CCF Transactions on High Performance Computing (2022) 4:394–406
https://doi.org/10.1007/s42514-022-00110-2

REGULAR PAPER

Smart scheduler: an adaptive NVM-aware thread scheduling approach on NUMA systems

Yuetao Chen[1,4] · Keni Qiu[2] · Li Chen[1] · Haipeng Jia[1] · Yunquan Zhang[1] · Limin Xiao[3] · Lei Liu[3]

Received: 27 February 2022 / Accepted: 26 May 2022 / Published online: 11 October 2022

Abstract
NVM provides large memory capacity, long-term data durability, and high memory bandwidth for multi-thread applications on cloud servers. Nowadays, cloud servers often employ NUMA architecture, where the thread scheduling mechanism plays a vital role in overall system performance because of the NUMA property. However, with the increase in server resources' diversity, i.e., hybrid memory systems using DRAM and NVM on NUMA nodes, the exploration space for thread scheduling is expanding rapidly. Unfortunately, the existing thread schedulers, including rule-based algorithms and scheduling domain methods, cannot provide ideal scheduling solutions in such complicated cases. And, those thread schedulers neglect customized heterogeneous memory structures, thus degrading overall system performance. Fortunately, reinforcement learning can choose actions with maximum rewards values in a specific environment, leading the scheduler towards an optimal solution. In this paper, we propose a thread scheduling approach, i.e., Smart Scheduler, by leveraging a reinforcement learning method. Smart Scheduler takes OS event information as input, extends LinUCB to explore the scheduling space, and guides thread-level scheduling. We evaluate Smart Scheduler on the off-the-shelf server equipped with NVM. The experimental results show that the proposed Smart Scheduler can converge faster (usually within 20 actions) than rule-based algorithms and scheduling domain methods and reduce program execution time by up to 59.9%. It also outperforms rule-based algorithms and scheduling domain methods by 4.1% and 19.1% in quality of service latency.

Keywords Non-volatile memory · NUMA · Reinforcement learning · Thread scheduling

1 Introduction

As the emerging non-volatile memory (NVM) technology rapidly advances, it is a growing demand to equip an extensive memory system for data-centric applications (Chen et al. 2020; Arulraj and Pavlo 2017). For instance, Intel delivers Optane Persistent Memory to support applications with a large memory footprint in cloud environments (Intel Technology Brief 2022). NVM has demonstrated the merits of enormous capacity, data persistence, flexibility, as well as low energy cost (Liu et al. 2016). In practice, NVM satisfies the memory requirements of the data-intensive applications, because it provides a larger capacity than DRAM under a given area, and it directly connects to the CPU via the memory bus. Therefore, more data blocks can be located in memory, exhibiting a higher speed than that in hard disks. As Intel White Paper (2019) has shown, deploying MySQL on a server equipped with NVM can handle 2.9x retail or order processing transactions compared with the NVMe SSD server. This performance improvement is obtained because more data can be placed on memory systems, and NVM provides a much larger memory capacity than DRAM-only systems.

Servers used in the cloud environments often employ the *Non-Uniform Memory Access (NUMA)* architecture where memory resources are shared among the NUMA nodes. Since remote memory access is slow, balancing the local and remote memory accesses is a primary challenge for the NUMA scheduler. Researchers have proposed many thread scheduling methods to exploit memory and computing resources efficiently. For example, the work in Piggin (2002) schedules threads among NUMA nodes to balance the CPU

✉ Lei Liu
liulei2010@buaa.edu.cn

[1] SKLCA, ICT, CAS, Beijing, China

[2] Capital Normal University, Beijing, China

[3] Beihang University, Beijing, China

[4] UCAS, Beijing, China

utilization, thus avoiding too many pending requests. However, this method may likely aggravate NUMA's remote memory access problem. Hence, researchers proposed the automatic NUMA balancing mechanism (Van Riel and Chegu 2014) that schedules data or threads from remote nodes to local nodes to alleviate the long latency problem of remote memory accesses (Piggin 2002).

