Cooperative Multi-Agent Reinforcement Learning-Based Co-optimization of Cores, Caches, and On-chip Network

RAHUL JAIN and PREETI RANJAN PANDA, Indian Institute of Technology Delhi
SREENIVAS SUBRAMONEY, Microarchitecture Research Lab, Intel India

Modern multi-core systems provide huge computational capabilities, which can be used to run multiple processes concurrently. To achieve the best possible performance within limited power budgets, the various system resources need to be allocated effectively. Any mismatch between runtime resource requirement and allocation leads to a sub-optimal energy-delay product (EDP). Different optimization techniques exist for addressing the problem of mismatch between the dynamic requirement and runtime allocation of the system resources. Choosing between multiple optimizations at runtime is complex due to the non-additive effects, making the scenario suitable for the application of machine learning techniques. We present a novel method, Machine Learned Machines (MLM), by using online reinforcement learning (RL) to perform dynamic partitioning of the last level cache (LLC), along with dynamic voltage and frequency scaling (DVFS) of the core and uncore (interconnection network and LLC). We have proposed and evaluated three different MLM co-optimization techniques based on independent and cooperative multi-agent learners. We show that the co-optimization results in a much lower system EDP than any of the techniques applied individually. We explore various RL models targeted toward optimization of different system metrics and study their effects on a system EDP, system throughput (STP), and Fairness. The various proposed techniques have been extensively evaluated with a mix of 20 workloads on a 4-core system using Spec2006 benchmarks. We have further evaluated our cooperative MLM techniques on a 16-core system. The results show an average of 20.5% and 19.1% system EDP improvement on a 4-core and 16-core system, respectively, with limited degradation of STP and Fairness.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; *System on a chip*; • **Hardware** → **Chip-level power issues**; • **Computing methodologies** → *Multi-agent reinforcement learning*;

Additional Key Words and Phrases: Energy efficient computation, computer architectures, low power architectures, cache management, dynamic voltage, and frequency scaling

Extension of Conference Paper: A preliminary version of this work was published at the Design Automation and Test in Europe (DATE-2016) conference with the title “Machine Learned Machines: Adaptive Co-optimization of Caches, Cores, and On-chip Network” in March 2016.

This extension has explored four additional co-optimization models. Two of the additional models are extensions of the DATE-2016 proposal, while two models are novel co-optimization models based on cooperative learning among the multiple agents. The two cooperative learner-based co-optimization techniques, coMLM and JMLM, are shown to scale well to higher core counts by evaluating it on a 16-core system, which exhibits 19.1% system EDP improvement.

Authors’ addresses: R. Jain and P. R. Panda, Dept. of Computer Science and Engineering, Indian Institute of Technology Delhi; Hauz Khas, New Delhi 110016, India; emails: {rahuljain, panda}@cse.iitd.ac.in; S. Subramoney, No 23-56, P Devarabeesanahalli Outer Ring Road, Varthur Hobli, Ballandur Post, Bangalore, Karnataka - 560037; email: sreenivas.subramoney@intel.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/11-ART32 \$15.00

<https://doi.org/10.1145/3132170>

ACM Reference format:

Rahul Jain, Preeti Ranjan Panda, and Sreenivas Subramoney. 2017. Cooperative Multi-Agent Reinforcement Learning-Based Co-optimization of Cores, Caches, and On-chip Network. *ACM Trans. Archit. Code Optim.* 14, 4, Article 32 (November 2017), 25 pages.
<https://doi.org/10.1145/3132170>

1 INTRODUCTION

A multi-core system is expected to run different programs simultaneously with their own runtime resource requirements. These resource requirements vary as the program executes, and any mismatch in the resource requirement and allocation leads to sub-optimal performance and power. Various proposed optimization techniques address this problem of resource mismatch.

Multiprocessor systems-on-chip (MPSoC) have been witnessing increasing core counts to enable higher computational capabilities. The increasing core count leads to an increase in the shared resources such as Last Level Cache (LLC) and interconnects. The increasing hardware size and complexity poses multiple power related challenges and requires careful optimization of the system energy-delay product (EDP). To obtain better EDP, we typically deploy multiple optimization techniques simultaneously which helps in reducing the power for specific system components. The various optimization techniques do not always give an additive improvement when applied simultaneously. This non-uniform effect on power and performance during various program execution phases renders the decision to find the best subset of the optimizations challenging. This problem is further complicated on a multi-core system due to global effects on the shared resources. We address this problem of simultaneous application of multiple power optimization techniques on multi-core systems using online reinforcement learning (RL). We also show that applying multiple optimization techniques in an integrated strategy gives superior results compared to their independent application.

The co-optimization problem is a sequential decision problem where a sequence of resource allocation decisions are required to ensure optimal system performance. Hence, the resource allocator should look to optimize for the long term. RL (Sutton and Barto 1998) is a machine learning technique, suited to sequential decision-making problems, which performs the optimization searches for long term cumulative rewards (Dietterich 2004).

The rest of the article is organized as follows. Section 2 discusses the related work. Sections 3 and 4 briefly introduce the motivation for co-optimization and online RL; Section 5 discusses efficient RL models for DVFS on the cores and uncore; and Section 6 discusses a novel LLC partitioning scheme and its RL model. Sections 7 and 8 discuss an independent RL technique and its extension to a cooperative information sharing technique. Section 9 details the various co-optimization systems built and evaluated by using different combinations of RL-controllers for cores, uncore, and the LLC. Sections 10 and 11 propose a novel cooperative joint action RL technique and the corresponding agent model, respectively. Sections 12 and 13 present the experimental setup and result analysis, respectively.

2 RELATED WORK

Runtime resource allocation for optimizing various multi-core system objectives such as reliability, performance, and power (Henkel et al. 2013; Wildermann et al. 2014) has been the focus of research attention in recent years. Dynamic voltage and frequency scaling (DVFS) and dynamic cache partitioning (DCP) techniques allow adaptive runtime resource allocation, with frequency and cache size being the respective resources.

The DVFS efforts focus on reducing dynamic power (Bircher and John 2008) and leakage power (Jejurikar and Gupta 2004; Zhong and Xu 2008) for system-wide energy minimization. DVFS can be applied effectively if the frequency scaling effects on the core performance can be accurately modeled considering realistic memory systems (Miftakhutdinov et al. 2012) and core-level activity prediction (Bircher and John 2011). RL is gaining traction as a popular technique for power management (Jung and Pedram 2010; Wang et al. 2011; Tan and Qiu 2008). An initial attempt at on-line learning-based CPU-DVFS used an estimate of the CPU-intensiveness of running applications (Dhiman and Rosing 2007). Recently, a comparative study showed that co-operative core/uncore DVFS using RL technique gives superior results compared to isolated core and uncore DVFS (Juan and Marculescu 2012).

DCP is a popular technique focused on improving the system throughput. Cache Partitioning can be broadly divided into two categories of techniques. The first partitions the cache based on some property of the stored data (Khan et al. 2014; Manikantan et al. 2012; Sharifi et al. 2012; Sanchez and Kozyrakis 2011), while the other partitions the cache among the sharing cores based on the utility to the cores (Qureshi and Patt 2006; Gordon-Ross and Vahid 2007; Moreto et al. 2008; Yu and Petrov 2010). Early work on DCP has focused on reducing the shared cache misses in order to increase system throughput (Qureshi and Patt 2006; Gordon-Ross and Vahid 2007). Recently, DCP methods based on memory-level parallelism (Moreto et al. 2008; Kaseridis et al. 2014), QoS guarantee (Moreto et al. 2009), and reducing off-chip memory bandwidth (Yu and Petrov 2010) have been shown to be effective. Vantage (Sanchez and Kozyrakis 2011) divides the shared cache into two regions: a managed and an unmanaged region. The technique is based on skew-associative caches (Seznec 1993) to improve the associativity and reduce conflicts. Read-Write partitioning (Khan et al. 2014) is based on the fact that reads are more critical than writes and based on attempts to increase the number of read-hits in the LLC. Probabilistic Shared Cache Management (PriSM) (Manikantan et al. 2012) proposes a fine-grain cache management by changing the allocation policy based on the eviction probability distribution. None of these works have considered applying additional power saving techniques such as core or uncore DVFS along with DCP.

While a significant body of research exists on the individual optimization techniques, applying multiple techniques simultaneously presents some interesting challenges due to the non-additive effects on the performance and power. Handling multiple optimizations also leads to an increase in the number of parameters/metrics to be considered at runtime. Efficient multiple resource allocation requires fast techniques to explore the large design space (Chen et al. 2011). The complexity of effectively applying multiple optimizations can be handled by the machine learning techniques.

Juan and Marculescu (2012) proposed a Power-Aware Reinforcement Controller (PARC) for modeling core and uncore DVFS. Chen and Marculescu (2015) proposed a distributed RL model to perform DVFS for maximum performance within a limited Thermal Design Power (TDP). RL-based techniques have been effectively applied to the memory controller optimization by using binary rewards (Ipek et al. 2008) and multi-objective offline reward derivation (Mukundan and Martinez 2012). Bitirgen et al. (2008) proposed a framework based on an Artificial Neural Network (ANN) to predict the Instructions per Cycle (IPC) for the cores and perform coordinated multiple interacting resource (CMIR) allocation. Chen and John (2011) have proposed a framework for performing coordinated multiple on-chip resource allocations by predicting application performance based on the data collected from various special hardware profilers. Wang and Martínez (2015) proposed a framework based on an economic market where different agents competitively bid for various resources based on the utility to the core.

