiple way allocation at a single cache level which enables it to explore cache sizes greater than the working set of the application and learn the cache level sensitivities. This simplifies the actions to $<i, 0>$ and $<0, j>$. In our system we have used $w_n = 2$ for L2 and $w_n = 4$ for L3.

3) Reward: The reward is the motivation for the agent which helps it to distinguish between the good and the bad actions from a particular state. In general, the reward should be some function of the desired metric being optimized. We propose to use IPC as a motivating reward for the agent. This represents that the MDP agent is working towards maximizing the application throughput by finding the best cache partition size at various cache levels.

B. Coordinated Multi-Agent Joint Actions

In the proposed system, each core has an independent MDP agent optimizing the performance for its application. The MDP model discussed in Section IV-A would request L2 and L3 cache ways for the best possible application IPC. Such a model would motivate the agent to secure the maximum possible cache, without any consideration to the overall system performance. Additionally, a cache is a limited resource, and the system cannot fulfill all the requests from the agents. This problem can be solved if the agents can cooperatively work together towards a common goal of higher system throughput.

Figure 4 shows an illustrative example for a system at time step t with four agents in states $s_t^1, s_t^2, s_t^3, s_t^4$ respectively, and five possible independent actions namely, $a1, a2, a3, a4$ and $a5$. Figure 4 shows the Q-Table entries for the possible actions for each of the agents in their respective states. A system with non-coordinating agents would result in each agent picking the maximum Q-value action from its respective state. In the example, this would result in a joint action of $<a5, a3, a5, a2>$ which may not be a feasible joint action for the system due to a mismatch in the combined allocation and deallocation requests. If the agents coordinate and jointly explore feasible actions, then the agents can observe the correct rewards for their respective actions. Additionally, the agents can work together to search for a joint action expected to maximize the system utility. Figure 4 shows four different feasible joint actions, $JA1, JA2, JA3, JA4$, for the system along with the respective Q-value sums. In this example, $JA2$ is found to be the optimal joint action with 6.1 units of utility.

1) Optimal Joint Action Search: We propose a coordinated joint action selection where a central controller decides the actions for the various MDP agents. A Q-Table entry, $Q[s][a]$ represents the expected IPC for the cache reassignment action by the agent. The central controller has access to the Q-Tables and attempts to select a joint action for the maximal system IPC. Algorithm 1 shows the proposed *Hill Climbing* algorithm to perform the optimal joint action search. The algorithm starts with an initial joint action computed by picking the

ALGORITHM 1: Optimal Joint Action Search

Input: Q[S_i] for each $Agent_i$ on $Core_i$, for i $\in 1, .., n$
Output: Optimal Joint Action Vector $A[1..n]$
1 $CurrentJA$ = [A_1,..., A_n] where $A_i = max_a$(Q[S_i][a])
2 makeValid($CurrentJA$)
3 $index$ = 0, $MaxQSum$ = 0, $OptimalJA$ = [0,...,0]
4 **repeat**
5 mutate($CurrentJA$)
6 $QSum = \sum_{i=1}^{n}$(Q[S_i][$CurrentJA[i]$]
7 **if** $MaxQSum < QSum$ **then**
8 $OptimalJA = CurrentJA$
9 $MaxQSum = QSum$
10 **end**
11 **until** $(index++) > Iterations$
12 return $OptimalJA$

maximum IPC actions for each agent, and converts it into a valid joint action by balancing the combined allocation and deallocation requests (*makeValid* on Line 2). In each iteration, the *CurrentJA* is randomized or changed in a small step (*mutate* on Line 5). To change *CurrentJA*, an agent is picked randomly and its action in the *CurrentJA* is changed to a neighboring action in the action space. This results in additional allocation or deallocation requests for the cache ways, which are balanced by adjusting other agents' actions. The new joint action is evaluated (Line 6) as the sum of the expected IPC for the actions in the *CurrentJA*. If the new joint action is better, then it is stored as the *OptimalJA* found so far (Line 8). The quality of a hill climbing search solution depends on the number of iterations performed to find the optimal solution. In our experiments we have used 500 iterations for an 8-core system.