To improve the performance of NUMA architecture, NVM has now been widely equipped on NUMA nodes to provide large memory capacity. It is often deployed together with DRAM, constructing a new memory system called *DRAM-NVM hybrid memory system*. Due to the reading and writing asymmetric property between NVM and DRAM, the memory system needs different access strategies. In the latest study (Wang et al. 2020), the remote NVM bandwidth is up to 30× slower than the local NVM bandwidth, but the maximum bandwidth gap between the local and remote DRAM is only 3.3×(Yang et al. 2019). This observation poses a new challenge to the thread scheduler. Previous thread scheduling approach (Piggin 2002), which trades the balance of memory accesses for high computing utilization, leads to significant overheads on the memory system that includes NVM. Several studies (Yang et al. 2020; Wang et al. 2020) reduce remote memory access by scheduling threads from remote NUMA nodes to local NUMA ones to improve system performance. However, too many threads on the local node lead to an imbalance of computing resources among NUMA nodes. The study (Yu et al. 2017) on memory scheduling strategy reduces imbalanced memory access by migrating memory pages from remote nodes to a local one. The cost of migrating data pages on NVM to different nodes is expensive (2.8× of DRAM (Wang et al. 2020)). Besides, this method cannot well guarantee data consistency. By contrast, we devise a smart thread scheduling by taking into account the hybrid memory's NUMA feature.

To realize a high-performance thread scheduling scheme on the NUMA with a hybrid memory system, we should consider the complicated features of threads due to the diversity of applications. The threads should face different memory patterns, which prior thread scheduling methods have not appropriately considered. *Reinforcement learning* has become a promising solution in the system field, such as hardware prefetching (Bera et al. 2021) and networks-on-chip (Lin et al. 2020). In this work, we are motivated to explore the reinforcement learning algorithms (Mnih et al. 2015; Ipek et al. 2008; Levy et al. 2020; Nishtala et al. 2020) to build the thread scheduling scheme. Specifically, we design *Smart Scheduler* (denoted as *SS* in the following), a reinforcement learning-based scheduler to improve the overall system performance on NUMA architecture with the latest commercial NVM-based memory. SS uses the run-time states of the system as inputs and customizes a Reinforcement Learning (RL) model to perform scheduling. In summary, we make the following contributions in this paper:

- **Using reinforcement learning to guide thread migration** To the best of our knowledge, Smart Scheduler (SS) is the first RL-based scheduler that schedules many threads simultaneously according to thread characteristics on NUMA servers equipped with NVM. SS trades off CPU utilization and NVM bandwidth to achieve an optimal solution. SS leverages enhanced *LinUCB* for algorithm convergence speed in the decision-making process. Besides, SS does not require the programmer to provide prior knowledge about the program information and data location.
- **Studying the system characteristics of the NUMA machine equipped with NVM.** We study the system characteristics of the NUMA system with NVM for multi-thread applications. We take an insight into the critical factors that impact the application performance. These factors can be used in the reinforcement learning algorithm. Besides, we create the path-bandwidth-CPU utilization rate as the indicator of the optimization objective of reinforcement learning algorithms.
- **Real implementation and detailed comparisons.** We implement SS prototype on the off-the-shelf NUMA server with a hybrid memory system (i.e., each CPU node has a hybrid DRAM-NVM system; NVM is Intel Optane Memory). SS works between the OS layer and the user layer. We compare SS with the rule-based scheduler and the original OS scheduler, and we show that SS is a promising approach.

2 Background and motivation

2.1 Thread scheduling scheme on NUMA

New multi-core systems often use non-uniform memory access architecture (NUMA) with non-volatility and large capacity. As we know, the location of threads and memory is critical to NUMA systems' performance. The industry has never stopped exploring thread migration techniques on NUMA to alleviate NUMA overheads and improve system performance, such as asymmetric interconnection oriented technique (Lepers et al. 2015), NUMA-aware (Li et al. 2007) and priority-aware (Blagodurov et al. 2010) techniques, memory partitioning (Kiefer et al. 2013) and data dependency (Virouleau et al. 2016) based techniques. Among them, the work in Piggin (2002) improves the program's performance on NUMA by balancing the computing resources between different NUMA nodes and has been widely adopted until now. However, NVM poses new challenges to

the traditional NUMA mechanisms due to the inconsistency and balance of NVM and DRAM. Our work is conducted on an off-the-shelf NUMA platform, and each of its nodes has a DRAM-NVM hybrid memory system composed in a horizontal approach (Liu et al. 2016). We show new challenges as below.

2.2 New challenges

2.2.1 Static scheduling vs. dynamic scheduling

Previous studies rely on the experience of skilled programmers to bound threads on specific nodes when the threads start (McCurdy and Vetter 2010). Threads are often bounded to the assigned nodes and will not be migrated to other nodes during the run time.

