Reinforcement Learning-driven Co-scheduling and Diverse Resource Assignments on NUMA Systems

Urvij Saroliya, Eishi Arima, Dai Liu, and Martin Schulz
Technical University of Munich, Garching, Germany
{urvij.saroliya, eishi.arima, dai.liu, martin.w.j.schulz}@tum.de

***Abstract*—As modern HPC systems are typically composed of fat and rich compute nodes, it is usually difficult to fully utilize all node resources with a single application. Co-scheduling, i.e., co-executing multiple complementary applications (or jobs) on the same node in a space sharing manner, is a promising solution and thus has been widely studied in the past decade. As one major drawback of co-scheduling is that it induces the interference effects among co-located applications due to contention among shared resources, the industry has started to support several resource/traffic partitioning features, e.g., in shared caches or memory controllers, on modern commercial processors. Recent studies proposed effective approaches to make use of these advanced features, however, the interactions between these features and (1) job scheduling decisions as well as (2) NUMA (Non-Uniform Memory Access) effects were generally overlooked. This paper explicitly targets these two missing pieces and comprehensively harmonizes the following decisions using reinforcement learning: (a) job selections for co-execution from a given job queue; and (b) diverse resource assignments to co-executed jobs, leveraging emerging hardware partitioning features, while taking NUMA-awareness into account. Our evaluation result demonstrates that our approach can improve the total system throughput by up to 78.1% over time sharing-based naive scheduling.**

***Index Terms*—NUMA Systems, Co-Scheduling, Resource Management, Reinforcement Learning**

I. Introduction

Ever since the end of Dennard scaling [1], the industry has continued to increase on-chip core counts. As a consequence, many-core processors have become common in HPC systems and optimizing the performance on such architectures has become essential for achieving performance at scale. In order to further enhance the node-level computational throughput and memory bandwidth, each compute node in those systems is typically composed of multiple processor sockets, which naturally makes them *NUMA* (Non-Uniform Memory Access) based designs [2]. Such a NUMA-based compute node is dividable into multiple NUMA domains, which are connected via an intra-node interconnect, and each NUMA domain consists of processor cores and dedicated memory resources. On such systems, localizing memory accesses by optimizing core/memory assignment to the running program is indispensable as accessing remote memories that belong to other domains typically incurs considerable overhead [3]–[12].

Meanwhile, as compute nodes in HPC systems are becoming fatter and richer, it is getting harder to fully utilize all node resources by a single application. On one hand, memory intensive applications typically need only a small fraction of on-node compute resources, while on the other hand, compute intensive applications can waste the plentiful bandwidth resources the node can offer.

One of the most promising solutions to mitigate the resource waste is *co-scheduling*, i.e., co-executing multiple applications (or jobs) simultaneously on the same node in a space sharing manner, which has been widely studied for servers and HPC systems [13]–[17]. By co-locating different types of applications that require complementary resources, resource waste can be significantly reduced. The major drawback is that it induces interference effects among co-executing applications due to contention for shared resources (e.g., shared caches and memory controllers). Therefore, industry has started to support new hardware features on recent commercial microprocessors that enable a partitioning of resources/traffic. Prominent examples are cache/bandwidth partitioning features (e.g., Intel's CAT/MBA [18]). Recent studies have shown the effectiveness of those partitioning features and proposed useful methodologies to optimize them [19]–[24]. However, the interactions between cache/bandwidth partitioning setups and (1) NUMA effects as well as (2) job scheduling decisions, i.e., selections of jobs from a job queue to co-locate them, were so far not covered by existing studies.

In this paper, we explicitly target *NUMA*-based systems that support such *emerging resource partitioning features*, and we comprehensively co-optimize (1) co-run job selections from a given job queue and (2) diverse resource assignments, including NUMA-aware core/memory assignments and shared resource partitioning based on the new hardware features. For this we apply a *reinforcement learning*-based holistic and systematic approach to harmonize those different types of hardware/software knobs. More specifically, we make the following major contributions:

1) We first demonstrate the correlation between core/memory mappings and cache/bandwidth partitioning setups.
2) We present the considerable impact of co-run job pair selections on performance and partitioning decisions.
3) Simultaneously targeting both the co-run job selections and the NUMA-aware resource assignments, we formulate the problem in a concrete mathematical form.
4) We provide a *reinforcement learning*-based holistic and systematic approach to solve the optimization problem.
5) We finally quantify the effectiveness of our approach, resulting in up to 78.1% system performance improvement.

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024

II. Background and Related Work

A. NUMA Systems and Optimizations

NUMA-based systems exist over the past few decades, and thus the data mapping as well as process/thread scheduling optimizations on them have been well studied. The IBM ACE multiprocessor workstation built in the 1980s was based on a NUMA architecture, and several page placement policies were explored for this particular machine [3]. Following this work, Bolosky et al. applied trace-based analyses and pointed that the optimal paging policy depends on the architectural parameters [4]. Li et al. then proposed a locality-based scheduling technique for parallel loops on NUMA [5]. Since the cache coherent NUMA (or CC-NUMA) architecture became dominating in the 1990s, Verghese et al. proposed an OS-assisted technique to improve the data locality for CC-NUMA [6]. After the end of Dennard scaling [1] in the mid 2000s, multi-/many-core architectures became dominant in the market, and the core count has been further pushed with using multiple sockets for high-end systems, which induced NUMA effects even within a node [8], [9]. Given this architectural trend, some studies targeted multi-threaded programs and attempted to minimize the thread-to-thread and thread-to-memory overheads for on-node NUMA optimizations [10], [11]. *Our work builds on this literature, and combines the NUMA-aware thread/data optimization with the emerging shared resource control features. It provides a holistic solution that includes co-scheduling job set selections using reinforcement learning.*

B. Co-scheduling and Resource Partitioning

Since multicore processors appeared in the market, various co-scheduling techniques have been proposed for servers and HPC systems. Bhadauria et al. explored the feasibility of co-scheduling and proposed a greedy-based scheduling policy [13]. Sasaki et al. proposed a scalability-based resource allocation approach for co-scheduled multi-threaded programs [14]. Breitbart et al. developed a resource monitoring tool for co-scheduling HPC applications [15] and provided a memory intensity-based co-scheduling policy [16]. Zhu et al. rather targeted CPU-GPU heterogeneous processors and proposed a co-scheduling approach suitable for them [17]. Saba et al. coordinated co-scheduling, resource partitioning, and power budgeting across CPUs and GPUs [25]. There exist several GPU-focused co-scheduling and resource partitioning studies [26], [27]. Álvarez et al. implemented a library to realize a system-wide co-scheduling for task-based applications on HPC systems [28]. Zou et al. explored the combination of co-scheduling and power capping for clusters [29]. *These co-scheduling studies focused on job selections and/or resource partitioning, however, did not target the combination of memory resource partitioning and NUMA effects.*