Fig. 4. Finding Optimal Coordinated Joint Actions

Actions	a1	a2	a3	a4	a5
$Q^1[s_t^1]$	0.5	0.6	0.7	0.8	**1.0**
$Q^2[s_t^2]$	0.8	0.8	**1.1**	1.0	1.0
$Q^3[s_t^3]$	1.0	1.6	1.7	1.8	**2.0**
$Q^4[s_t^4]$	1.5	**2.6**	0.7	0.8	1.0

Agent	1	2	3	4	QSum
JA1	a1	a3	a2	a1	4.7
JA2	a1	a4	a5	a1	6.1
JA3	a5	a1	a2	a2	6.0
JA4	a4	a3	a3	a4	4.4

V. Smart Updates to Q-Table

The effectiveness of Q-Learning depends on how quickly the Q-Values converge to the optimal policy. The Q-Update equation in Section IV performs a one-step look-ahead on the Q-Value by computing the new Q-Value based on the reward (r_{t+1}) received, and the maximum expected utility ($max(Q(s_{t+1}, a_i))$) from the new state (s_{t+1}) of the agent. With coordinated joint actions, the agent is no longer guaranteed to take the maximal utility action in state s_{t+1}, as the next action a_{t+1} is determined by the optimal joint action. Hence, the look-ahead for the maximal expected utility can be avoided by setting γ to zero, and converting the Q-Table function to represent the immediate expected reward.

In our setup, the Q-Table with $\gamma = 0$ represents the expected IPC of the application in different states as a function of cache size changes, i.e., the $Q[s]$ row represents the expected IPC function for different cache way changes performed when in state s. This expected IPC function is a monotonically non-decreasing function on cache size. Based on the observed IPC due to the last action, the expected IPC for other cache assignments can be inferred. For example, if the agent had taken an action of deallocating a cache way on L3 cache, and received a reward (IPC) as r, then one can safely conclude that at larger cache sizes of L3, the IPC would be at least r. Similarly, at lower cache sizes the IPC would not exceed r. This implies that an observed IPC value from any state-action pair, can be used to infer the lower (upper) bound of the expected IPC for the configurations with larger (smaller) cache size. Performing these additional learning inferences enables the agent to adapt to the application phase changes quickly. We propose to perform multiple Q-Learning updates using r for the Q-values based on the inferred IPC bounds.