We had proposed a co-optimization architecture, Machine Learned Machine (Jain et al. 2016), which deployed multi-agent online RL to perform Core-DVFS, Uncore-DVFS, and DCP on the LLC. In this article, we have extended the proposals to fine tune the individual optimization

Fig. 1. Cores and Uncore frequency variation and 8-way DP-LLC with changing program phases.

techniques based on RL. We have fine tuned the Core-Ctrlr performing Core-DVFS by including branch misprediction into the controller state, which produces better results. We have also explored two additional state-reward models for the LLC-Ctrlr performing the LLC partitioning outperforming the earlier model (Jain et al. 2016). We have further proposed a strategy based on the cooperative agents technique (Tan 1993) to enable information sharing between the LLC-Ctrlr and Core-Ctrlrs resulting in better system EDP. We have proposed another novel multi-agent cooperation technique, by searching for joint actions, which enables the multiple agents to work together toward a global target optimization. We have addressed the hardware overhead computations and evaluated the proposals extensively using 20 different 4-benchmark workloads. Our cooperative RL-based multiple resource optimization models are evaluated against adaptations of CMIR (Bitirgen et al. 2008) and XCH (Wang and Martínez 2015).

3 MOTIVATION FOR CO-OPTIMIZATION

Different applications running simultaneously on a multicore system impose different runtime resource requirements on the system. These resource requirements change with the execution phases of the applications. The actual resources allocated to each application are dependent on the total available resources and demands from other competing processes. In order to optimize performance and energy, runtime optimizations must attempt to match the requirements to allocations.

A system trying to optimize the individual applications in isolation would let the process demands decide on the resource allocation without any consideration to global performance. This may not be the optimal way to handle allocation. For example, in a 4-core system (Figure 1), when one program is in a CPU-intensive phase, with the other three being in memory intensive phases (CMMM), a shared resource such as the last level cache could be mostly occupied by the memory intensive applications as they have a larger working set. This default allocation is not optimal because the memory intensive programs might access relatively large amounts of data without much reuse, which may flush out the frequently used data of the CPU-intensive program, thereby degrading its performance. A more efficient system may prefer to exclusively allocate most of the LLC to C0 (as shown in Figure 1), so that its performance is not degraded. Additionally, it may run C0 at a high frequency, and the other three cores at low frequency, since these cores may be stalling frequently due to high memory latency. Since this would create higher traffic on the uncore, the uncore should be run at a high frequency. This LLC-way partitioning decision would change dramatically if C1 changes its phase to CPU-intensive (CCMM). Now most of the LLC must be allocated to the two CPU-intensive cores (C0, C1), as shown in Figure 1. Figure 1 shows some of

Fig. 2. Online RL flow and agent interaction with the architecture.

the possible desired system configurations due to the changing phases of the different programs running on four cores simultaneously.

A CCXX phase combination, corresponding to C0 and C1 being in CPU-intensive phase, with C2 and C3 being in mixed phase, would change the uncore to an average frequency due to mild traffic. This combination of program phases may allocate slightly more LLC to the mixed applications depending on the cache utility to the various processes. A CCCC phase combination would most likely have all cores at a high frequency and uncore at a low/high frequency depending on the private caches showing a low/high miss-rate. An XXXX phase combination may have all cores at average frequency and the LLC being partitioned equally among the processes. In each of these cases, the cache partitioning needs to be based on the utility to the cores. This might need some action-response feedback at runtime to arrive at the best partitioning.

The previous discussion illustrates the complexity of performing multiple optimizations simultaneously, and motivates us to use a machine learning-based approach to perform the runtime adaptation in response to system phase changes.

4 ONLINE RL

RL is a computational model for reward-based learning by interaction with the system. RL is useful in cases when the examples of the desired system behavior are not available but there is a way to interact with the system and sample the observations which can be evaluated as good or bad. RL is particularly useful in problems that require a sequence of decisions to achieve the desired goal.

4.1 Markov Decision Process (MDP) and Q-Learning Algorithm

In RL, an autonomous agent in a dynamic environment learns a good control policy based on the cumulative numerical rewards earned by interacting with the environment. Figure 2 shows how an agent would interact with the environment (architecture) by sending actions (resource reconfiguration), and receiving the reward (measure of optimization target) and the new state (agent's view of the architecture). An RL agent's environment can be described by an MDP. An MDP can be represented by a set of states S , a set of actions A , a state-transition matrix $T(s_t, a_t, s_{t+1})$, and a reward function $R(s_t, a_t, s_{t+1})$, where s_t is the state at interval t , a_t is the currently known best action from s_t .

Q-Learning (Watkins and Dayan 1992) is one of the most popular algorithms for model-free online RL. This implies that an explicit utility model of the environment (architecture) is not required, but rather learned at runtime. This decouples the implementation of the optimization technique from the underlying architecture and enables the hardware designer to focus on the

target optimization and architecture variables to achieve a good control policy. All our proposed models use the utility metrics, which are available as hardware counters on most modern processors such as energy counters and instruction counters. Q-Learning is efficient in finding the best action selection policy even with stochastic state transitions and rewards from the environment. 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 r_{t+1} is the reward received at the start of interval $t + 1$ for performing an action a_t , Q is the Action-Value function, α is the learning rate, and γ is the long term reward discount factor. The learning rate (α) can take a value between 0 and 1, and determines the importance of the newly acquired information over the old information while performing the learning. If $\alpha = 0$, then the agent does not perform any new learning, while $\alpha = 1$ would make the agent to learn only on the new information. The reward discount factor (γ) can take a value between 0 and 1, and determines the importance of the future rewards received by the agent. The Q-Value represents the expected long term reward for various state-action pairs, and the agent learns Q over time by interacting with the environment. The Action-Value function can be implemented as a Q-Table ($|S| \times |A|$ two-dimensional array) with each entry $Q[S_t][A_t]$ representing the quality of the action in a state.

RL-based agents (controllers) are self adapting, and can plan for long term cumulative rewards. An RL-based controller would learn by associating various system states (architecture) and actions (resource reconfiguration) with rewards (metric representing the desired optimization target). The controller exploits the learning by mostly performing the maximum utility reconfigurations and explores the architecture by performing random resource reconfigurations with a small probability. We have used the ϵ -greedy exploration algorithm (Sutton and Barto 1998) in all the experiments.

4.2 Multi-Agent RL-Based Architecture Co-optimization

A coordinated management of multiple resources would require a system to explore a very large resource allocation decision space, making it an unfeasible problem to be solved by a single agent. This problem requires multiple agents to solve the problem in a distributed fashion, which motivates the requirement for multi-agent RL. We have evaluated various controller models for implementing multi-agent RL with Independent Learning and Cooperative Learning. We explore and evaluate three different RL techniques for the co-optimization problem targeting system EDP metric.

An efficient way to model a multiple resource management system is to model individual RL-agents for each of the resources under management. This enables each RL-agent to focus on its target optimization. The multiple agents can be homogeneous or heterogeneous and execute simultaneously to perform their respective optimal resource management. The independent agents are customized toward their respective optimization goals.

5 DVFS OF CORES AND UNCORE

DVFS is an extensively studied optimization technique where the core is run at a lower frequency during periods of low load. This lowering of frequency enables the core to operate at a lower voltage and, hence, save substantial power due to the cubic relationship between power and voltage.

5.1 RL Controller for DVFS

5.1.1 State Function. A state function enables the controller (agent) to view the architecture (environment) and observe the changes due to the requested resource reconfiguration (action). A controller would attempt to execute reconfiguration with the maximum expected utility. DVFS is

a power optimization technique, which operates a system component at different operating frequencies to exploit performance-power tradeoffs. A core can be operated at low frequency during the memory intensive program phase without much performance degradation. A lower core frequency results in lower memory request rate, which could result in higher performance due to lower contention on the shared resources.

The various resource metrics such as cycles-per-instruction (CPI), misses at different cache levels, and memory controller queues can be used to formulate a good state function. However, expanding the parameter set results in exponential growth in the state space. An effective state function should be based on metrics that capture the core performance and global resource contentions. CPI is one such commonly used metric that increases due to the stalls in the processor introduced by branch misprediction, cache misses, and memory access latencies. When applying the core-DVFS technique, CPI is not an appropriate metric due to the changing core frequency. Instead, we propose to use Time Per Instruction (TPI), which is sensitive to the changing core clock cycle time and captures global effects such as contention delays. We model the controller states based on the ratio of the TPIs in the current and previous intervals. The $tr = TPI_{Curr}/TPI_{Prev}$ ratio represents the degree of performance change due to the last frequency change performed by the controller. We propose to quantize the tr value into multiple levels, which influences the state space of the controller. We can choose to increase the number of levels, which would result in a state function that captures the system state in a more accurate way with higher memory storage overhead.

5.1.2 Frequency Reconfiguration (Agent Actions). The agent action space defines how the controller interacts with the architecture. In case of a DVFS controller, the reconfiguration actions should result in change in the operating frequency of the system component. The proposed DVFS controller has three reconfigurations corresponding to increase/no-change/decrease in operating frequency, represented as 1/0/-1, respectively. One could have reconfigurations with multiple V/F level jumps while still remaining independent of the target processor. This approach ensures that the reconfiguration space can be kept small as per the desirable memory overheads.