This static scheduling method may impair the overall system performance for three reasons. (1) Different programs exhibit different computing and memory access patterns. Therefore, the decisions made by skilled programmers lead to inferior results in many cases. (2) Different programs' memory and computing behaviors vary during runtime and are difficult to predict. Hence, the static scheduling rule likely causes performance degradation and unqualified *quality of service (QoS)* due to the varying program behaviors. (3) NVM memory's read and write accesses present the asymmetric property in terms of latency and bandwidth. The static rules in previous efforts have not considered these situations and, therefore, cannot support the NUMA system with DRAM-NVM hybrid memory.

2.2.2 Migrating thread vs. migrating data

To shorten the time for accessing data on remote nodes, it is a problem whether to migrate thread or migrate data. We choose thread migration rather than data migration for three reasons. (a) The NVM supports byte access, but the previous data migration method is designed for page granularity. Therefore, the existing data migration method does not apply to the DRAM-NVM hybrid memory system. (b) To maintain consistency, each data movement and modification on the NVM must follow the coherence mechanism. The coherence mechanism may cause additional overhead upon each data modification. (c) On the DRAM-NVM hybrid memory system, the endurance of NVM is worse than that of DRAM. Therefore, frequent data migration will quickly shorten the NVM lifetime.

2.2.3 Balance of computing resources vs. competition of computing resources

The balancing operation of computing resources introduces remote accesses. Yet, too many local accesses aggravate thread competition on the same node. In practice, we observe that the thread remote accesses to NVM is expensive than that of competing for computing resources on the same CPU. Therefore, we cannot just consider the balance of computing resources during scheduling.

Table 1 shows the correlation between the proportion of remote threads access (72 threads in total) to the NVM and the execution time of them. There are 36 cores on the current node, so each core has two competition threads on average. As the number of threads that exhibit remote access decreases (Table 1), the local CPU becomes saturate, and the competition between threads for CPU will become serious. When all threads are located on the local CPU node, the remote access ratio is 0. We observe that the total time consumption of the program is lower. This is because the overhead of the thread cross-node access is more significant; the remote data accesses to NVM are much slower than local access, which exceeds the overhead of competing CPU resources. We can migrate threads to data or migrate data to threads rather than remote data access based on these observations.

2.3 The goal of our design

The Linux kernel has the concept of CPU affinity to support the migration of threads and control the position of threads by enabling the *sched_setaffinity()* function. NVM is accessed via the *direct access interface (DAX)* (The Linx Kernel documentation, 2022). Our goal is to devise the technique to predict whether the current access pattern will lead to longer access paths or lower bandwidth and automatically make the appropriate migration decisions. Our goal is to achieve a high overall bandwidth utilization and, therefore, a high overall system performance.

3 Correlation analysis

In this section, we study the performance of the representative multi-thread programs on a NUMA server with a hybrid memory system. First, we collect the traces of write-dominate and read-dominate applications, respectively. Then, we analyze the correlation between the performance of the applications and the thread characteristics (shown in Table 2), and we provide insights for designing

Table 1 The correlation of remote threads access and running time (72 threads in all)

Remote Threads Access (%)	10	5	3	1	0
Time Cost (s)	151.08	129.35	126.97	123.97	123.04

Table 2 The events and indexes monitored in our system

	Input	Description
THREAD	Mjflt	Major faults are the number of page faults that caused Linux to read a page from disk on behalf of the process.
	Minflt	Minor faults are the number of faults that Linux could fulfill without resorting to a disk read
	IPC	Instructions per CPU cycle
CORE	Remote NVM Access	The num of instructions that data source is in remote-NVM
	Local NVM Access	The num of instructions that data source is in local-NVM
	Remote DRAM Access	The num of instructions that data source is in remote-DRAM
	Local DRAM Access	The num of instructions that data source is in local-DRAM
	L1 Cache Miss Latency	L1 cache miss latency
	CPU Frequency	Core frequency in run time
	NVM Read Bandwidth	Read bandwidth on NVM
OS	NVM Wirte Bandwidth	Write bandwidth on NVM
	CPU Utilization	Sum of work handled by a CPU