Fig. 5. Smart Updates for L2 and L3 reassignment request actions

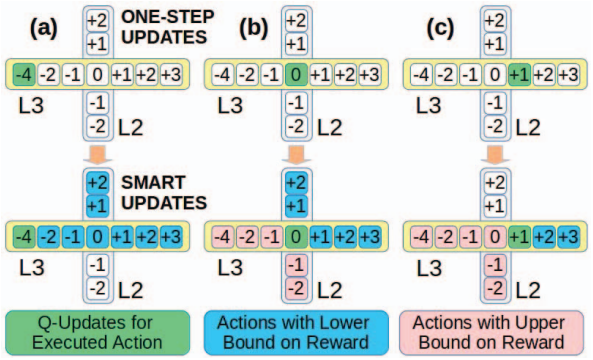


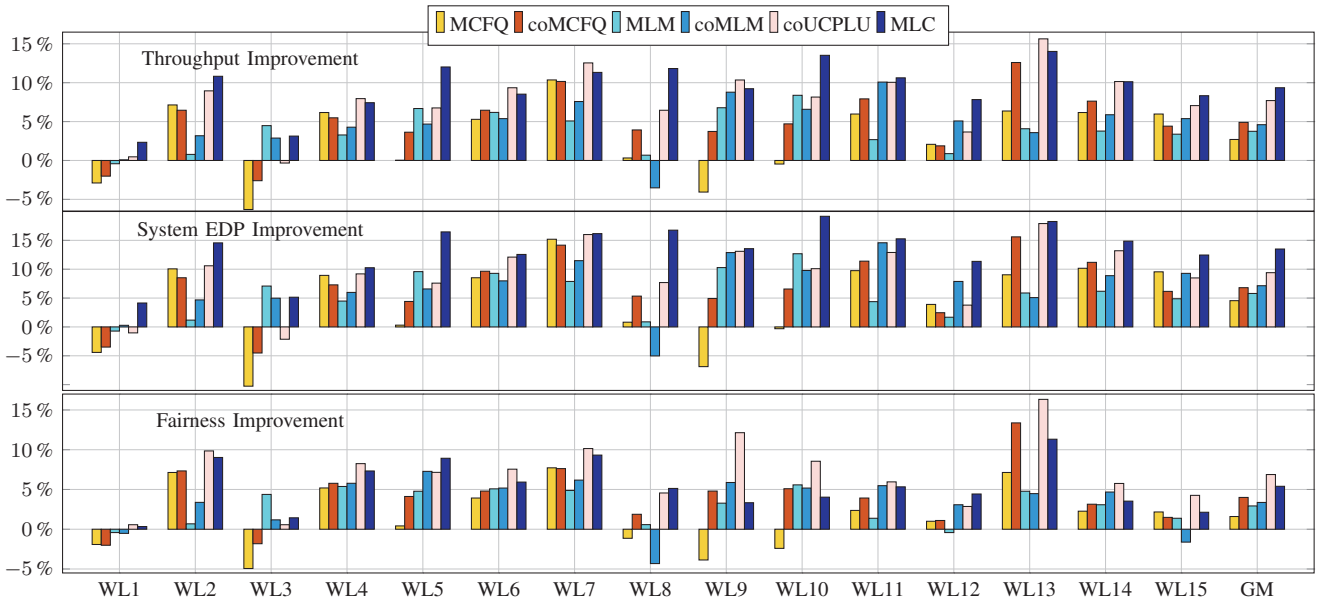
Figure 5 illustrates Smart Updates for three different actions. The figure shows the action space as reassignment on L2 or L3 with an intersection at [0,0] (no-change) action. In Figure 5(a), a reward update is performed for an action [0, -4] (marked green), corresponding to deallocating four L3 ways. The received IPC can be used to infer the lower limit on the actions marked in blue. Note that this reward can also be inferred as a lower limit along the L2 actions where the L2 has not decreased. For the L2 actions that reduce the L2 cache size (marked white), one cannot infer any such relation. Figure 5(b) shows an update for action [0, 0] corresponding to no change in cache size at any level. The observed IPC in this case can be used to infer the IPC for other actions as shown in Figure 5(b). Figure 5(c) shows an update for action [0, +1] corresponding to an additional L3 cache way allocation. The observed IPC can be used to infer the lower bound on the expected IPC for actions [0,+2] and [0,+3] (marked blue) and upper bound on some actions (marked pink).

VI. Experiments

A. Experimental Setup

We used the Sniper simulator [18] integrated with McPAT for power estimation and the *pybrain* library for the machine learning algorithm implementation. Table I shows the baseline 8-core system used for evaluation. The system simulation is

Fig. 6. System metrics improvement for 15 workloads on the 8-core system normalized to the baseline system (Table I) metrics



divided into fixed-time intervals of 1.56 ms ($\sim$ 5 million cycles). At the end of each fixed-time interval, a reconfiguration time penalty based on the partition algorithm latency (Section VI-C) is added to enable the computation of the new cache partitions. The cache way reassignments do not flush out the previous owner's data during reconfiguration but it gets evicted as the new owner starts filling this additional cache way.

A single 250 million instruction PinPoint, which is a representative and repeatable program region, is identified for each Spec2006 benchmark. The benchmarks are classified into two categories, cache sensitive (S) and cache non-sensitive (N). The cache sensitive benchmarks exhibit medium to high L2 and L3 cache sensitivity, while the non-sensitive benchmarks exhibit low or no sensitivity. A total of 15 workload mixes are created with different combinations of benchmark categories as listed in Table II. During simulation, a benchmark is restarted if it completes earlier than the other benchmarks. The evaluation has been performed for average IPC (STP), system energy-delay-product (EDP), and harmonic mean of the weighted IPC (Fairness). All the results presented in Section VI-B are normalized to the system metrics of the baseline system (Table I).

B. Result Analysis

We compared our co-partitioning technique, MLC, on an 8-core system with MCFQ [10], MLM [11] and UCP-LU [19]. These techniques have been extended to the co-partitioning problem by applying them simultaneously on L2 and L3 cache levels, and labeled as coMCFQ, coMLM and coUCPLU. MCFQ utilizes a variant of UCP-LA [4] along with additional hardware profilers to perform cache partitioning. UCP-LU is an offline supervised machine learning based cache partitioning technique. MLM is a single MDP agent based LLC partitioning, which is extended as coMLM, by executing one MDP agent for each L2 and L3 cache, resulting in an uncoordinated multi-agent RL model. Figure 6 compares the individual workload STP, EDP and Fairness improvements, and the geometric mean (GM) of the metric improvements for the 15 workloads (Table II).