During co-scheduling, last level caches and underlying memory controllers are usually shared by multiple different cores on modern microprocessors. Hence, partitioning/isolating shared caches as well as main memory bandwidth traffics, while optimizing the assignments in accordance with the demands, is an effective approach. Qureshi et al. first proposed the concept of cache partitioning with an associated microarchitectural design and quantified the effectiveness via simulations [30]. Rafique et al. then devised a software/hardware mechanism to control memory bandwidth assignments among co-scheduled applications [31]. Driven by those pioneering studies, the industry has started supporting cache/bandwidth partitioning features in commercial processors [18]. Several recent studies explored the benefits of these new partitioning features and also proposed several useful techniques to optimize them [19]–[24]. Some studies focused only on the cache partitioning feature [19], [20], while others explored only the memory bandwidth partitioning [21], [22]. Park et al. first evaluated the combination of these two features [23], and then Chen et al. applied a machine learning approach to optimizing the setups of those hardware features [24]. Our work also covers the combination of those two knobs, *however ours newly introduces the following aspects in the optimization: (1) NUMA-aware resource assignments; and (2) job set selections to co-schedule from a given job queue.*

III. Observational Analysis

A. Target Systems

We target HPC systems that consist of multiple NUMA domains and support cache/bandwidth partitioning features controllable from software. As an example, Fig. 1 depicts the system we use in our evaluations. Note that the details of our evaluation environment are presented in §V. For the cache partitioning, we assume caches are partitioned and assigned to co-located programs at the granularity of cache ways. As for bandwidth partitioning, we assume the memory controllers have a function to isolate and prioritize memory access traffic from each co-running program by enabling and enforcing a limit of memory bandwidth utilization. These hardware features are widely supported in recent server-class commercial microprocessors and are controllable via software, such as the `rdtset` package [18]. On such a system, we co-schedule multi-threaded or multi-processed jobs/applications and, at the same time, decide the resource assignments, including cores, cache ways, and bandwidth, to the co-scheduled jobs when launching them.

B. Core/Memory Affinity Policies: Compact v.s. Distributed

We consider two different types of core mapping affinities for NUMA-based systems: *compact* and *distributed*. In the former, cores are preferably selected from the same NUMA domain, while in the latter, they are chosen from different NUMA domains in a round-robin manner. For both options, we consider different memory mapping policies, including *first touch* [3], [6], *round robin*, etc., so that memory accesses are optimized for the given co-run jobs and for the selected core mapping. In Fig. 1, the distributed policy is used for J_3, but the compact policy is applied to the others. The compact option is typically preferable when the NUMA interconnect becomes

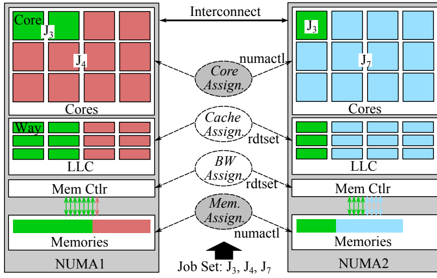

Fig. 1: An Example of Our Target Systems

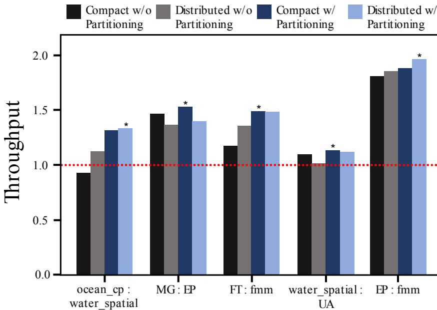

Fig. 2: Throughput Comparison among Different Core Affinity Policies

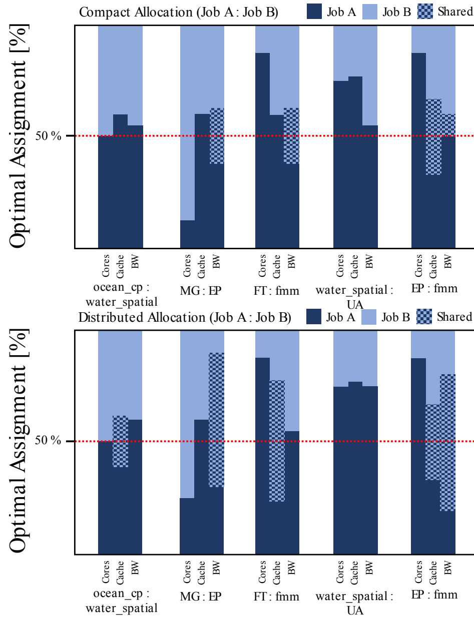

Fig. 3: Optimal Resource Allocations for Different Affinity Policies (Top: "Compact w/ Partitioning", Bottom: "Distributed w/ Partitioning")

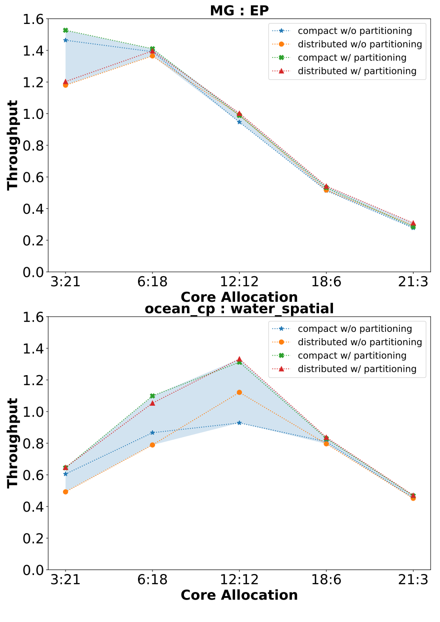

Fig. 4: Throughput v.s. Core Allocation for Different Co-run Pairs (Top: `MG:EP`, Bottom: `ocean_cp:water_spatial`)

the performance bottleneck[1]. This can be the case when a running job induces frequent inter-core communications or intensive remote memory accesses, e.g., due to irregular and sparse memory references. The distributed option, on the other hand, is suitable when the memory references are rather regular or localized within each NUMA domain so that more cache capacity and memory bandwidth is potentially available for the job (because these resources are also distributed). Once the set of co-scheduling jobs and the core/memory mappings are given, the cache and bandwidth are partitioned in order to mitigate the interference effects among co-located jobs on each NUMA domain.