5.1.3 Reward Function. The reward function defines the motivation for the controller to perform a reconfiguration in a particular system state. The observed reward helps the controller to learn the utility model and identify the desirable reconfigurations from various states. It is important to choose the reward function carefully since a good reward function can help the controller to perform better optimization. An efficient DVFS optimization should tradeoff energy savings with performance degradation. A reward function based on EDP can be an effective motivation. A reward function based on energy-per-instruction (EPI) would motivate the agent for better energy with less importance to performance. An *energy – delay*² product (*ED²P*)- based reward function would motivate the agent to emphasize more on performance than energy saving. The proposed framework can be easily used to perform exploration of the different reward functions.

5.2 Core-DVFS Controller Reward Exploration

As discussed in Section 5.1.1, we model the Core-DVFS controller (Core-Ctrlr) states based on the $tr = TPI_{Curr}/TPI_{Prev}$ ratio. This ratio is quantized to obtain a finite number of states, in accordance with the modeling feasibility. We propose to quantize the tr ratio into 3 states using δ as the performance change factor:

$$State(tr) = \begin{cases} -1 & tr > 1.0 + \delta \\ 1 & tr < 1.0 - \delta \\ 0 & (1.0 - \delta) \leq tr \leq (1.0 + \delta) \end{cases}$$

In this work, we have assumed a target of $<5\%$ performance loss similar to state-of-the-art DVFS (Chen et al. 2013); this motivates the value of $\delta = 0.05$. We perform an exploration on the reward function to study the impact of different motivations. We study three different DVFS models each using the TPI-based core state function with three possible reconfigurations: increase/no change/decrease in the frequency, represented as $+1/0/-1$, respectively. The three models have different reward functions as *epi*, *ed²p*, and *edp*. Figure 8 compares the explored Core-DVFS models with PARC (Juan and Marculescu 2012). As discussed in Section 13.1.4, *edp* model outperforms the other models by exhibiting higher EDP improvement of 9.0%. The *edp* Core-Ctrlr is extended to include branch misprediction into the state function.

5.2.1 Including Branch Misprediction. Branch misprediction (BMP) causes a processor to stall and fetch the instructions for the correct branch and increases the TPI. A processor experiencing high BMP would see increased TPI levels, influencing the Core-Ctrlr to reduce the operating frequency. Branch misprediction for a cpu-bound application must be resolved at the earliest to start execution of the correct instructions. Hence, a cpu-bound application experiencing high BMP should be operated at a high frequency. In case of a memory bound application, the TPI would be dominated by the memory latency and, hence, the system can operate at low frequency even during high-BMP situations. This difference in expected core frequency behavior for high BMP (both resulting in higher TPI/performance degradation) motivated the inclusion of BMP into the Core-Ctrlr state function. We also explored the inclusion of L1D, L2, and L3 cache misses into the state function and BMP appeared to be a high-priority attribute during our workload study.

We extend the previously mentioned Core-Ctrlr by considering the effect of branch misprediction on the core frequency. We introduce three states for representing BMP by quantizing the branch misprediction per kilo instructions (bmpi) into three levels. These three additional states are used along with the three TPI-based states, resulting in a total of nine states to represent a core agent. As discussed in Section 13.1.5, TPI-BMP-based Core-Ctrlr performs better than the TPI-based Core-Ctrlr by exhibiting 10.2% EDP improvement (Figure 8). The core DVFS model can be extended by including other metrics such as uncore latency measurements (Miftakhutdinov et al. 2012).

5.3 Uncore DVFS Controller

The uncore frequency influences the packet transmission time through the NoC and the LLC access latency. If the applications are experiencing high uncore traffic, then the uncore is expected to be operating at high frequency, so that the uncore requests can be transmitted to the LLC quickly. With frequent incoming requests on the LLC, the access latency gains importance. Similarly, if the applications are experiencing low uncore traffic, it could allow the system to operate the uncore at a lower frequency depending on the LLC throughput sensitivity on the applications. Since the uncore frequency impacts the overall system performance, we propose to use an uncore DVFS controller similar to a Core-Ctrlr with *tr* being the arithmetic mean of all the cores' individual *tr* values. This uncore *tr* is quantized into three states and the reward is the average TPI of all the cores for better system throughput. The Uncore-Ctrlr actions are the same as the ones discussed previously for the Core-Ctrlr: increase/no-change/decrease in the frequency and voltage, represented as $+1/0/-1$, respectively.

6 DYNAMIC PARTITIONING of LLC

A shared LLC can help in better application performance as each sharing core can potentially use the full cache during the execution phase of a large working set. However, this could also degrade the performance of cores if there is a high contention for the LLC. Partitioning a shared cache can be an effective technique; here, different fractions of the cache are made available to different

cores. This allocation ensures that each sharing core is often able to retain a desired working set in the shared cache, isolated from other cores' cache requirement. Efficient management of the LLC can significantly affect the overall system performance.

The on-chip caches commonly use the Least Recently Used (LRU) or some approximation of LRU as the replacement policy. The LRU policy attempts to exploit the temporal locality of the running applications and implicitly partitions the shared cache among the competing applications on the basis of demand. The overall utility of the cache to an application may not correlate with its memory requirement. For example, a streaming application would read in a lot of data into the cache without much reuse.

Several researchers have proposed ways to perform DCP of shared caches by allocating cache ways to individual cores (Qureshi and Patt 2006; Gordon-Ross and Vahid 2007; Moreto et al. 2008; Yu and Petrov 2010). Figure 3 shows one such configuration where some cache ways are exclusively assigned to cores.

6.1 LLC Fixed Shared Ways Approach

Way partitioning a shared cache can potentially lead to fragmentation, with some cores being allocated more cache than required. This is more prevalent for applications with very low cache footprint. To benefit from the exclusive cache allocation and avoid fragmentation, a hybrid approach to cache architecture is required. With a hybrid approach, certain cache ways are always shared and cannot be allocated exclusively to any core. This approach addresses situations with relatively unbalanced cache requirements from cores, permitting cache hungry applications to use most of the exclusive cache ways and still allowing them to occupy the shared ways if needed.

The read and write operations for the hybrid cache are identical to the traditional LRU-based cache. The cache controller still searches all the ways for the data and performs the read and write operations. Hence, this proposal does not impact the cache timing for the cache hit cases. The hybrid architecture impacts the operation only when there is a miss and a replacement needs to be performed. Now the cache controller needs to read an additional bit vector to find the legal cache ways available to the core for replacement. Since an LLC miss leads to an off-chip DRAM access, the miss latency is very high and, hence, the additional bit vector reading is not on the critical path.

Additionally, DCP needs to be a fast operation as it may be performed after every execution interval; the current allocation may not be an optimal decision, and may need to be rolled back if not found to result in a positive reward. To achieve this, we allow weak-exclusiveness of the cache ways, where the deallocated cache ways are not invalidated. This saves the overhead of writing back the dirty blocks to the lower cache levels, and in case of a rollback, results in a low penalty for the originally owning core.

6.2 RL Controller for Dynamic Cache Way Allocation

6.2.1 State Function. One of the main challenges of RL is an effective and efficient way to model the controller. A W -way set-associative shared cache, with F fixed shared ways, has $E(= W - F)$ ways available for exclusive allocation between the C cores sharing the cache. The effective number of ways this can be done is C^E . Assuming a 16-way cache shared between four cores, with four ways as fixed shared, we have a total of 4^{12} possible cache configurations. Due to this large number of possibilities, the agent states cannot be directly based on the distinct cache configurations. We require a technique to model the shared cache state, which can capture the cache reconfiguration changes on all the sharing cores and still result in a feasible state space. We propose to model the controller state based on the states of the cores that share the cache. For a C core shared cache, with each core being in one of S states, the total states for the cache are S^C . For this modeling to be feasible, the number of core states being used to collectively represent the LLC state must be small.

6.2.2 Cache Reconfiguration Action. The actions of the controller would result in cache way reassignment. Since there can be a large number of possible cache reconfigurations, the action space must be constrained to certain reconfigurations, which are general enough to be feasible from each cache state and still are not too large. We propose to have the cache reconfiguration action be a vector representing way allocation/no change/deallocation (1/0/-1) request per core. A C -core shared cache would have its reconfiguration represented by a C -element vector. For example, for a 4-core system, $[1,0,0,-1]$ represents a reconfiguration where Core C0 is allocated an additional cache way by deallocating from Core C3. With the fixed shared cache ways technique, if a core is allocated an additional cache way, another core must be deallocated a cache way. Under this constraint, the total number of valid reconfigurations for a C -core shared cache is given by $\sum_{i=0}^{C/2} \binom{C}{i} \binom{C-i}{i}$. This action space is independent of the cache associativity and only determined by the number of cores sharing the cache. For a 4-core system sharing an LLC, there are 19 valid reconfigurations.

6.2.3 Reward Function. The reward function would typically correspond to the runtime measurable metric that represents the intended optimization target. An LLC can be optimized for improving the overall system performance, reducing the LLC misses, reducing the off-chip traffic, EDP, and so on.