TABLE I
8-CORE BASELINE SYSTEM CONFIGURATION

Architecture	Intel x86 Nehalem
Cores	8-SMT cores (2-SMT per physical core), 3.2 GHz
L1 Cache	split, private 32 KB L1I, 32 KB L1D
L2 Cache	private, 256 KB, 8-way, 10 cycle access
L3 Cache	8 MB, 32-way, 30 cycle access, 8-core shared
Uncore	NoC+L3, 3.2 GHz
DRAM	60ns ($\sim$ 200 cycles) access

TABLE II
15 WORKLOAD MIX USED FOR EXPERIMENTS

#	WL Mix	8-core Workloads using Spec2006
1	NNNNNNNN	zeusmp,libq,mcf,namd,calculix,mcf,zeusmp,Gems
2	SNNNSNNN	omne,milc,namd,sjeng,gobmk,zeusmp,calculix,sphinx3
3		povray,astar,sjeng,GemsFDTD,gcc,lbm,milc,zeusmp
4	SNNNSSSN	tonto,mcf,astar,milc,xalancbmk,gcc,hmmer,namd
5		xalanc,astar,sjeng,Gems,omnetpp,gobmk,povray,lbm
6	SNSNSNSN	gromacs,mcf,gcc,namd,perlbench,calculix,hmmer,mcf
7		gromacs,sphinx3,tonto,astar,hmmer,sjeng,bzip2,Gems
8		povray,zeusmp,omnetpp,Gems,bzip2,astar,gobmk,libq
9		soplex,sphinx3,bzip2,lbm,perlbench,mcf,xalanc,astar
10		tonto,lbm,soplex,milc,gobmk,zeusmp,xalancbmk,libq
11		xalanc,sjeng,gcc,calculix,hmmer,Gems,omnetpp,libq
12	SSSNSSSN	gcc,hmmer,omne,namd,soplex,bzip2,perlbench,sjeng
13		gcc,perlbench,hmmer,lbm,povray,omnetpp,bzip2,mcf
14		povray,gcc,soplex,calculix,gromacs,tonto,hmmer,Gems
15	SSSSSSSS	hmmer,omne,soplex,bzip2,perlb,xalanc,omne,gobmk

As shown in Figure 6, our proposed co-partitioning technique, MLC, outperforms by exhibiting higher STP and EDP improvements on most workloads. MLC is able to outperform the evaluated state-of-the-art techniques by exhibiting average improvements on STP and EDP by 9.3% and 13.5%, compared to coMCFQ (4.9% and 6.8%), coMLM (4.6% and 7.14%),

and coUCPLU (7.7% and 9.4%). MLC also shows the effectiveness of performing coordinated multi-agent optimization, demonstrating better results than the uncoordinated coMLM technique. MLC was effective in trading off the individual application throughput for higher system throughput with improved average Fairness (5.4%), which is slightly worse than the offline trained coUCPLU (6.88%).

As shown in Figure 6, coMCFQ and coMLM improved STP by 4.9% and 4.6% respectively, outperforming the LLC only DCP techniques MCFQ (2.7%) and MLM (3.75%). This shows the effectiveness of performing simultaneous partitioning on multiple cache levels.

C. Hardware Overheads

This section discusses the hardware overheads of the MLC and state-of-the-art techniques for the co-partitioning problem. Table III summarizes the average estimated overheads for all co-partitioning techniques on an 8-core system. The computation energy due to the partitioning algorithm is estimated using McPAT and added to the system energy, while the latency overheads are used as the reconfiguration time penalty in Section VI-A.

MCFQ, UCP-LU partitioning are UCP-based techniques requiring special UMON profiler circuits [4], and a partitioning algorithm hardware. To perform the co-partitioning of the cache hierarchy, one UMON is required per core at each cache level, resulting in 16 UMONs for an 8-core system. Since the UMON circuits are always active and maintain the cache tags' LRU information, they can have a significant impact on the total cache power. The UMON circuit power overheads are estimated using the CACTI tool. MCFQ uses the Look-Ahead partitioning algorithm [4] which has a large overhead on the delay and power [19]. UCP-LU uses offline machine learning to reduce this partitioning overhead.