C. Observations on a NUMA System

Fig. 2 compares throughput among different core affinity policies using different job pairs. The horizontal axis depicts different policies per job pair, while the vertical axis indicates relative throughput normalized to that of time-shared scheduling with using exclusive solo runs. We test both the compact and the distributed core affinity with and without using the cache/bandwidth partitioning features. Note, the best policy out of four is marked with "*" for each job mix. When the cache/bandwidth partitioning features are enabled, we select the best setup in a exhaustive manner, i.e., executing the given job pair repetitively while testing all the possible setups and picking the best one that maximizes throughput. The numbers of cores assigned to co-scheduled applications are also optimized in the exhaustive search procedure as well, and, at the same time, the memory mapping is selected from three different options (presented in §V) so that the co-run throughput is maximized. The exact search space with respect to the hardware assignments is presented in §V.

[1]NUMA interconnects usually offer less bandwidth than the total memory bandwidth aggregated across different NUMA domains. Further, they typically do not support any features to control quality of services, and thus affinity optimizations are the only way to mitigate their bottlenecks and contention.

As shown in the figure, the emerging cache/bandwidth partitioning features are very effective for several job mixes (e.g., `ocean_cp:water_spatial` and `FT:fmm`). This is because some of these programs are memory intensive or cache friendly, and thus isolating/partitioning these memory resources significantly mitigates the interference effects on them, which in turn improves the throughput considerably (when the resource assignments are configured accordingly). However, these new partitioning features are ineffective for other workloads, but rather the core affinity setup matters for them (e.g., `MG:EP`). One major reason is that the inter-NUMA communications can also be a bottleneck, for instance due to interference effects on the interconnect caused by irregular memory accesses. *Thus, depending on the selected jobs to be co-located, we need to carefully choose the affinity policy from the compact or distributed options. At the same time, we also need to carefully select the job pair as well — for instance, mixing `water_spatial` with `ocean_cp` outperforms doing so with `UA` as shown in the figure.*

Next, Fig. 3 illustrates the breakdown of resource allocations for different application pairs when they are optimized for *compact* or *distributed* core affinity options, top or bottom graph respectively. The X-axis lists different resources per job mix, whereas the Y-axis accumulates the resource allocation rates. For the cache and bandwidth partitioning, we apply also the *shared* option where resources can be used by both co-scheduled programs. *As shown in the figures, the core*

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024

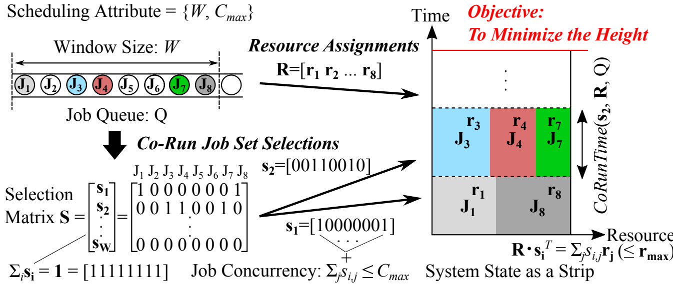

Fig. 5: Co-Scheduling and Resource Allocation as Strip Packing

affinity option (compact or distributed) can significantly affect the decisions on the resource partitioning setups including the core/cache/bandwidth partitioning. At the same time, the job pair selection considerably affects the decisions on these resource assignments as well, because optimal assignments highly depend on the given job mix.

Finally, Fig. 4 demonstrates throughput as a function of numbers of cores using the four different resource assignment options when co-executing `MG` and `EP` (top) / `ocean_cp` and `water_spatial` (bottom). In the figures, the horizontal axis indicates the numbers of cores assigned to the co-scheduled programs, while the vertical axis shows the relative throughput normalized to that of time-shared scheduling with exclusive solo runs. Here, when the cache/bandwidth partitioning features are enabled, the best partitioning setup is selected for the given core affinity policy based on the results of an exhaustive search. *As shown in the figures, the numbers of cores assigned to given jobs that are co-executed significantly affect the overall throughput, and, at the same time, the impact on throughput also can depend on the given job mix, the selected affinity policy (compact or distributed), and the cache/bandwidth partitioning setups.*

IV. Optimization via Reinforcement Learning

A. Problem Formulation

Fig. 5 depicts the scheduling and resource allocation problem we solve in this paper. We target the first W jobs in the job queue ($\mathrm{Q} = \{\mathbf{J_1}, \mathbf{J_2}, \cdots, \mathbf{J_W}\}$) for our co-scheduling and resource allocation optimization. Each $\mathbf{J_i}$ is a vector that lists parameters collected via profiling, in order to characterize the features of the ith job. We then introduce a $W \times W$ binary matrix ($\mathbf{S}$) that consists of W row vectors, each of which represents a set of jobs to co-locate — if an element ($s_{i,j}$) is set to 1, we launch the associated job (jth one) in the queue. Further, we utilize another matrix ($\mathbf{R}$) that aggregates a set of column vectors ($\mathbf{R} = [\mathbf{r_1}\mathbf{r_2}\cdots\mathbf{r_w}]$) to describe the resource assignments to the jobs. Here, $\mathbf{r_i}$ is corresponding to the ith job, and each element in the vector represents a certain resource (core, memory, cache way, or bandwidth) in one of the NUMA domains. All the parameters we use in this optimization are listed in Table I.

The optimization problem is formulated as follows:

$$
\begin{aligned}
&input && \mathrm{Q} = \{\mathbf{J_1}, \mathbf{J_2}, \cdots, \mathbf{J_W}\}, \quad W, \quad C_{max} \\
&output && \mathbf{S} = [\mathbf{s_1}^T \mathbf{s_2}^T \cdots \mathbf{s_w}^T]^T, \quad \mathbf{R} = [\mathbf{r_1}\mathbf{r_2}\cdots\mathbf{r_w}] \\
&min && \sum_{1 \le i \le W} CoRunTime(\mathbf{s_i}, \mathbf{R}, \mathrm{Q}) \\
&s.t. && s_{i,j} \in \{0,1\} \qquad (1 \le \forall i, \forall j \le W) \\
& && \mathbf{R} \cdot \mathbf{s_i}^T = \sum_{1 \le j \le W} s_{i,j}\mathbf{r_j} \le \mathbf{r_{max}} \quad (1 \le \forall i \le W) \\
& && \mathbf{s_i} \cdot \mathbf{1}^T = \sum_{1 \le j \le W} s_{i,j} \le C_{max} \qquad (1 \le \forall i \le W) \\
& && \mathbf{1} \cdot \mathbf{S} = \sum_{1 \le i \le W} \mathbf{s_i} = \mathbf{1} \qquad (\Rightarrow \mathbf{1} \cdot \mathbf{S} \cdot \mathbf{1}^T = W)
\end{aligned}
$$