6.3 LLC Controller Reward Exploration

The LLC has an independent RL controller (LLC-Ctrlr), which performs DCP among the four cores sharing the cache. We have explored different reward functions and performed a comparative study for the various LLC-Ctrlrs. LLC-Ctrlr state is represented using a vector of the TPI-based core-states (Section 5.1) with 19 reconfiguration actions (Section 6.2). The three models have different reward functions as off-chip traffic (LLC1), total executed instructions (LLC2), and system EDP (LLC3).

The LLC1 (Jain et al. 2016) model attempts to reducing the off-chip traffic as an effective metric for performing DP-LLC (Yu and Petrov 2010). The model titled LLC2 is aimed to focus more on the overall system throughput by rewarding reconfigurations leading to higher number of executed instructions. Hence, the LLC2 model's reward function is an arithmetic mean of the executed instructions on the cores. LLC2 performs better than LLC1 in terms of all the three measured system level metrics of EDP, STP, and Fairness. The third LLC-Ctrlr (LLC3) is motivated to optimize for a better system EDP metric. The LLC3 model results in better system EDP, STP, and Fairness metrics, as shown in Section 13.1.6. Figure 9 shows the average system EDP, STP, and Fairness measured on 20 different 4-benchmark workloads (Section 12.2). These results are discussed in more detail in Section 13.1.6.

7 CO-OPTIMIZATION SYSTEM WITH MULTI-AGENT INDEPENDENT LEARNERS

Figure 3 shows the co-optimization system architecture, Machine Learned Machine (MLM), used for the study in this work. The 4-core system has a controller running on each of the four cores for performing DVFS. The uncore (NoC+LLC) has an independent controller performing DVFS. The LLC has an independent controller performing dynamic cache partitioning. All the six controllers execute independently and do not explicitly coordinate with each other. The multiple independent controllers sense the changes to the environment caused by the reconfiguration performed by all the controllers and execute their best actions. Section 9 explores different co-optimization models using various combinations of Core-Ctrlrs, LLC-Ctrlr, and Uncore-Ctrlr.

Fig. 3. Co-optimization architecture with various independent RL controllers.

8 CO-OPTIMIZATION WITH INFORMATION SHARING COOPERATIVE LEARNERS

We propose techniques to enable the controllers to share information among each other for more effective reconfigurations. We also perform an exploration of the co-optimization system by proposing and evaluating different MLM techniques.

The independent learning multiple agents (Section 7) do not explicitly cooperate with each other, but sense the changes to the architecture caused by the reconfiguration performed by the other agents (controllers). We have extended this MLM proposal to enable cooperation among the agents. We propose to use the agent action broadcasting technique for cooperation among the multiple agents.

Tan (1993) showed that multiple agents can cooperate among each other by sharing their experiences. If this additional information can be integrated into the agent models effectively, this can result in significant learning improvement for the agents. In our MLM system, the LLC cache partitioning decisions impact the performance of the individual applications. This impact can be observed in the TPI values for each core. The Core-Ctrl executing on each core senses the TPI value to perform DVFS optimizations. On an independent learning multi-agent setup, the Core-Ctrl is not aware of the LLC-Ctrl and its impact on the application performance. The Core-Ctrl assumes that all TPI changes being observed are due to the DVFS optimizations being performed by itself. If the Core-Ctrl can be made aware of the LLC-Ctrl and its impact on the TPI, then the Core-Ctrl would be able to infer better learning.

We propose to perform the information sharing by broadcasting the LLC-Ctrl reconfiguration to the Core-Ctrl. Since the LLC-Ctrl reconfiguration results in cache reallocation per core, this cache size reallocation information can be used by the Core-Ctrl to learn its effect on the TPI values. Now the Core-Ctrl is aware of another external factor that can result in TPI changes. This additional LLC reconfiguration information is integrated into the Core-Ctrl state function.

8.1 Information Sharing Cooperative Core-Ctrl Model

The information sharing cooperative Core-Ctrl results in changes to the state function only. The reconfiguration actions and reward function are the same as the ones discussed in Section 5.

8.1.1 State Function. In Section 6.2, the proposed LLC-Ctrl state function is a vector of the state of the cores sharing the LLC. This enables a flow of information from the cores to the LLC-Ctrl. Further, the Core-Ctrl state is based on TPI. This TPI value can change with changing

Fig. 4. Information flow between Cooperative LLC-Ctrlr and Core-Ctrlr.

Table 1. Co-optimization Exploration

Co-opt Model	Core-Ctrlr (state, reward)	Uncore-Ctrlr (state, reward)	LLC-Ctrlr
MLM1	<tpi, bmp>, edp	tpi, tpi	LLC1
MLM2	<tpi, bmp>, edp	tpi, tpi	LLC2
MLM3	<tpi, bmp>, edp	tpi, tpi	LLC3
coMLM	<l3action, bmp, tpi>, edp	tpi, tpi	LLC3

LLC allocation to the individual cores. A larger LLC allocation to a core can result in higher throughput and, hence, reduced TPI. We propose to inform the Core-Ctrlr about the cache allocation changes at LLC, which can enable it to learn about the impact on TPI with changing frequency along with the cache allocation. The LLC-Ctrlr reconfiguration consists of the individual cache allocation decisions for the cores. We propose to communicate the core about this cache allocation change as shown in Figure 4. As discussed in Section 5, the Core-Ctrlr state is a vector of quantized values of branch misprediction and TPI. The TPI information from all the sharing cores is used by the LLC-Ctrlr to determine its state vector. As shown in Figure 4, the LLC state of [0,-1,1,0] is determined from the TPI information of the four cores. The LLC-Ctrlr based on the state and reward received searches for the best action of [0,1,0,-1]. This action would result in Core2 being allocated an additional LLC way, Core4 being deallocated an LLC way, and no change in the LLC allocation for Core0 and Core3. The LLC allocation information is passed to the respective cores, which use it as part of the Core-Ctrlr state as shown in Figure 4.

9 MLM CO-OPTIMIZATION EXPLORATION

Section 5 discussed the DVFS controllers for the cores and uncore. Section 6.3 proposed and evaluated three different models for LLC-Ctrlr. While building the co-optimization system, the various RL-controllers can be treated as different building blocks, which can be used in various combinations for performing multiple optimizations.

Table 1 shows the various different co-optimization models explored in this article. MLM1, MLM2, and MLM3 co-optimization models correspond to the multi-agent independent learning setup. All three co-optimization models use the TPI-BMP-based Core-Ctrlr (Section 5.2.1), and a TPI-based Uncore-Ctrlr (Section 5.1). MLM1, MLM2, and MLM3 co-optimization models use the LLC1, LLC2, and LLC3 models (Section 6.3), respectively, for LLC-Ctrlr. Section 13.1.6 discusses

Actions	a1	a2	a3	a4	a5	Agent	1	2	3	4	QSum
$Q^1[s_t^1]$	<u>0.7</u>	0.6	0.7	0.8	1.0	JA1	a1	a3	a2	a5	4.8
$Q^2[s_t^2]$	0.7	0.8	1.1	<u>1.0</u>	1.0	JA2	a1	a4	a5	a1	<u>6.2</u>
$Q^3[s_t^3]$	1.0	1.6	1.8	1.8	<u>2.0</u>	JA3	a5	a1	a2	a2	5.9
$Q^4[s_t^4]$	<u>2.5</u>	2.6	0.7	0.8	1.4	JA4	a4	a3	a3	a4	4.5

Fig. 5. Finding the best coordinated joint actions.

the comparison of the three MLM models and shows that MLM3 is able to outperform the other two models.

We have extended the MLM3 model to a cooperative MLM (coMLM) model by implementing the information flow between the LLC-Ctrlr and the Core-Ctrlrs as discussed in Section 8, resulting in better system metrics. The experimental results are discussed in more detail in Section 13.1.

10 CO-OPTIMIZATION USING JOINT ACTION MULTI-AGENT COOPERATIVE LEARNERS

As discussed in Section 7 and Section 8, the proposed independent learning and information sharing cooperative learning multi-agent techniques have been effective for the co-optimization problems. The agents perform their respective actions independently and the agent decisions are often greedy. This greedy optimization by the multiple agents may not always result in superior global optimization. In this section, we propose a technique to enable the agents to work together by choosing a coordinated joint action to perform global optimization. This is achieved by performing distributed learning for the multiple agents combined with a central controller, which enables the distributed multiple agents to execute globally best actions.

10.1 Coordinated Multi-Agent Joint Actions

The multiple agents in a coordinated environment are required to perform the actions which are globally best. In such a setup, the agents must search for the actions jointly. This can be achieved by executing a central controller that can search for the best joint actions and communicate them to the agents.

Figure 5 shows an illustrative example for a system at timestep t with four controllers in states $s_t^1, s_t^2, s_t^3, s_t^4$, respectively, and five possible independent reconfiguration actions: $a1, a2, a3, a4$, and $a5$. Figure 5 shows the Q-Table entries for the possible reconfiguration for each controller in its respective state. A system with non-coordinating controllers would result in each controller picking the maximum Q-value action from its respective state. In the example, this would result in a joint action of $\langle a5, a3, a5, a2 \rangle$, 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 reconfigurations, then they can observe the correct rewards for their respective actions. Additionally, the controllers can work together to search for a joint action expected to maximize the system utility. Figure 5 shows four different feasible joint actions, JA1, JA2, JA3, and JA4 for the system along with the respective Q-value sums. In this example, JA2 is found to be the best joint action with 6.2 units of utility.