Table 3 Platform Specification

Configuration	Platform
CPU Model	Intel(R) Xeon(R) Gold 6240M
OS	CentOS 7.6.1810 (kernel 4.18.0)
Sockets	2
Logical Processor Cores / Socket	36 Cores (18 physical Cores)
Processor Speed	2.60GHz
Main Memory / Channel / BW	256GB, 2933MHz DDR4 / 4 Channels / 87.42 GB/s
Private L1I & L1D & L2 Cache Size	32KB & 32KB & 1MB
Shared L3 Cache Size	24.75MB / 11 ways
Optane DCPMMs	1TB / 4 Channels
Ultra Path Interconnect(UPI) (Intel UPI 2022)	10.4GT/s
NVM/DRAM Capacity per Socket	512 / 128 GB

new schedulers. Please note that the "Remote/Local NVM/DRAM Access" in Table 2 denotes the memory accesses that come from the CPU's last level cache misses.

3.1 Platform and performance monitoring

The platform used in our work is described in Table 3. Our platform has two nodes, and each is equipped with a hybrid DRAM-NVM system. We employ *Pmemkv* (PDMK 2022), which is a local/embedded key-value datastore optimized for persistent memory. The memory access features of the typical NVM-optimized programs can be captured through changing storage engines and access modes. We use the *Processor Counter Monitor (PCM)*, and its corresponding user library (Intel PCM 2022), to capture platform features. We collect the traces for CPU cores and OS for applications. These traces reflect the impact introduced by the running programs on the cores, OS, and cross node data access.

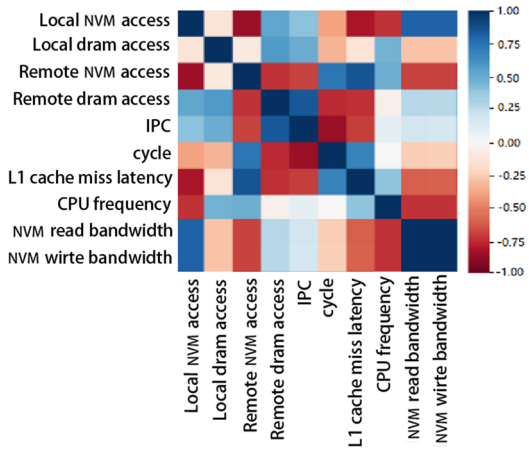
Besides, the types of events that PCM can monitor are limited. Therefore, we also use the *proc* file system's kernel interface to monitor relevant data generated by system activities, such as core frequency, page loss interrupts, etc.

3.2 Correlation analysis of write-dominated applications

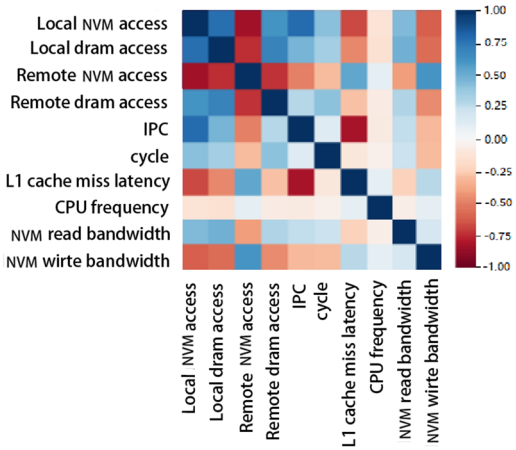
We use *Pmemkv* to build typical write-main program on NVM in which the thread number is 64, and random write operations are performed concurrently in a 50GB address space.

In order to clarify the impact of remote/local thread write operations on the overall bandwidth, we experiment with two steps. (1) We bind all threads and their accessed data onto node 0. (2) We bind all threads to node 1 but place their data on the NVM on node 0. Then, we collect the program performance traces of execution time and analyze the correlation, which are presented in Fig. 1.

We focus on the characteristics closely related to NVM performance (i.e., NVM write and read bandwidth). As shown in Fig. 1a, the local NVM access has a highly positive correlation with NVM bandwidth, but a significant negative correlation with the remote NVM access, indicating the local access has advantages. The L1 cache miss latency is negatively correlated with the NVM bandwidth because the cache misses cause additional memory access or disk data loading. As for CPU frequency, it shows a high negative correlation with the NVM bandwidth. The reason behind is that the cores need to provide more additional operations for remote access than that for local access. It can be proved in this figure that the CPU frequency presents a negative correlation with a local NVM access but a positive correlation with a remote NVM access.