The objective is to minimize the total co-execution run time, which is equivalent to maximizing system throughput. The first constraint restricts the elements of the job selection matrix ($\mathbf{S}$) to 0 (job not selected) or 1 (job selected). The second constraint is the resource constraint, i.e., the sum of the resource allocation vectors across the ith set of co-located jobs ($s_{i,1}\mathbf{r_1}$, $s_{i,2}\mathbf{r_2}$, ...) must be less than or equal to the limits denoted as $\mathbf{r_{max}}$. The third constraint keeps the job concurrency within C_{max}, i.e., the number of selected job (i.e, 1s in the vector $\mathbf{s_i}$) must be less than or equal to C_{max}. The last constraint states that the W jobs in the queue are scheduled in a mutually exclusive and collectively exhaustive fashion, i.e., they are launched only once and thus in total W different jobs are scheduled. Further, to limit the number of variants of $\mathbf{S}$, the vectors $\mathbf{s_1}$, $\mathbf{s_2}$, ... are sorted by the co-run effectiveness (speedup over solo executions).

This optimization can be considered a variant of the well-known strip packing problem [32]. In a strip packing problem, multiple items (usually with rigid rectangular shapes) and a strip are given, and the objective is to minimize the height after filling the strip using all given items. In our problem, the shapes of items can be changed depending on the resource assignment setups ($\mathbf{R}$), and also each item has multiple dimensions. As the basic strip packing problem is known as NP-hard [32], ours falls at least in the same category, given that ours even has a higher degree of freedom in its decisions.

TABLE I: Definitions of Symbols/Functions

Symbol	Remarks
Q	The set of queuing jobs in the window: $\mathrm{Q} = \{\mathbf{J_1}, \cdots, \mathbf{J_W}\}$
$\mathbf{J_i}$	The vector to list job feature parameters for ith job in the queue
W	The window size of the queue
C_{max}	The maximum number of concurrently running jobs
$\mathbf{S}$	The matrix of co-scheduling decisions: $\mathbf{S} = [\mathbf{s_1}^T \cdots \mathbf{s_w}^T]^T$
$\mathbf{s_i}$	The row vector to state ith set of co-locating jobs ($s_{i,j} = 0 or 1$)
$\mathbf{R}$	The matrix of resource assignment decisions: $\mathbf{R} = [\mathbf{r_1} \cdots \mathbf{r_w}]$
$\mathbf{r_i}$	The resource allocation setup for ith job — each element represents one of the resources in a certain NUMA domain
$\mathbf{r_{max}}$	The resource constraint vector — each element represents the limit of one of the resources in a certain NUMA domain
$\mathbf{1}$	The row vector whose elements are all 1: $\mathbf{1} = [1 \cdots 1]$

Function	Remarks
$CoRunTime(\mathbf{s_i}, \mathbf{R}, \mathrm{Q})$	The total execution time when co-locating jobs selected from Q and assigning resources based on $\mathbf{s_i}$ and $\mathbf{R}$

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024

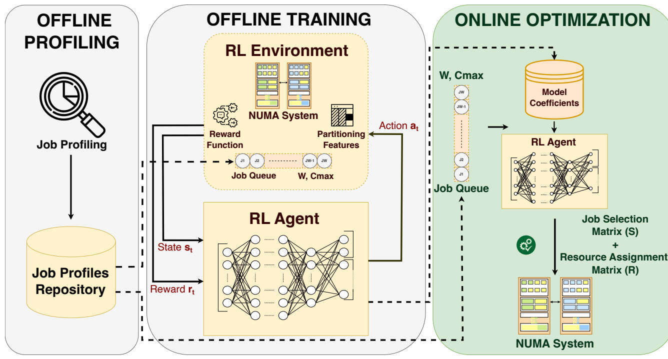

Fig. 6: Our Reinforcement Learning-Based Solution

B. High-level Procedures

Fig. 6 presents the entire system architecture of our solution. As depicted in the diagram, it comprises three distinct components: (1) offline profiling, which involves the collection of application profiles; (2) offline training, aimed at identifying the coefficients of our agent; and (3) online optimization, where the trained agent is deployed for decision-making purposes.

To characterize jobs, we utilize a profile-based approach, and thus we have an offline profiling as a component in our solution. To perform application profiling, we utilize hardware performance counters, and the exact counters used in our evaluation are listed in Table III in §V. These profiles must be collected in advance for all co-scheduling targets in both the offline and online phases. We collect solo-run profiles for all benchmark programs used in the offline training phase. We conduct the solo-run profiling also for queuing jobs in the online optimization phase if their profiles are not available. More specifically, if the associated application profile is not available for a job, it is excluded from the co-scheduling targets, and such a job is executed with exclusively using node resources while collecting the profile. If an application is previously executed on the system and a profile is available, it is included as a co-scheduling target[2].

For the offline model training, we create a set of benchmark program mixes to co-schedule on the target NUMA system in order to train our agent. For each program mix, we continuously probe the co-run throughput while changing the resource allocations, including core/memory assignments and cache/bandwidth partitioning. This procedure is conducted based on *reinforcement learning*, i.e., we update the mapping policies and resource allocations accordingly in the next co-run based on the reward function output. During the procedure, the state-action table for our agent is trained, and, as we approximate the table with a neural network in this work, the coefficients of the neural network are eventually identified.

In the online phase, we deploy the trained agent mentioned above to solve our optimization problem. The agent regards the optimization as a classification problem and uses the model to choose an optimal set of two matrices: co-scheduling job mixes ($\mathbf{S}$) and resource allocations ($\mathbf{R}$). We use this machine learning approach as the scheduling and resource optimization problem is strongly hard as mentioned in the last section, and we utilize reinforcement learning-based offline training rather than well-known offline supervised learning because the labeling is infeasible for our purposes. More specifically, labeling here associates a given set of jobs (Q) with the best set of co-scheduling decisions ($\mathbf{S}$) and resource allocations ($\mathbf{R}$), which takes tremendous amount of time with exhaustive searches, rendering any possible run time advantage moot.

[2]Input can highly influence the application behavior, however this effect is ignored by using the same inputs in this study. Future extensions are possible by extending existing studies [33], [34].