11 RL MODEL FOR JOINT ACTION COOPERATIVE LEARNING

We propose a controller for coordinated joint action. In the proposed system, each core has an independent controller optimizing the EDP for its application.

ALGORITHM 1: Best Joint Action Search

Input: $Q[S_i]$ for each $Agent_i$ on $Core_i$, for $i \in 1, \dots, n$
Output: Best Joint Action Vector $A[1..n]$

```

/* Q-value represents  $1/EDP_{Utility}$ , maximum Q-value corresponds to minimum EDP */
1  CurrentJA =  $[A_1, \dots, A_n]$ ,  $A_i = \max_a(Q[S_i][a])$  // seed with minimum EDP actions
2  makeValid(CurrentJA) // balance the total allocation and deallocation requests
3  index = 0, MaxQSum = 0, BestJA =  $[0, \dots, 0]$ 
4  repeat
5    mutate(CurrentJA) // randomize actions or change a random agent's action
6    makeValid(CurrentJA) // balance the total allocation and deallocation requests
7    QSum =  $\sum_{i=1}^n (Q[S_i][CurrentJA[i]])$  // evaluate new solution as sum of utility
8    if MaxQSum < QSum then
9      BestJA = CurrentJA // store a better joint action solution found
10     MaxQSum = QSum
11  end
12  until (index + 1) > Iterations
13  return BestJA // return the best joint action found

```

11.1 State Function

The state function is required to capture the system state such that it is representative of all the resources being managed by the controller. As discussed in Section 5.1.1, $tr = TPI_{Curr}/TPI_{Prev}$ ratio of the application executing on the core is a good metric for capturing the system state with varying operating clock frequency. The LLC reconfiguration can result in changes to the tpi values; hence, we propose the controller state as a vector of TPI_{Curr} and tr ratio, each quantized into five levels, resulting in 25 states.

11.2 Reconfiguration Actions

A multi-resource reconfiguration action is a vector *Action* representing the allocation and deallocation requests for the core DVFS and LLC space. The reconfiguration action has two elements, and is represented by $Action = \langle f, c \rangle$, where integer $f \in \{-v_n, 0, +v_n\}$ representing the change in V/F levels, and $c \in [-w_n, +w_n]$ refers to the requested change in the number of ways for the LLC. The w_n is the maximum number of cache ways that can be reassigned in a single step. The controller can request for multiple cache way allocations at the LLC, which enables it to explore cache sizes greater than the working set of the application. Positive and negative numbers in the previous vector represent the allocation and deallocation of the resources.

We propose to perform the reconfiguration of only one resource in each interval to enable the controller to learn the resource reallocation sensitivities. This simplifies the reconfiguration to $\langle f, 0 \rangle$ and $\langle 0, c \rangle$. In our system, we have used $v_n = 1$ and $w_n = 4$, resulting in reconfiguration actions as: $Action(Core) = \{-1, 0, 1\}$ and $Action(LLC) = \{-4, -2, -1, 0, 1, 2, 4\}$. Hence, the multi-resource reconfiguration actions are: $Actions([Core, LLC]) = \{[0, 0], [-1, 0], [1, 0], [0, -4], [0, -2], [0, -1], [0, 1], [0, 2], [0, 4]\}$. The previous action space has a total of nine reconfigurations.

11.3 Reward Function

The main objective of the controller is to reduce the EDP for its application. Hence, we have used the EDP-based reward to motivate the controller toward lower system EDP. To compute

the energy consumption per application, the private resources such as the CPU, caches, and translation-lookaside buffers (TLBs), can be easily accounted. The energy consumption of the shared resources such as LLC, interconnects, and DRAM, have been accounted for as per the request count by the respective cores. The TPI during the simulation interval represents the delay component for the application.

This RL controller would be motivated to secure the maximum possible LLC space, resulting in non-optimal reconfigurations for the overall system performance. Additionally, an LLC is a limited shared resource, and the system cannot allocate all the requested cache to all the controllers. This problem can be solved if the controllers can coordinate to work together toward a common goal of lower system EDP.

11.4 Joint Reconfiguration Action Search

We propose a coordinated joint reconfiguration action selection where a central controller decides the reconfiguration for the various RL-Ctrlrs. A Q-Table entry, $Q[s][a]$ represents the expected EDP Utility for the various reconfigurations by the RL-Ctrlr. The central controller has access to the Q-Tables and attempts to select a joint action for the minimum EDP. Algorithm 1 shows the proposed *Hill Climbing* algorithm to perform the joint action search for the co-partitioning problem. Algorithm 1 starts with an initial joint reconfiguration computed by picking the minimum EDP actions for each RL-Ctrlr, and converts it into a valid joint action by balancing the combined allocation and deallocation LLC requests (*makeValid* function on Line 2). In each iteration, the *CurrentJA* is randomized or changed in a small step (*mutate* on Line 5). To change *CurrentJA*, an RL-Ctrlr 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 RL-Ctrlrs' actions. The new joint action is evaluated (Line 7) as the sum of the expected EDP Utility for the reconfiguration in the *CurrentJA*. If the new joint action is better, then it is stored as the *BestJA* found so far (Line 9). The quality of a hill climbing search solution depends on the number of iterations performed to find the best solution. In our experiments we have used 100 iterations for a 4-core system.

12 EXPERIMENTAL SETUP

12.1 Simulation Setup

For the experimental validation of our proposed techniques, we used the Sniper Simulator (Heirman et al. 2012) integrated with McPAT (Li et al. 2009), an architecture power estimation tool. We tuned various Sniper and McPAT parameters using Spec2006 benchmarks to power/performance data on a commercial processor with a similar configuration. We have used the DRAM power estimation model provided with the Sniper-McPAT integration. We have used the *pybrain* (Schaul et al. 2010) python library for the machine learning algorithm implementation. All the RL-agents use the $\alpha = 0.5$ (learning rate), and $\gamma = 0.75$ (reward discount factor). All the results presented in Section 13 are normalized to the system EDP, STP, and Fairness metric of the baseline system.

12.1.1 Baseline System Configuration. Table 2 shows the system configuration (similar to Intel Nehalem architecture) we used to perform the experiments. The caches are strictly inclusive, i.e., any data in the higher cache levels must be present in the lower cache levels. We have used the sniper DRAM model with 60ns access latency (~ 200 CPU cycles), similar to other state-of-the-art works.

Table 2. 4-core Baseline System Configuration

Architecture	Intel Nehalem
Cores	3.2GHz, 4CPU+L1+L2
L1 Caches	Private L1-D (8-way), L1-I (4-way), 32KB
L2 Cache	Private Unified, 128KB, 8-way
Uncore	NoC+L3, 3.2GHz
L3 Cache	4-Core Shared, 2MB, 16-way
Core/Uncore DVFS Table	<i>GHz</i> 3.2 3.0 2.7 2.4 2.0 <i>V</i> 1.2 1.1 1.0 .9 .8
DRAM latency	60ns (~ 200CPU cycles)

Table 3. Spec2006 Benchmark Classification

Type	Benchmarks
cpu-bound	c wrf, bzip2, gromacs, gobmk, xalancbmk, hmmer, sjeng, h264ref, povray, calculix, namd, perlbench, games
mixed	x leslie3d, GemsFDTD, libquantum, gcc, sphinx3, lbm, zeusmp, bwaves, cactusADM
memory-bound	m mcf, astar, soplex, milc, omnetpp, tonto

Table 4. 4-Benchmark Workloads Used for Experiments

#	Mix	Workload	#	Mix	Workload
1	cccc	xalancbmk, hmmer, sjeng, h264ref	11	xccm	GemsFDTD, hmmer, h264ref, milc
2	mmmm	mcf, astar, soplex, milc	12	xccm	sphinx3, calculix, perlbench, astar
3	xxxx	leslie3d, GemsFDTD, libquantum, gcc	13	xxcm	bwaves, libquantum, wrf, omnetpp
4	xxxx	sphinx3, lbm, zeusmp, bwaves	14	xxcm	cactusADM, GemsFDTD, calculix, mcf
5	xxxc	zeusmp, bwaves, cactusADM, xalancbmk	15	xxmm	libquantum, sphinx3, mcf, soplex
6	ccmm	games, bzip2, omnetpp, astar	16	xxmm	zeusmp, cactusADM, omnetpp, astar
7	cmm	xalancbmk, h264ref, tonto, soplex	17	xxmm	cactusADM, leslie3d, omnetpp, tonto
8	cccc	bwaves, perlbench, games, wrf	18	xxcc	bwaves, leslie3d, perlbench, wrf
9	cccc	lbm, povray, calculix, namd	19	xxcc	lbm, libquantum, namd, gromacs
10	cccc	leslie3d, gromacs, xalancbmk, sjeng	20	xxcc	zeusmp, gcc, hmmer, povray

12.1.2 Reconfiguration and Power Estimation. The system simulation is divided into fixed-time intervals of 1.56ms, which is approximately 5 million cycles on a 3.2GHz processor. This is similar to the fixed-time intervals considered in other studies (Qureshi and Patt 2006). A reconfiguration penalty of 2μ s (Intel 2011) along with new configuration search time (Section 13.2.1) is added at the end of each fixed-time interval. McPAT is used to perform the power estimation at the end of each fixed time interval. To account for DVFS, the maximum frequency circuit power numbers are scaled appropriately depending on the actual operating voltage and frequency of the components.