(a) write-dominated applications.



(b) Read-dominated applications.

Fig. 1 Spearman correlation (Ban 2019; Bonett and Wright 2000) of the thread features for multi-thread programs based on NVM. The value of Spearman correlation coefficient is between - 1 and + 1. It is used to measure the monotonicity of two features, which is more sensitive to the nonlinear monotone correlation. The deeper the blue, the higher the positive correlation, and the deeper the red, the higher the negative correlation

3.3 Correlation analysis of read-dominated applications

We still employ *Pmemkv* to construct a typical read-dominated program on NVM in which the thread number is 64, and random write operations are performed concurrently in a 50 GB address space. The results are shown in Fig. 1b. We look into the features of NVM write bandwidth and NVM read bandwidth. As shown in Fig. 1b, the L1 cache miss latency shows a positive correlation with NVM read bandwidth, but a negative correlation with NVM write bandwidth. The reason is that the write operation will consume more time for loading data. Therefore, this index can help us to distinguish the read and write operations of NVM. In addition, the access number of local NVM is positively correlated with NVM read bandwidth but negatively correlated with NVM write bandwidth. The underlying reason is that NVM read operation has higher priority than write operation, indicating that more weight should be given to write operations to balance the impact of read priority when we design the scheduling policies.

4 The overview of our design of SS

We present SS, a thread scheduling mechanism for hybrid memory systems to shorten the access path of memory and automatically determine the optimal thread distribution for different access patterns. SS is designed to balance computing resources and memory resources to reduce access latency to NVM. It advances the current thread scheduling practice in three ways. (1) Replace the existing static thread scheduling algorithm with the adaptive intelligent thread scheduling algorithm. (2) Expand the traditional CPU-utilization-based dimension into the multi-parameter dimension. (3) Exploit the reinforcement learning algorithm to correct the errors during runtime scheduling.

4.1 Workflow

Figure 2 shows the overview of the high-level workflow of SS. Based on the fact that the access behavior of the application is changing, in the characteristic value collection module ①, we collect feature data from all levels of the architecture and temporarily store them for subsequent use, shown in Fig. 3. According to the data in ① of contextual information about NVM resources (such as remote access, CPU load, read-write ratio, and bandwidth), the choice of imbalanced cores module ② determines whether to call ③ to alleviate the NUMA phenomenon or computing resource enrichment. The state of the RL model contains the parameters in Table 2. The state of the RL model represents the system condition. The RL model leverages the LinUCB algorithm to decide the number of threads to be migrated. The scheduling decision module ③ migrates the specific thread of the target application to other NUMA nodes according to the runtime state of the current server using our scheduling strategy. Finally, the validation and execution operations are performed on the server by the mitigation engine ④. In addition, ④ helps eliminate the uncertainty RL model in ③ through the real-time feedback of post-deployment effects.

4.2 Characteristic values collection

The platform details and performance monitoring methods are presented in Sect. 3.1. In our system, the hardware