C. Mapping Reinforcement Learning to Offline Training

In reinforcement learning, an agent ought to take actions while learning by interacting with the environment in order to maximize the cumulative reward [35]. By continuously performing such trial-and-error cycles, the agent eventually reaches the goal state and returns optimal answers to our optimization problem. To this end, we need to carefully define entities/properties of reinforcement learning for our problem, and the followings are the definitions in this approach.

1) Agent: The agent takes the job queue (Q) as an input (or a state), and returns a set of two output matrices as an action: the sets of jobs to be co-scheduled ($\mathbf{S}$) and their corresponding resource allocations ($\mathbf{R}$). It learns an optimal policy to govern the actions so as to maximize the accumulation of the reward signals in the offline training in our approach.

2) Environment: Environment is the agent's context in which it operates and interacts. In this work, the environment consists of the target NUMA system and its hardware features, including cache/bandwidth partitioning options.

3) State: The state is defined as a set of information that an agent has about the environment at any given time, encompassing what is required for deciding the actions. In our approach, the state contains information about all of jobs in the current window ($\mathrm{Q} = \mathbf{J_1}, \mathbf{J_2}, \cdots, \mathbf{J_W}$) characterized by their profiles, job selections, and resource allocations.

4) Action: An action is a state-transitioning operation based on the current system state. In our approach, an action involves determining matrices: co-scheduling job selections ($\mathbf{S}$) and their corresponding resource allocations ($\mathbf{R}$).

5) Reward: The effectiveness of the reinforcement learning approach is ultimately determined by the definition of the reward signal. For every action, the agent receives the reward signal as a numerical value, which quantifies and evaluates an action at a given state of the system.

D. Obtaining State-Action Relationship via Soft Actor-Critic

Reinforcement Learning is used to establish a state-action relationship driven by a provided reward signal. At a given time-step t, the state $st_t \in \{\mathbf{Q}\}$ and action $a_t \in \{(\mathbf{S}, \mathbf{R})\}$ define the current system state and action taken, respectively. The goal is to learn a strategy that maximizes the anticipated cumulative reward upon executing action a_t in state st_t. This learning process is steered by the reward signal rw_t.

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024

We opt for the Soft Actor-Critic (SAC) method [36] for offline training. SAC, being an off-policy algorithm, simultaneously maximizes both cumulative reward and entropy. The off-policy nature allows the agent to learn from diverse policies, thereby enhancing sample efficiency. Co-optimizing for cumulative rewards and entropy empowers the agent to learn the most optimal policies while effectively exploring the environment. SAC was originally designed for continuous action spaces; nevertheless, its effectiveness has led to its adaptation for discrete action settings as well [37]. We have incorporated the SAC developed for discrete action settings by Christodoulou [37] into our approach.

To guide the agent effectively, we implement a two-level reward function comprising an *intermediate* reward and a *final* reward. The intermediate reward (rw_i) serves as a signal during the formulation of the scheduling policy, while the final reward (rw_f) is utilized for evaluating the effectiveness of the scheduling policy (see also Table VI in §V).

V. Evaluation Setup

A. Evaluation Environment

Table II lists the specifications of our platform which is composed of two processors that support the emerging cache/bandwidth partitioning features, i.e., Intel CAT/MBA, controllable via the `rdtset` command [18]. In our evaluation, the cache partitioning feature, i.e., Intel CAT, is applied only to the last level caches. Although the accuracy and effectiveness of these partitioning features depend on the CPU generation as reported in a recent study by Sohal et al. [38], our agent learns their performance impact throughout the training procedure. The core/memory mapping affinities are controlled by using the `numactl` command [2].

Our approach is implemented with Python using multiple standard libraries as follows. We build our reinforcement learning environment using the `gymnasium` python library [39]. For implementing the agent, we use the `PyTorch` library [40] for implementing the deep neural networks for Soft Actor-Critic Method [37]. Further, we use `scikit-learn` for performing additional data pre-processing and feature engineering [41].

We collect hardware performance counters listed in Table III using the `Linux Perf` command [42] to profile and characterize the applications. These statistics enable us to assess the applications' characteristics in terms of compute intensity, memory intensity, cache friendliness, and so forth, which are indispensable for our approach. Furthermore, these statistics have been selected based on the initial feature engineering, where we analyzed the correlation between various application features and co-scheduling throughput.

TABLE II: Evaluation Platform

Name	Remarks
CPU	Intel(R) Xeon(R) Silver 4116 x2 sockets
OS	Ubuntu 20.04.4 LTS, Kernel Version: 5.13.0-22-generic
Software	Gcc/Gfortran Version: 9.4.0, Numactl Version: 2.0.12-1, Rdtset Version: 3.2.0, Python Version: 3.10.12

B. Exploration Space

Here, we present the universe of hardware configurations we use in our evaluation. The agent in our approach explores configurations in this space and picks an optimal combination from them. As described in §III, we choose the core mapping policy from two options: *compact* or *distributed*. As for the memory mapping policy, we have three options: *first-touch*, *round-robin*, and *local-alloc* (designated by the `numactl` command). For the core counts, we choose from the following options: 3, 6, 12, 18, and 21 (out of 24 cores). For the cache allocation, we have five options: 4, 8, 12, 16, and 22 (out of 22 ways on 2 sockets). For the memory bandwidth allocation, we set from the following five options: 20%, 40%, 60%, 80%, and 100%. During the allocation of cache and memory bandwidth, we distribute the selected cache ways and memory bandwidth either in a *balanced* manner or *proportionally* (based on the core allocation ratio) to each socket. In addition to these actions, a *skip* option is available to move the current job to the end of the job window, facilitating the job selection process. Hence, the exploration space comprises 257 unique actions [(*schedule-level:* 2×3) + (*job-level:* $5 \times 5 \times 5 \times 2$) + *skip*].

C. Workloads

We utilize the common HPC benchmarks: `Parsec` benchmark suite [43] and the `NAS Parallel` benchmark suite [44] as listed in Table IV, as they have been widely used in other multi-/many-core processor studies. For the offline training, we exclude 7 programs marked with * in the table and use the remaining 12 programs. The objective of the exclusion procedure is to check if our approach can generalize to unseen applications.