The MLM and MLC partitioning are RL-based techniques which maintain a Q-Table per MDP agent along with a one-step learning update circuit. Both the techniques read certain system metrics available as hardware counters on modern processors and do not require any custom hardware profilers. This provides an option to implement these techniques in software on a system which supports DCP. The power and latency overheads are due to the computations and memory accesses while performing the *Q-Table* updates. Additionally, MLC technique has power and latency overheads due to the hill climbing search computations, resulting in higher overheads compared to MLM. As shown in Table III, MLM and MLC exhibit very low hardware overheads compared to UCP-based techniques.

TABLE III
AVERAGE ESTIMATED OVERHEADS ON A 8-CORE SYSTEM

Co-partitioning	Storage	Power	Latency
UCP-LU	69.0 KB	1.9%	0.23%
MCFQ	67.7 KB	2.63%	2.67%
MLM	1.3 KB	0.002%	0.02%
MLC	1.75 KB	0.008%	0.16%

VII. CONCLUSION AND FUTURE WORK

We presented a motivation for the co-partitioning of multiple cache levels. We proposed and evaluated a coordinated multi-agent RL technique, Machine Learned Caches (MLC), to address the co-partitioning problem, which has low implementation overhead, and does not require any special hardware circuits for data profiling. MLC showed an average system throughput and EDP improvement of 9.35% and 13.5% on an 8-core system outperforming the state-of-the-art techniques. In the future, we plan to extend the technique to co-partition L1 caches along with L2 and L3 caches, and plan to evaluate other learning algorithms on the co-partitioning problem.

ACKNOWLEDGMENT

We would like to thank Intel, CII and SERB for partial sponsorship of this research, and Mausam for his comments and suggestions on the machine learning models.

REFERENCES

[1] C. Yu and P. Petrov, "Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms," in *DAC*, 2010.

[2] W. Wang, P. Mishra, and S. Ranka, "Dynamic cache reconfiguration and partitioning for energy optimization in real-time multi-core systems," in *DAC*, 2011.

[3] S. Mittal, Y. Cao, and Z. Zhang, "Master: A multicore cache energy-saving technique using dynamic cache reconfiguration," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 22, no. 8, 2014.

[4] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006.

[5] Y. Xie and G. H. Loh, "PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches," in *ISCA*, 2009.

[6] K. Kamil, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-lru replacement policies," in *IPDPS*, 2010.

[7] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ISCA*, 2011.

[8] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs," in *HPCA*, 2012.

[9] S. Khan, A. R. Alameldeen, C. Wilkerson, O. Mutlu, and D. A. Jiménez, "Improving cache performance by exploiting read-write disparity," in *HPCA*, 2014.

[10] D. Kaseridis, M. F. Iqbal, and L. K. John, "Cache friendliness aware management of shared last-level caches for high performance multi-core systems," *IEEE Transactions on Computers*, vol. 63, no. 4, 2014.

[11] R. Jain, P. R. Panda, and S. Subramoney, "Machine Learned Machines: Adaptive co-optimization of caches, cores, and on-chip network," in *DATE*, 2016.

[12] A. Sharifi, S. Srikantaiah, M. Kandemir, and M. J. Irwin, "Courteous cache sharing: being nice to others in capacity management," in *DAC*, 2012.

[13] K. Aisopos, J. Moses, R. Illikkal, R. Iyer, and D. Newell, "PCASA: Probabilistic control-adjusted selective allocation for shared caches," in *DATE*, 2012.

[14] H. Wang, I. Koren, and C. M. Krishna, "Utilization-based resource partitioning for power-performance efficiency in smt processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 7, 2011.

[15] J. Chen and L. K. John, "Predictive coordination of multiple on-chip resources for chip multiprocessors," in *ICS*, 2011.

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press, Cambridge, 1998.

[17] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

[18] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: Scalable and accurate parallel multi-core simulation," in *HiPEAC*, 2012.

[19] I. A. Guney, A. Yildiz, I. U. Bayindir, K. C. Serdaroglu, U. Bayik, and G. Kucuk, "A machine learning approach for a scalable, energy-efficient utility-based cache partitioning," in *ISC*, 2015.