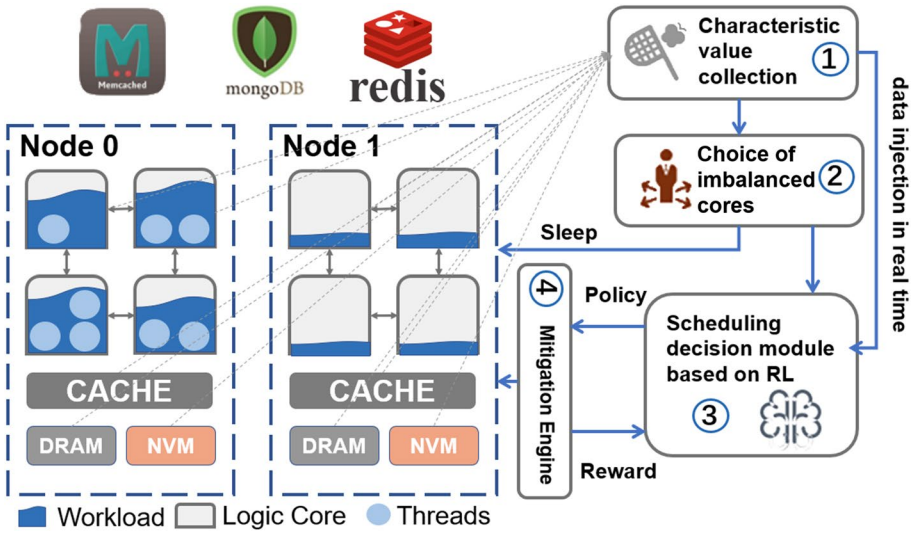


Fig. 2 High-level workflow of SS

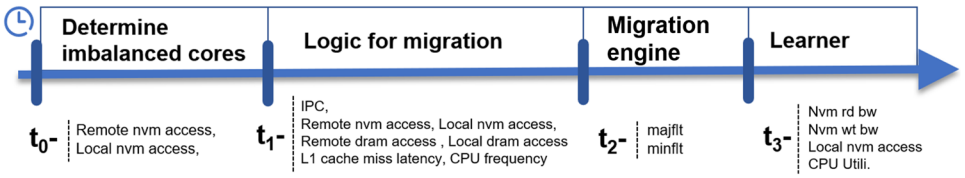


Fig. 3 The traces needed by intelligent scheduler in different phases

counters are used to dynamically collect the programs' runtime information. Compared with the traditional information collection methods, such as software instrumentation, the hardware counter brings less overhead and less interference to the execution of programs. The related technologies have been widely used in OS supports (Azimi et al. 2009). Besides, the types of events that PCM can monitor are limited. We also use the proc file system's kernel interface to monitor relevant data generated by system activities, such as core frequency, page loss interrupts, etc. We set the single sampling duration to 1s, and the indicators involved in each stage of the scheduling are shown in Fig. 3.

4.3 Key optimization metric

SS's goal is to achieve a balanced solution between CPU utilization and the shortest access path to memory. To this end, defining an effective evaluation metric is crucial. In smart scheduling, we consider the three factors: access path length, CPU utilization rate, and system bandwidth. Therefore, we comprehensively design these three parts as the evaluation metric of system performance. Moreover, they are the parts of the reward function of the reinforcement learning algorithm, which evaluates the actions decided by the reinforcement learning algorithm.

$$Reward = \alpha \cdot LNA + \beta \cdot BWVN + (1 - \alpha - \beta) \cdot Util \quad (1)$$

Local NVM Access (LNA) represents the change in the number of times a thread accesses local data before and after migration. Util represents the change in the total system bandwidth utilization before and after migration. *Bandwidth Weight Value of NVM (BWVN)* is defined as:

$$BWVN = BW_{nr} + 3 \times BW_{nw} \quad (2)$$

BW_{nr} is the sum of the system's NVM read bandwidth, BW_{nw} is the sum of the NVM write bandwidth, in MB/s. The write bandwidth on NVM is lower than the read bandwidth, so remote write access is more harmful. Therefore, we increases the weight of the write bandwidth to balance the impact of this difference. The weight is taken from the NVM read and write bandwidth difference: 3x.

The formula (1) is our final indicator for comprehensively considering the performance of the system. The value range of α is $0 \leq \alpha < 1$, and the value range of β is $0 \leq \beta < 1$, and $\alpha + \beta \leq 1$. Since the cost of remote access is greater than the cost of thread competition for CPU, we increase the weight of the first two and reduce the weight of CPU utilization. In this article, we focus on the NVM accesses which might impair the system performance, so the coefficients of *LNA* and *BWVN* is larger than that of*Util*. After our experiments

of tuning the coefficients, $\alpha = 0.45$, $\beta = 0.45$ perform best. The final formula can be expressed as below:

$$Reward = 0.45 \cdot LNA + 0.45 \cdot BWVN + 0.1 \cdot Util \quad (3)$$

5 Finding access imbalanced cores

The first step is to confirm which cores have access imbalance. To avoid the overhead of sampling all threads, we need to have an algorithm to choose the imbalance cores.

We analyze the wrong thread distributions cases, and formulate the imbalance core judgment rules. The formula to determine whether the core is unbalanced is as below:

$$P_{ub} = \frac{REMOTE\_NVM}{REMOTE\_NVM + LOCAL\_NVM} \quad (4)$$

where P_{ub} is the ratio of remote access number to the total number of NVM accesses. We tend to reduce the frequency of migrating threads due to the migration overheads. In this work, we use 80% as the threshold, which means that threads should be migrated when $P_{ub} \geq 80\%$. Alg. 1 shows how to find the unbalanced cores. 80% is an empirical value that we find it works well for our current experiments. Our system has an interface that people can change this parameter accordingly.