12.2 Benchmarks

Sniper supports PinPoint (Patil et al. 2004), which is the SimPoint methodology (Sherwood et al. 2002; Perelman et al. 2003) using the Intel Pin tool (Luk et al. 2005). A single 250 million instruction PinPoint (Pinball), which is a representative and repeatable program region, is identified for each Spec2006 benchmark for simulation. The benchmarks are classified into three categories based on the measured CPI as: cpu-bound, memory-bound, and mixed. The various benchmarks in each category are listed in Table 3. Table 4 lists the 20 4-benchmark workloads created with different combinations of benchmark categories. Each benchmark in the three categories participates equally to avoid any bias toward a particular benchmark in a category. The workloads are simulated until each benchmark has completed at least 250 million instructions. A benchmark completing its execution of the 250 million instructions earlier than other benchmarks is restarted to ensure that all cores continue to contend for resources. This setup is similar to the experimental setups used by the state-of-the-art architecture research work.

12.3 Metrics

12.3.1 System EDP. EDP is a common metric used to measure the joint impact of architecture optimizations on both energy and performance for battery operated devices. The EDP measure

Fig. 6. System EDP improvement and STP, Fairness degradation for 4-core workloads.

correctly captures the situations where energy saving is accompanied by a significant delay penalty. Since instruction is the basic unit of work performed by the processor, the system EDP can be given as $EDP = (time/instruction) \times (energy/instruction)$.

12.3.2 System Throughput (STP). While evaluating multiprogram workloads, evaluation based on raw IPC values could be misleading (speeding up high-IPC applications at the expense of others leads to better average IPC). Eyerman and Eeckhout (2008) proposed system throughput, which considers weighted IPC values, thereby measuring the speedup or slowdown in comparison to the application running in a stand-alone mode. State-of-the-art works also call this weighted speedup, given as $STP = \sum_i \frac{IPC_i^{Shared}}{IPC_i^{StandAlone}}$.

12.3.3 Fairness. Luo et al. (2001) proposed computing the harmonic mean of the IPC values as a notion of Fairness. The harmonic mean of the weighted IPC values is given as $Fairness = \frac{n}{\sum_i \frac{IPC_i^{StandAlone}}{IPC_i^{Shared}}}$.

13 RESULT ANALYSIS

13.1 Co-optimization System

13.1.1 Comparison of Different MLM Models. Table 1 shows the different co-optimization models we explored. Figure 7 shows the comparative results for the different co-optimization models. All the metrics reported are normalized to the baseline system configuration (Section 12.1.1). MLM1, MLM2, and MLM3 use the TPI-BMP-based Core-Ctrl (Section 5.2.1) with LLC1, LLC2, and LLC3 (Section 6.3) as the LLC-Ctrl model, respectively. We evaluated the three independent learning MLM models with the 20 workloads (Section 12.2). On an average, MLM3 performs better than

Fig. 7. Average system metrics for various co-optimization techniques on a 4-core system.

MLM1 and MLM2 on all the three measured system metrics. MLM3 shows a system EDP improvement of 17.1%, compared to 14.3% and 16.6% with MLM1 and MLM2. MLM3 is able to achieve this with 4.7% (4.5%) penalty on the STP (Fairness), compared to a 4.8% (4.6%) and 5.2% (5.0%) penalty with MLM1 and MLM2. MLM3 is extended to the cooperative information sharing MLM model called coMLM (Section 8). The coMLM model is able to outperform MLM3 by improving the system EDP by 20.1% with 4.7% penalty on STP and Fairness. The cooperative joint action model JMLM (Section 10), performs slightly better than the coMLM model by improving system EDP by 20.5% with a 2.3% and 2.6% penalty on STP and Fairness.

13.1.2 Comparison with Other Work. The proposed co-optimizations have been compared with CMIR (Bitirgen et al. 2008) and XCH (XChange) (Wang and Martínez 2015). The state-of-the-art techniques are proposed for optimization under a power budget. Since these proposals address the problem of multiple resource management, we have modified them for EDP, and the implementation can be considered “CMIR-based” and “XCH-based.” As shown in Figure 7, all MLM models are able to outperform both CMIR and XCH.

CMIR proposed a supervised learning approach using the ANN-based framework for performing multiple resource allocation. The proposed technique consists of periodic online supervised training phases. We implemented the ANN-based co-optimization (allocation of core frequency, uncore frequency, and LLC ways) targeting system EDP optimization. The CMIR technique showed a system EDP improvement of 11.75% with a high penalty of 9.4% and 9.2% on STP and Fairness, respectively. CMIR trains a neural network periodically and later uses this trained network to predict the system EDP. The learning of the ANN depends on the ability of the network to generalize the prediction function. An ANN prediction would be sub-optimal once the application changes their execution phase and requires a retraining of the ANNs. The ANN would try to generalize this new data to its current learning, resulting in changes to the various neuron edge weights. This generalization can result in loss of earlier learning. The RL behind MLM is able to adapt to changing application phases much better since a change in an application phase results in a different system state and, hence, the Q-Learning updates are performed to a different row of the Q-Table. This preserves the earlier learning, which can be exploited again once the application experiences the previously learned phases.

XCH is an economic market based framework to perform multiple resource allocation using competitive bidding by the various agents. XCH has proposed cache utility and power utility functions, and performs LLC partitioning and core DVFS for maximum throughput within a power budget. The proposed utility functions compute the expected application execution latency (t_{exe}) and

memory phase (t_{mem}) as a function of the core frequency and cache allocation, respectively. The expected energy (E) is computed by scaling the current energy hardware counter values with the new evaluation frequency. The XCH power utility function is given by $P = \frac{E}{t_{exe} + t_{mem}}$ and the agents bid under a fixed power budget. In our setup, the EDP utility would be $EDP = E \times (t_{exe} + t_{mem})$ and the agents bid under a fixed average frequency budget. The problem setup tolerates around 5% STP penalty, hence, the frequency budget is set as 95% of the baseline frequency (3.2GHz). The average of the cores' frequency in each interval is kept less than or equal to 3.04GHz. Additionally, an uncore DVFS was implemented, which is similar to the uncore DVFS proposed in Juan and Marculescu (2012). We have used the wealth redistribution model of the XChange technique. XCH was able to show a system EDP improvement of 12.9%, with 5.25% and 5.75% penalty on STP and Fairness, respectively. XCH technique requires the agents to bid competitively with each other, which can result in sub-optimal decisions at the global level. All our proposed techniques do not result in a competitive bidding among the agents. The LLC-Ctrl performs a centralized LLC partitioning attempting to optimize the reward metric at the global level. Also, the Core-Ctrls are free to choose the best operating frequency independent of the other core frequencies. XCH cannot perform DVFS under this configuration as it would result in all cores operating at the maximal frequency, which constrains the agents by allocating them wealth. The JMLM technique encourages the multiple agents to jointly work toward a global optimization.

13.1.3 Individual Workload Results. Figure 6 shows the comparison of the evaluated system metrics for the individual 4-benchmark workloads. We have compared the Core-DVFS (TPI-BMP-based model, Section 5.2.1), Dynamic LLC partitioning (LLC3 model, Section 6), Independent MLM (MLM3 model, Section 7), Co-operative information sharing MLM (coMLM, Section 8), Joint action MLM (JMLM, Section 10), and XCH (Wang and Martínez 2015). We observe that MLM, coMLM, and JMLM are able to outperform the Core-DVFS and DCP on most workloads. Also, the savings on individual optimizations are not always additive when performing co-optimization.

cccc1 workload showed very less LLC and DRAM requests resulting in DCP technique being ineffective. The ineffectiveness of DCP did not help the information sharing between the LLC-Ctrl and Core-Ctrl of the coMLM, resulting in a lower EDP saving of 7.6% (coMLM) compared to 10.4% (MLM). The co-optimizations were able to save energy by performing aggressive uncore DVFS.

xxxx4 workload exhibited no EDP improvement with DCP (-0.5%), but the MLM (20.1%), coMLM (26.4%), and JMLM (26.9%) were able to outperform without much STP degradation. The Core-DVFS resulted in a lower request rate to the DRAM. This reduced the average DRAM read queuing delay from 82.5ns (baseline and LLC) to less than 72ns for MLM, coMLM, and JMLM. The reduced request rate from the cores enabled the Uncore-Ctrl to perform effective DVFS on the uncore, which could be operated at an average frequency of 2.43GHz (MLM) and 2.3GHz (coMLM and JMLM).

cmm6 workload experienced lesser EDP improvement with MLM (16.5%) compared to DCP (19.6%), but the coMLM (22.5%) and JMLM (21.6%) were able to outperform DCP, while XCH (1.2%) provided low EDP improvement. DCP optimization using our LLC3 model exhibited lower mpki at LLC and 15% lower DRAM requests, resulting in 6.5% improvement in STP. MLM, coMLM, and JMLM exhibited 5%, 15%, and 12% lower DRAM requests, which enabled effective DVFS. XCH resulted in 20% higher DRAM requests than the baseline, which resulted in low average core frequency of 2.2GHz with 19% STP degradation.

cccc9 workload performed well with the DCP resulting in 12% reduction in DRAM requests. The DCP effectiveness helped the coMLM technique to perform effective DVFS and LLC partitioning resulting in 15% reduction in DRAM requests, outperforming MLM (9.5%), JMLM (13%), and XCH (10%).

xxmm15 workload showed a degradation in EDP due to DCP (-1.7%). The DCP and all co-optimization techniques resulted in more than 1% additional DRAM requests. MLM (19%), coMLM (19.4%), and JMLM (22.6%) are still able to outperform the core-DVFS (12.6%) EDP improvement due to additional saving from uncore DVFS. The various MLM techniques operated the core and uncore between 2.4–2.6GHz. XCH operated the core and uncore at 2.8GHz and 3.1GHz resulting in 11% EDP savings.

xxcc19 was the only workload where XCH outperformed all proposed MLM techniques. XCH was effective in performing cache partitioning resulting in 15% reduction in DRAM requests compared to MLM (10%), coMLM (10%), JMLM (14%), and DCP (14%). This enabled all the co-optimization techniques to perform effective DVFS and still improve the STP. Since DCP only performed cache partitioning, it showed maximum STP improvement of 11.5% compared to 8.7% for XCH.