In our evaluation, we first fix the job window size (W) to 6. We later scale the size as well to assess the impact of the window size selection. In this work, we fix the maximum

TABLE III: Collected Hardware Performance Counters

Statistics
duration_time, task-clock, context-switches, cpu-cycles, instructions, page-faults, branch-misses, L1-dcache-load-misses, L1-icacheload-misses, LLC-load-misses, dTLB-load-misses, iTLB-loadmisses

TABLE IV: Evaluation Benchmarks

Suite	Benchmark Applications
Parsec	barnes*, cholesky*, fft, fmm, lu_cb, lu_ncb, ocean_cp, ocean_ncp, raytrace*, water_nsquared*, water_spatial
NAS Parallel	BT*, CG, DC*, EP, FT, IS*, MG, SP

TABLE V: Tested Job Mixes ($W = 6$)

Name:{ Jobs ... }
Q1:{fmm, FT, EP, fft, CG, lu_ncb}, **Q2**:{IS*, lu_ncb, EP, CG, SP, cholesky*}, **Q3**:{fft, MG, IS*, lu_ncb, lu_cb, EP}, **Q4**:{fmm, IS*, MG, FT, SP, fft}, **Q5**:{CG, MG, FT, fmm, DC*, water_nsquared*}, **Q6**:{lu_ncb, water_spatial, fft, DC*, fmm, MG}, **Q7**:{ocean_ncp, CG, fmm, barnes*, ocean_cp, lu_ncb}, **Q8**:{IS*, MG, FT, CG, DC*, ocean_ncp}, **Q9**:{MG, EP, IS*, fft, DC*, water_nsquared*}, **Q10**:{DC*, barnes*, cholesky*, lu_ncb, CG, fmm}, **Q11**:{lu_cb, MG, barnes*, cholesky*, fmm, MG}, **Q12**:{raytrace*, cholesky*, DC*, BT*, fft, lu_ncb}

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024

concurrency (C_{max}) to 4, i.e., at most 4 jobs can be co-located at the same time. Note that the concurrency can be less than 4 depending on the decision made by the agent.

We create 16 different job queues for the agent training, each of which consists of W programs randomly selected from the 12 programs. As for the online inference, we test our approach by randomly creating 12 different job queues by sampling the jobs from all 19 available jobs. The exact job mix selections for $W = 6$ are listed in Table V. Note, the programs marked with * are unseen in the training.

D. Setup for Training and Inference

Table VI lists the setup used for the reward function and the agent. As mentioned in §IV, we use two kinds of rewards in this evaluation: (i) intermediate reward rw_i and (ii) final reward rw_f. On one hand, the intermediate reward evaluates the resource allocation for a selected job, which can be assessed before launching the job using the associated profile. It returns a higher reward when: (i) allocating more cores to a job with higher compute scale factor, (ii) allocating more cache/bandwidth to a job with higher L3 cache misses, in other words to a job which might need to access main memory more often. On the other hand, the final reward refers to the measured throughput improvement over the time-sharing executions, which is obtained only after the completion of co-running a job mix. Most importantly, the intermediate reward rw_i is heuristics-based and is only used for an approximate idea, whereas the learning is mainly driven by the final reward rw_f. *Note, the reward function can be customized for other factors such as maximizing energy efficiency, minimizing application slowdown, and achieving fairness by modifying the definition specified in the table.*

In the table, $CoreAllocRatio$, $CacheAllocRatio$ and $BWAllocRatio$ are (i) the ratio of allocated cores to the total number of available cores, (ii) the ratio of allocated cache ways to the total number of available cache ways, and (iii) the ratio of allocated memory bandwidth to the total available memory bandwidth, respectively. $ScaleFactorRatio$, $DurationRatio$ and $L3CacheMissesRatio$ are defined as follows: (i) $ScaleFactorRatio$ is the ratio of scale factor of the current job to the mean scale factor of the job window. Here, scale factor is the ratio of running the job on single core to running the job on all available cores. (ii) $DurationRatio$ is the ratio of solo-run execution time of the current job to the mean solo-run execution time of the job window.

TABLE VI: Agent and Reward Function Setups

Type	Setups
Reward Function	$rw_i = CoreAllocRatio * (ScaleFactorRatio^2 + DurationRatio^2) + (CacheAllocRatio + BWAllocRatio) * L3CacheMissesRatio^2$ $rw_f = (SoloRunTime/CoRunTime - 1) \times 100$
Agent	[**# of neurons in the input layer**]: $W \times (f + 9)$, [**# of hidden layers for each Network** 3, [**# of neurons in each hidden layer**]: 256/256/256, [**# of neurons in the output layer**]: 257, [**Layer NW**]: Fully connected, [**Activation function**]: Rectified Linear, Softmax

(iii) $L3CacheMissesRatio$ is the ratio of L3 cache misses of the current job to the mean L3 cache misses of the job window.

As described earlier, the agent is configured with the soft actor-critic method for discrete settings [37], and the details are listed also in Table VI, where W is the window size and f is the count of job performance counters used as job features. In this method, we need three separate neural networks for function approximations: 1 actor network and 2 critic networks. Specific details about the update rules for each of the networks can be seen in the work by Christodoulou [37].

The mentioned procedure takes a state vector as input and provides an action to be taken. At the first time step of an episode, the schedule-level policies are defined. Next, each time-step of the training episode determines all of the required actions for resource allocation for a particular job. This procedure is meant to converge to the global optimal as far as possible. After the training procedure is completed, we perform tests in the evaluation mode for the trained models.

VI. Experimental Results

A. Throughput Comparison among Different Methods

We compare our approach ($C_{max} : 4$) against several scheduling and resource allocation methodologies. For "Time Sharing (Baseline)", jobs in the given queue are executed with full system resources without co-scheduling. "OCRP-ML" utilizes a state-of-the-art CPU/GPU co-scheduling and resource partitioning framework proposed by Saba et al. [25], which works only with $C_{max} = 2$. "OCRP-ML - BEST" assesses the theoretical maximum throughput of their work ($C_{max} : 2$) — an exhaustive search is employed to identify the optimal combination of job-sets and resource allocations. To emphasize the significance of job selection, we assess our reinforcement learning approach with the FIFO order scheduling in "RL - FIFO" ($C_{max} : 4$).

Fig. 7 shows the comparison of throughput for the mentioned scheduling and resource allocation methods. The X-axis represents job queues used for evaluations (AM: Arithmetic Mean) as indicated in Table V, while the Y-axis represents the relative throughput normalized to that of *Time Sharing* for each job queue. We set $W = 6$ in this comparison.