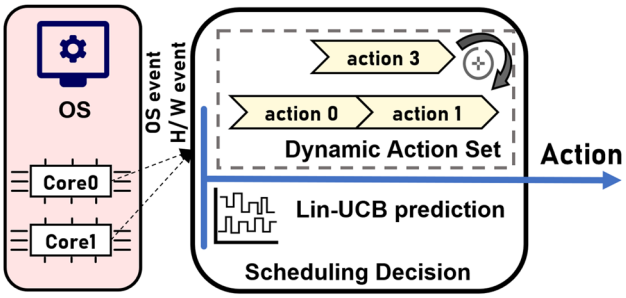


Fig. 4 The workflow of adaptive decision based on Lin-UCB

Mohri 2005), and employ the LinUCB algorithm (Li et al. 2010; Agrawal and Goyal 2012) to intelligently make decisions. The workflow of intelligent scheduling is presented in Fig. 4.

6.1 Bandit modeling

The straightforward way to determine whether a thread should be migrated or not is to evaluate the application's impact before and after the migration. We always take actions that maximize the expected return, but also need to avoid falling into a locally optimal solution. In essence, this is a classic exploitation and exploration problem (Auer 2002). The exploitation and exploration problem can be modeled as a multi-armed bandit problem where we have

Algorithm 1 Rule-based selecting for unbalanced cores

Input: Remote NVM Access, Local NVM Access
Output: List of unbalanced cores

```
Initialize unbalanced_cores_list = empty;
while each logical core do
        Collect the access information of NVM
    if P_ub > 80% then
        Add current core to the unbalanced_cores_list
    else
        continue
    end if
end while
        return unbalanced_cores_list
```

6 Decision logic for thread migration

After determining the unbalanced core, we need to decide the number of threads that should be migrated. The traditional static decision-making methods based on domain knowledge are inefficient. This motivates the adaptive decision design of SS. In this paper, we abstract the thread migration as a multi-armed bandit problem (Vermorel and Mohri 2005). Our goal is to ensure the balance by comparing the actions known as the most profitable and the unknown actions that may be more successful.

Exploration Algorithm. We need to explore different actions to see which action is the most beneficial to the many possible action and each action will return different feedback. Therefore, we model the thread migration as a multi-armed bandit problem (Vermorel and Mohri 2005). Our goal is to ensure the balance by comparing the actions known as the most profitable and the unknown actions that may be more successful.

application. However, at the same time, because the thread access situation on each core is different, we also need to use additional information to describe the decision-making process. In other words, we need to make decisions based on the access characteristics of different cores. We use the LinUCB as the algorithm of reinforcement learning because it provides suitable personalized decision-making scenarios, flexible action strategies, and more interpretability. In LinUCB, we model the reward value as a function of the action and model parameters and select the action based on maximizing the sum of the expected reward and the confidence interval. At the crux of the algorithm, it is expected to find the arm with the highest UCB at time t:

$$a_t \equiv argmax_{a \in A_t}(x_{t,a}^T \hat{\theta}_a + \alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t-a}}) \quad (5)$$

where a_t is the action chosen to perform, $x_{t,a}^T \hat{\theta}_a$ is the expected value of action, where θ_α is result from ridge regression, $\alpha \sqrt{x_{t,a}^T A_a^{-1} x_{t-a}}$ is corresponding upper confidence bound, where α is a hyper parameter. The higher α is, the wider the confidence bounds become. Thus it results in a higher emphasis placed on exploration instead of exploitation.

Dynamic action set. After the unbalanced core is confirmed, SS will choose among several possible scheduling actions. The action implies that the to-be-migrated threads on the core to the target node. Since the number of threads on the core changes dynamically, it is required that the output of the algorithm corresponds to the same number of actions. Otherwise, if the number of threads is larger than the number of actions, some threads will not have matching actions, causing threads not to participate schedule. On the other hand, if the number of threads is less than the number of actions, some actions will have no practical significance, thus causing prediction errors.

In order to solve the above problems, this paper proposes a *dynamic action set* method. i.e., to dynamically maintain an action set: We first detect the number of application threads on the unbalanced core. Supposing the number of threads is larger than the current number of actions, the algorithm dynamically increases actions and loads parameters so that each thread has a corresponding number of actions. Otherwise, if the number of threads is less than the current number of actions in the actual algorithm prediction, we reduce the number of actions involved in the prediction and only allow actions with the number of threads to participate in the calculation process and algorithm update.