13.1.4 Core DVFS Exploration Results. Figure 8 compares the average system metrics for the various DVFS-Ctrlrs explored in Section 5.2 on the 20 4-core workloads. The explored DVFS-Ctrlr uses the TPI-based state model with three actions as discussed in Section 5.2 with reward functions as *epi*, *ed²p*, and *edp*. The DVFS-Ctrlr motivation is different for the three DVFS models, and this impacts the evaluation metrics. The agent with *epi* as the reward function would be motivated to perform the DVFS with the least importance to performance, while the *edp* and the *ed²p* reward models are expected to optimize for energy with varying emphasis on performance. The *epi*, *ed²p*, and *edp* motivated models resulted in 7.7% (5.6%), 8.3% (5.0%), and 9.0% (5.4%) EDP (STP) improvements (degradation), respectively. This is in line with the expectations and the *edp* motivated model outperforms other models on the EDP metric. The *epi* motivated model resulted in more STP degradation. Since the co-optimization proposals are intended to optimize the system EDP, we have extended the *edp* motivated model to include the branch misprediction (Section 5.2.1). As shown in Figure 8, the extended DVFS-Ctrlr labeled as *edp_{bmp}* outperforms the other three DVFS-Ctrlrs by exhibiting 10.2% EDP improvement with 4.7% STP and Fairness degradation.

The proposed DVFS-Ctrlrs are compared with the PARC (Juan and Marculescu 2012) technique as shown in Figure 8. The PARC model is an RL model motivated for EDP to perform a fair comparison with our proposal. Our proposed EDP motivated models *edp* and *edp_{bmp}* outperform PARC in all the three evaluation metrics.

13.1.5 Core DVFS with Branch Misprediction. We analyzed the various workloads to understand how the agent handles high BMP. The *xxxx10* WL has two cpu-bound applications with high BMP, *gromacs* (8.6%), and *sjeng* (12.4%). The *edp_{bmp}* DVFS controller operated the two applications at an average frequency of 2.65GHz and 3.1GHz, while the *edp* DVFS controller operated at 2.5GHz and 2.4GHz, respectively, resulting in better EDP and STP. We found that the agents optimizing CPU bound applications had a tendency to operate at a higher frequency. The *xxmm15* WL applications experienced medium to high BMP of 6.94%, 2.15%, 11.82%, and 9.08%. The *edp_{bmp}* and *edp* DVFS controllers operated at [2.44, 2.66, 2.55, 2.49] GHz and [2.71, 2.73, 2.72, 2.62] GHz, which resulted in additional 5% EDP savings. We found that the non-CPU bound applications with high BMP tend to operate at a lower frequency since their TPI is usually dominated by the long latency memory accesses.

13.1.6 DCP Model Exploration. Figure 9 compares the different LLC-Ctrlr based on system EDP, STP, and Fairness. The reported values are the mean of the metrics normalized to the baseline configuration for the 20 workloads used for experimentation. LLC1, LLC2, and LLC3 are the various LLC-Ctrlrs targeted toward optimizing off-chip bandwidth, executed instructions, and system EDP, respectively. LLC3 outperforms LLC1 and LLC2 by exhibiting an average system

Fig. 8. Core DVFS results.

Fig. 9. LLC partitioning results.

EDP improvement of 5.9% with 2.61% and 2.72% improvement in STP and Fairness, respectively. LLC1 (LLC2) exhibited an average system EDP improvement of 3.9% (4.76%) with 1.33% (2%) and 1.25% (2%) improvement in STP and Fairness, respectively. The DCP results obtained by our RL technique are compared with the Utility-based Cache Partitioning (UCP) (Qureshi and Patt 2006) and Memory-level Parallelism and Application’s Cache Friendliness for Efficient Management of Quasi-partitioning (MCFQ) (Kaseridis et al. 2014) techniques. Both the state-of-the-art techniques have been proposed for STP optimization (not EDP). Since it is non-trivial to construct an EDP improvement function from the cache misses, the proposals are not modified in our evaluations, and the implementation may be considered “UCP-based.” The LLC2 and LLC3 models outperform the state-of-the-art on the STP metric. The UCP (MCFQ) shows an average improvement of 4.6% (5.3%), 1.4% (1.9%), and 1.2% (1.7%) in EDP, STP, and Fairness, respectively. UCP and MCFQ require special hardware profilers for various data collection during runtime such as cache hits per way. MCFQ additionally requires the Miss Status and Handling Register (MSHR) queue length. LLC3 is able to outperform UCP and MCFQ on all three system metrics as shown in Figure 9.

13.1.7 MLM on Multithreaded Workloads. We have experimented with eight Splash2 benchmarks (*barnes*, *cholesky*, *fft_forever*, *fft_O0*, *fmm*, *lu.cont*, *raytracer_opt*, and *water.sp*) using four threads on a 4-core system. The JMLM co-optimization is able to outperform XCH by exhibiting 15.6% EDP improvement with 6.5% STP degradation. XCH results in a very high STP degradation of 19.5%. This can be mainly attributed to the analytical model used by XCH which does not account for the inter-thread data dependencies on performance.

13.2 Hardware Overheads for Co-optimization

13.2.1 MLM Overheads. A typical implementation of an RL-based controller would have the Q function as an $|S| \times |A|$ table, as shown in Figure 2, which results in a storage overhead. Additionally, to perform the Q -Table updates, the one-step Q -Learning equation is required to be computed. This computation would require two multiplications, three additions, and finding the maximum Q -Value of the new state of the system over all possible actions. The computation overhead would be within a small multiple of $|A|$. A hardware implementation of the controller would have a small overhead.

We could also implement the controller as an independent software thread with similar overheads. A software implementation when running with user applications can potentially degrade their performance by flushing out some useful data from the caches. A Q -Value update accesses a single row of the Q -Table to compute the maximum Q -Value and update the new

Table 5. Overhead Comparison for a 4-core System

Overhead	CMIR	XCH	MLM	coMLM	JMLM
Memory Storage (KB)	$1.3 + 4 \times 7.6 = 31.7$	$3.63 \times 4 = 14.5$	3.08	3.66	1.76
Search Time (cycles)	25,000	6,000	<100	<100	<2,000

value into the same row. Hence, the cache requirement for the controller thread would be only for $|A|$ table entries. Since all the proposed controllers have a very small action space, we expect negligible cache interference.

13.2.2 Memory Overhead and Search Time Comparison. Table 5 compares the hardware overhead for using CMIR, XCH-based co-optimization, Independent MLM (MLM), Cooperative MLM (coMLM), and Joint Action MLM (JMLM). CMIR proposed to use a total of 16 ANNs on a 4-core system, with each core running a 4-ANN ensemble. They proposed to multiplex a single ANN circuit with 52 multipliers, 1.3KB storage for edge weights, and 7.6KB memory per core for storing 300 data samples required for re-training the ANNs. They did not address the hardware overhead for implementing the stochastic hill climbing search. The XCH hardware overheads have been taken from Wang and Martínez (2015). The MLM co-optimization, corresponding to MLM, coMLM, and JMLM incurs only 3.08KB, 3.18KB, and 1.76KB of memory overhead required for storing the various Q-Tables. CMIR and XCH exhibit higher storage overheads of 31.7KB and 14.51KB, respectively.

Table 5 also compares the search time required by various co-optimization techniques. The MLM- and coMLM-based approaches have six different independent controllers executing which can multiplex the one-step *Q-Learning* hardware discussed in Section 13.2.1. The total reconfiguration search time would be the sum of computation cycles for all controllers. With five controllers having an action space of 3 and one with an action space of 19, the total cycles required for computing the *Q-Value* for all the controllers is equal to just 83 cycles. Since the MLM models perform searches by computing the maximum Q-Value action corresponding to the controller state, this results in a very small search time (<100 cycles) for MLM and coMLM. The JMLM requires a Hill Climbing search with a few iterations, resulting in a few thousand cycle search time. The state-of-the-art techniques require a much higher search time, as shown in Table 5, as reported in the corresponding research papers. coMLM and JMLM have lower hardware overheads than both the state-of-the-art techniques.

13.3 Cooperative MLM Techniques on 16-core System

The system with a higher core count can have a single monolithic LLC shared by all the cores or a physically sliced LLC with each LLC slice shared by a fixed number of cores.