Overall, our approach outperforms all other scheduling methods. As compared to the "Time Sharing", our approach achieves a factor of 1.303 throughput improvement. Additionally, we observe a maximum throughput improvement of up to x1.781. In contrast to the "OCRP-ML" methods, where cache/memory bandwidth partitioning is not available and concurrency is limited, our approach holds a distinct advantage. Lastly, emphasizing the critical role of job-set selection quality in co-scheduling, the use of the "RL - FIFO" scheduling policy limits the agent from achieving better performance.

B. Verification of Core/Memory Affinity Selection

Here, we evaluate whether the selected job sets are allocated using the optimal core/memory affinities (or NUMA policies) for each job queue. Fig. 8 demonstrates the verification by

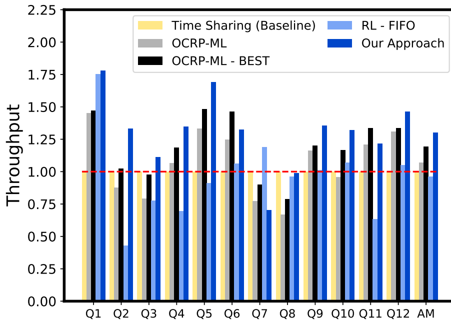

Fig. 7: Throughput Comparison among Different Methods ($W = 6$)

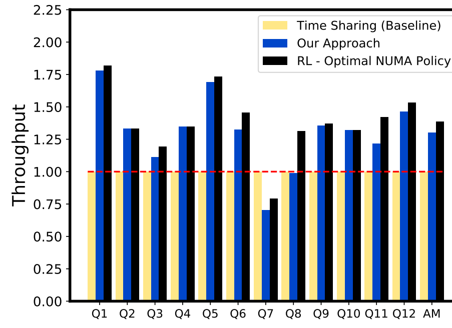

Fig. 8: Throughput Comparison against the Optimal Affinity Selection ($W = 6$)

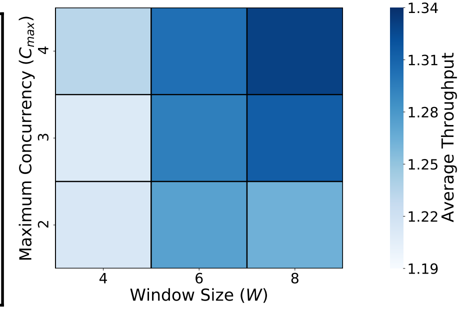

Fig. 9: Average Throughput Comparison for Varying Values of Window Size (W) and Maximum Concurrency (C_{max})

comparing throughput among different methods for each job queue. The X-axis represents the various job queues as referred from the Table V, and the Y-axis represents the relative throughput normalized to ”Time Sharing”. ”RL - Optimal NUMA Policy” method refers to running the jobs selected by the RL agent along with the corresponding resource assignments with the optimal core/memory affinity (or NUMA policy), i.e., cores and memory mapping policy (e.g., compact with first touch) are selected based on the exhaustive search.

Overall, our approach achieves near-optimal throughput for almost all cases, and the average throughput degradation (or room for improvement) compared to ”RL - Optimal NUMA Policy” is only 6.38%. Note, we can achieve even further throughput improvement by fixing the reward function accordingly. In the current implementation, an intermediate reward is designed in a job-wise manner. However, a global view that considers all co-located jobs would be a help for the core/memory mapping affinity selections. This is simply because the possible affinity selections for a job are restricted by those of the other co-located jobs.

C. Scaling Scheduling Attributes

In the experiments above, we fixed the window size (W) at 6 and the maximum concurrency (C_{max}) at 4. In this section, we scale both of these scheduling attributes, with W to values in the range $[4, 6, 8]$ and C_{max} in the range $[2, 3, 4]$. Fig. 9 depicts the comparison of average throughput for different values of W and C_{max}. The x-axis indicates the window size (W), and the y-axis shows the maximum concurrency (C_{max}). The color coding represents the average throughput for each combination of the $W - C_{max}$ setup. The evaluations have been done using the same set of jobs as mentioned in Table V. As we have in-total 72 jobs (6×12) in this table, we create similar job queues using $W = 4$ and $W = 8$. We observe that our approach scales effectively with the scheduling attributes, as the average throughput shows a gradual improvement with the scaled values of W and C_{max}.

D. Time and Memory Overheads

Finally, we report the time and memory overheads. The time overhead of online inference compared with each job execution time is *only 1.06%* on average across our workloads. The model training time is in the range of roughly 10-12 hours, which however is needed *only once per system*. During inference, the memory overhead of our RL agent is *only 13MiB* which is negligible compared with the node memory capacity in modern systems.

VII. Discussion

Our reinforcement learning-based approach is generic and naturally applicable to other kinds of HPC systems including CPU-GPU heterogeneous architectures and hybrid memory-based systems equipped with emerging memory technologies. In general, these systems have different types of QoS control knobs, and in principle, our approach is extensible to optimize the setups with minor modifications in our agent, reward function, and benchmark set. Further, optimizing the combination of NUMA affinity and resource partitioning setups would continue to be significant in HPC throughout the future given the following architectural trends: (1) HPC nodes are becoming fatter and fatter; and (2) HPC processors are getting larger and larger by integrating multiple chiplets, which is inducing NUMA effects even inside a package.

We plan to extend our work to a cluster scale. This extension opens up new fundamental challenges such as: synchronizing resource partitioning setups across nodes to balance the node performance for multi-node jobs; and decision making on inter-node core assignment, i.e., *compact* or *distributed* across nodes. Nevertheless, the contributions presented here still apply and form the necessary foundation. The extension will come with an integration of our solution here with an existing cluster management tool, such as the `Slurm` workload manager [45]. The integration requires such as (1) porting our solution to the `Slurm` framework as a scheduling plugin and (2) realizing the resource partitioning functionalities inside of `Slurm` via such as custom prolog/epilog scripts that internally handle the `Rdtset` commandline interface or calling the `PQoS` library inside the plugin.

As job selections and core resource/affinity assignments are typically static decisions in HPC, we focus on the profile-driven static management. For some dynamically controllable knobs/resources (e.g., Intel CAT and MBA), runtime control can be promising, in particular when the applications change

Author’s Preprint: To Appear in IEEE International Conference on Computer Design 2024

their behaviors dynamically. This will be a promising extension for our work by updating our agent or combining our work with an existing tool (e.g., DRLPart [24]).

VIII. Conclusion

We targeted co-scheduling and resource partitioning on modern NUMA systems with emerging cache and bandwidth partitioning features. We first explored the correlations between NUMA-aware core/memory assignments and these new hardware partitioning features, and also demonstrated the impacts on job pair selections. Based on the observations, we formulated the job co-scheduling and resource assignment problem in a concrete mathematical form, and we then proposed a reinforcement learning-based systematic approach to solve the optimization problem. Our proposed technique achieved significant throughput improvement up to 78.1%.