13.3.1 System with Single Monolithic LLC. The proposed cooperative MLM techniques coMLM (Section 8) and JMLM (Section 10) are evaluated on a 16-core system with a configuration similar to the 4-core system shown in Table 2 with an 8MB, 64-way LLC shared by the 16 cores. The resource allocation search time is appropriately accounted for the 16 cores. The coMLM technique requires a modification to the RL-based DCP to ensure a small state space. The LLC is logically divided into multiple LLC slices of four sharing cores and executes an independent LLC-Ctrl on each logical LLC slice. A 16-core system results in four logical LLC slices.

Figure 10 shows the 10 workload mixes used to experiment on the 16-core system and compares the evaluation metrics for the coMLM, JMLM, and XCH techniques. The JMLM technique scales well to higher cores and outperforms the coMLM and XCH by exhibiting an average system EDP saving of 19.1% with only 2.6% penalty on the STP and Fairness. The coMLM technique exhibits

Fig. 10. coMLM, JMLM, and XCH evaluation on a 16-core system (monolithic LLC).

Fig. 11. coMLM evaluation on a 16-core system (sliced LLC).

higher EDP savings of 18.1% compared to 8.14% with XCH, but results in higher STP degradation of 6.4%.

13.3.2 System with Physically Sliced LLC. XCH assumes a single monolithic LLC for effective market creation. It is not clear from the proposal how it can handle an architecture with physically sliced LLC. We have only evaluated the coMLM proposal for this system. The 16-core system configuration is similar to the 4-core system shown in Table 2 and has 4 slices of LLC, each shared by four cores. Figure 11 shows the system metrics for five workloads used to experiment on a 16-core system using coMLM. The workloads have been created by mixing the 4-core workloads of Table 4 as shown in the table in Figure 10. WL1 corresponds to a 16-benchmark mix by combining the 4-benchmark WL numbers 1, 6, 11, and 16 from Table 4. The coMLM technique scales well to higher cores resulting in an average system EDP saving of 23.63%, with only a 1.94% and 1% penalty on the STP and Fairness, respectively.

Overall, the proposed MLM co-optimization performs much better than the CMIR and XCH co-optimization on all three evaluation metrics of system EDP, STP, and Fairness, with a lower hardware overhead.

14 CONCLUSION AND FUTURE WORK

We proposed and evaluated two cooperative multi-agent RL-based co-optimization techniques to jointly perform different performance and power optimizations involving cache partitioning and DVFS of core and uncore. We found that the multiple optimizations, when efficiently applied together, are able to outperform the individual optimization techniques. We were able to reduce the average system EDP by more than 20% (19%) with less than 5% (2.6%) penalty on the system throughput on a 4-core (16-core) system. The complexity of applying multiple optimizations on multi-core systems is effectively handled by the adaptiveness of the RL approach, with a very low overhead.

In the future, we plan to extend the techniques by studying the applicability of other learning approaches, the effect of other metrics such as Fairness, as well as sensitivity to interval size and the number of controller states and actions.

REFERENCES

- W. Lloyd Bircher and Lizy K. John. 2008. Analysis of dynamic power management on multi-core processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 327–338.
- W. Lloyd Bircher and Lizy Kurian John. 2011. Core-level activity prediction for multicore power management. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 1, 3 (2011), 218–227.
- Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 318–329.
- Jian Chen and Lizy Kurian John. 2011. Predictive coordination of multiple on-chip resources for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing*. ACM, 192–201.
- Jian Chen, Lizy Kurian John, and Dimitris Kaseridis. 2011. Modeling program resource demand using inherent program characteristics. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 1–12.
- Xi Chen, Zheng Xu, Hyungjun Kim, Paul V. Gratz, Jiang Hu, Michael Kishinevsky, Umit Ogras, and Raid Ayoub. 2013. Dynamic voltage and frequency scaling for shared resources in multicore processor designs. In *Proceedings of the 50th Annual Design Automation Conference*. ACM, 114.
- Zhuo Chen and Diana Marculescu. 2015. Distributed reinforcement learning for power limited many-core system performance optimization. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*. EDA Consortium, 1521–1526.
- Gaurav Dhiman and Tajana Simunic Rosing. 2007. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*. ACM, 207–212.
- AGBTG Dietterich. 2004. Reinforcement learning and its relationship to supervised learning. *Handbook of Learning and Approximate Dynamic Programming*, IEEE (2004), 47–60.
- Stijn Eyerman and Lieven Eeckhout. 2008. System-level performance metrics for multiprogram workloads. *IEEE Micro* 3 (2008), 42–53.
- Ann Gordon-Ross and Frank Vahid. 2007. A self-tuning configurable cache. In *Proceedings of the 44th Annual Design Automation Conference*. ACM, 234–237.
- Wim Heirman, Trevor Carlson, and Lieven Eeckhout. 2012. Sniper: Scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 91–94.
- Jörg Henkel, Vijaykrishnan Narayanan, Sri Parameswaran, and Jurgen Teich. 2013. Run-time adaption for highly-complex multi-core systems. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*. IEEE, 1–8.
- Intel. 2011. Intel Xeon Processor 5500 Series. *Datasheet, Volume 1* (2011), 136.
- Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture, 2008. ISCA'08*. IEEE, 39–50.
- Rahul Jain, Preeti Ranjan Panda, and Sreenivas Subramoney. 2016. Machine learned machines: Adaptive co-optimization of caches, cores, and on-chip network. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM.
- Ravindra Jejurikar and Rajesh Gupta. 2004. Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. In *Proceedings of the 2004 International Symposium on Low Power Electronics and Design*. ACM, 78–81.
- Da-Cheng Juan and Diana Marculescu. 2012. Power-aware performance increase via core/uncore reinforcement control for chip-multiprocessors. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, 97–102.
- Hwisung Jung and Massoud Pedram. 2010. Supervised learning based power management for multicore processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 29, 9 (2010), 1395–1408.
- Dimitris Kaseridis, Muhammad Faisal Iqbal, and Lizy Kurian John. 2014. Cache friendliness-aware management of shared last-level caches for high performance multi-core systems. *Computers, IEEE Transactions on* 63, 4 (2014), 874–887.
- Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez. 2014. Improving cache performance by exploiting read-write disparity. In *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*.
- Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO-2009*.
- Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, Vol. 40. ACM, 190–200.

- Kun Luo, Jayanth Gummaraju, and Manoj Franklin. 2001. Balancing throughput and fairness in SMT processors. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 164–171.
- Raman Manikantan, Kaushik Rajan, and Ramaswamy Govindarajan. 2012. Probabilistic shared cache management (PriSM). In *ACM SIGARCH Computer Architecture News*, Vol. 40. IEEE Computer Society, 428–439.
- Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N. Patt. 2012. Predicting performance impact of DVFS for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 155–165.
- Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. 2009. FlexDCP: A QoS framework for CMP architectures. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 86–96.
- Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, and Mateo Valero. 2008. MLP-aware dynamic cache partitioning. In *High Performance Embedded Architectures and Compilers*. Springer, 337–352.
- Janani Mukundan and Jose F. Martinez. 2012. MORSE: Multi-objective reconfigurable self-optimizing memory scheduler. In *Proceedings of the 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.
- Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large Intel® Itanium® programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 81–92.
- Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. 2003. Using SimPoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 31. ACM, 318–319.
- Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, run-time mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 423–432.
- Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 57–68.
- Tom Schaul, Justin Bayer, Daan Wierstra, Yi Sun, Martin Felder, Frank Sehnke, Thomas Rückstieß, and Jürgen Schmidhuber. 2010. PyBrain. *Journal of Machine Learning Research* 11 (2010), 743–746.
- André Seznec. 1993. A case for two-way skewed-associative caches. In *ACM SIGARCH Computer Architecture News*, Vol. 21. ACM, 169–178.
- Akbar Sharifi, Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. 2012. Courteous cache sharing: Being nice to others in capacity management. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 678–687.
- Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review* 36, 5 (2002), 45–57.
- Richard S. Sutton and Andrew G. Barto. 1998. *Reinforcement Learning: An Introduction*. Vol. 1. MIT Press, Cambridge.
- Ming Tan. 1993. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the Tenth International Conference on Machine Learning*. 330–337.
- Ying Tan and Qinru Qiu. 2008. A framework of stochastic power management using hidden Markov model. In *Proceedings of the Conference on Design, Automation and Test in Europe*. ACM, 92–97.
- Xiaodong Wang and José F. Martínez. 2015. XChange: A market-based approach to scalable dynamic multi-resource allocation in multicore architectures. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 113–125.
- Yanzhi Wang, Qing Xie, Ahmed Ammari, and Massoud Pedram. 2011. Deriving a near-optimal power management policy using model-free reinforcement learning and Bayesian classification. In *Proceedings of the 48th Design Automation Conference*. ACM, 41–46.
- Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3–4 (1992), 279–292.
- Stefan Wildermann, Michael Glaß, and Jürgen Teich. 2014. Multi-objective distributed run-time resource management for many-cores. In *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 221.
- Chenjie Yu and Peter Petrov. 2010. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *Proceedings of the 47th Design Automation Conference*. ACM, 132–137.
- Xiliang Zhong and Cheng-Zhong Xu. 2008. System-wide energy minimization for real-time tasks: Lower bound and approximation. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 3 (2008), 28.

Received November 2016; revised June 2017; accepted August 2017