Acknowledgement

This work has received funding from the REGALE project from EuroHPC JU under grant agreement no. 956560 and the German Federal Ministry of Education and Research (BMBF) under grant number 16HPC039K. Additionally, it received funding from the Plasma PEPSC project under EuroHPC JU (grant agreement no. 101093261) and the BMBF (grant no. 16HPC075). Further, it was supported by BMBF through the initiative SCALEXA and the PDExa project (16ME0641).

References

[1] R. H. Dennard, F. H. Gaensslen, H.-N. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[2] A. Kleen, "A numa api for linux," *Novel Inc*, 2005.

[3] W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for numa memory management," in *SOSP*, 1989, pp. 19–31.

[4] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, and A. L. Cox, "Numa policies and their relation to memory architecture," in *ASPLOS*, 1991, p. 212–221.

[5] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and loop scheduling on numa multiprocessors," in *ICPP*, 1993, pp. 140–147.

[6] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum, "Operating system support for improving data locality on cc-numa compute servers," in *ASPLOS*, 1996, p. 279–289.

[7] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending openmp for numa machines," in *SC*, 2000, pp. 48–48.

[8] C. McCurdy and J. Vetter, "Memphis: Finding and fixing numa-related performance problems on multi-core platforms," in *ISPASS*, 2010, pp. 87–96.

[9] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for numa-aware contention management on multicore systems," in *PACT*, 2010, pp. 557–558.

[10] S. Imamura, H. Sasaki, K. Inoue, and D. S. Nikolopoulos, "Power-capped dvfs and thread allocation with ann models on modern numa systems," in *ICCD*, 2014, pp. 324–331.

[11] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on numa systems: Asymmetry matters," in *USENIX ATC*, 2015, p. 277–289.

[12] I. Sánchez Barrera *et al.*, "Modeling and optimizing numa effects and prefetching with machine learning," in *ICS*, 2020.

[13] M. Bhadauria and S. A. McKee, "An approach to resource-aware co-scheduling for cmps," in *ICS*, 2010, pp. 189–199.

[14] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura, "Scalability-based manycore partitioning," in *PACT*, 2012, pp. 107–116.

[15] J. Breitbart *et al.*, "Case study on co-scheduling for hpc applications," in *ICPP Workshops*, 2015, pp. 277–285.

[16] ——, "Dynamic co-scheduling driven by main memory bandwidth utilization," in *CLUSTER*, 2017, pp. 400–409.

[17] Q. Zhu, B. Wu, X. Shen, L. Shen, and Z. Wang, "Co-run scheduling with power cap on integrated cpu-gpu systems," in *IPDPS*, 2017, pp. 967–977.

[18] Intel, "Intel(r) rdt software package," https://github.com/intel/intel-cmt-cat, 2021, accessed: Nov 30, 2023.

[19] G. Aupy, A. Benoit, B. Goglin, L. Pottier, and Y. Robert, "Co-scheduling hpc workloads on cache-partitioned cmp platforms," in *CLUSTER*, 2018, pp. 348–358.

[20] K. Nikas *et al.*, "Dicer: Diligent cache partitioning for efficient workload consolidation," in *ICPP*, 2019.

[21] J. Park, S. Park, M. Han, J. Hyun, and W. Baek, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers," in *PACT*, 2018.

[22] Y. Xiang, C. Ye, X. Wang, Y. Luo, and Z. Wang, "Emba: Efficient memory bandwidth allocation to improve performance on intel commodity processor," in *ICPP*, 2019.

[23] J. Park, S. Park, and W. Baek, "Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *EuroSys*, 2019.

[24] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "Drlpart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *HPDC*, 2021, p. 175–188.

[25] I. Saba, E. Arima, D. Liu, and M. Schulz, "Orchestrated co-scheduling, resource partitioning, and power capping on cpu-gpu heterogeneous systems via machine learning," in *ARCS*, 2022, p. 51–67.

[26] E. Arima, M. Kang, I. Saba, J. Weidendorfer, C. Trinitis, and M. Schulz, "Optimizing hardware resource partitioning and job allocations on modern gpus under power caps," in *ICPPW*, 2022.

[27] U. Saroliya, E. Arima, D. Liu, and M. Schulz, "Hierarchical resource partitioning on modern gpus: A reinforcement learning approach," in *CLUSTER*, 2023, pp. 185–196.

[28] D. Álvarez, K. Sala *et al.*, "nos-v: Co-executing hpc applications using system-wide task scheduling," *arXiv preprint arXiv:2204.10768*, 2022.

[29] P. Zou *et al.*, "Contention aware workload and resource co-scheduling on power-bounded systems," in *NAS*, 2019, pp. 1–8.

[30] 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, pp. 423–432.

[31] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *PACT*, 2007, pp. 245–258.

[32] S. Martello *et al.*, "An exact approach to the strip-packing problem," *INFORMS Journal on Computing*, vol. 15, no. 3, pp. 310–319, 2003.

[33] E. Ipek *et al.*, "An approach to performance prediction for parallel applications," in *Euro-Par*, 2005, pp. 196–205.

[34] A. Calotoiu *et al.*, "Fast multi-parameter performance modeling," in *CLUSTER*, 2016, pp. 172–181.

[35] R. S. Sutton, *Reinforcement learning: An introduction*. MIT press, 2018.

[36] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *ICML*, 2018, pp. 1861–1870.

[37] P. Christodoulou, "Soft actor-critic for discrete action settings," *arXiv preprint arXiv:1910.07207*, 2019.

[38] P. Sohal *et al.*, "A closer look at intel resource director technology (rdt)," in *RTNS*, 2022, pp. 127–139.

[39] F. Foundation, "Gymnasium documentation," https://gymnasium.farama.org/, 2022, accessed: Nov 30, 2023.

[40] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *NIPS*, 2019, pp. 8024–8035.

[41] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[42] "Perf wiki," https://perf.wiki.kernel.org/, accessed: Nov 30, 2023.

[43] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008, p. 72–81.

[44] D. H. Bailey *et al.*, "The nas parallel benchmarks—summary and preliminary results," in *Supercomputing*, 1991, p. 158–165.

[45] "Slurm workload manager," https://slurm.schedmd.com/, accessed: Nov 30, 2023.

Author's Preprint: To Appear in IEEE International Conference on Computer Design 2024