7 Migration engine

The migration engine is the core component of SS. Internally, it is mainly composed of three modules, as shown in Fig. 5. All modules are organized together to implement the reinforcement learning algorithm, thereby completing the entire scheduling logic.

There are three key issues to be solved in the *migration engine*: (1) selecting the to-be-migrated threads; (2) determining their migration destinations; (3) performing the migration.

7.1 Select the to-be-migrated threads

After the migration decision is made by the scheduling algorithm, the following question is which threads on the imbalanced core need to be migrated. First, we design a function to filter the to-be-migrated threads. There are two key indicators in the filter: *majflt* and *minflt*. The first one, *majflt*, refers to the major fault that causes the process to read the disk's page; and the latter one, *minflt* denotes the number of interrupts that Linux can complete processing without reading the disk.

We maintain two variables majflt and minflt for each thread to evaluate the frequency of the thread's page fault interruption. Once creating the thread migration task, we sort to-be-migrated threads on the core according to the value of majflt. If the size is equal, it will be further sorted according to the size of minflt. For the sorted list of to-be-migrated threads, we select threads according to the scheduling algorithm's action values. For example, if the algorithm's action value is 2, we will migrate the top 2 threads in the list to other nodes.

7.2 Determine the migration destination

Once a thread to be migrated is selected, the next step is to determine the migration destination. This determination

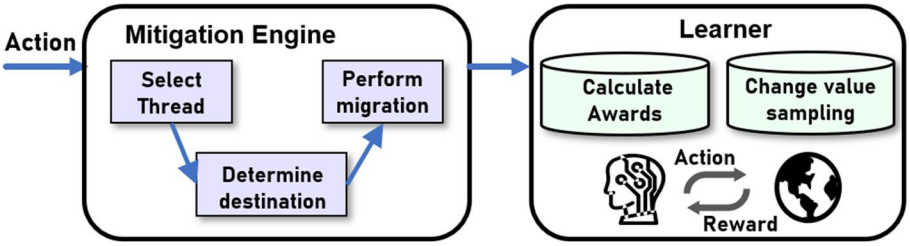


Fig. 5 The architecture of Mitigation Engine and Learner. The Mitigation Engine is consists of thread selecting, destination determining and migration performing

requires a clear understanding of where the data are originally stored. If the thread is migrated from one wrong node to another wrong node, it will pay a high cost. The reason is that if migrating to the wrong node, we still have to load data from the remote node to reheat the cache.

If the correct node is chosen, CPU competition is another consideration. For the scenario where multiple threads are on the same CPU, it is known that extra time is required for each thread to wait for execution opportunities. In order to solve the above problems, this paper proposes a strategy to determine the migration destination. In order to relieve CPU competition, after confirming the target node, we sort the cores according to the CPU utilization rate in the node and then select the core with the lowest CPU utilization rate as the final migration destination. In this way, we can complete the entire migration process.

7.3 Perform migration

The Linux kernel has *cpu_affinity* interface to support thread binding. In this way, OS can control the location of thread nodes. In the kernel, all threads maintain a data interface named *task_struct*, where the *cpus_allowed* bit mask is directly related to affinity. The mask consists of N bits, where N corresponds to N logical cores in the system. SS can force the thread to be bound to the migration target core by setting the thread's flag to be migrated, thus executing the entire process of thread migration.

8 Learner design

The learner is one of the components of SS. It can understand the impact of different scheduling actions on the system environment. The architecture is shown in Fig. 5. Application memory access behavior, hardware and software changes caused by actions can be quickly learned and applied in the scheduler decision making.

This component is mainly responsible for two tasks: reward value collection and reinforcement learning algorithm training. The reward value collection function samples the current NVM access information, system bandwidth, and CPU utilization. It then correlates the sampled information with the state information before the migration to determine the migration thread's cost and the impact on the system environment. The training of reinforcement learning is mainly carried out online, and the trained parameter values will be stored in the suffix of *.npy* file and record the current training times. The algorithm deployed on the new platform can quickly load parameters from it.

Table 4 Pmemkv workload patterns

Workload	Pattern
readrandom	100% random read
readseq	100% sequential read
fillrandom	100% random write
fillseq	100% sequential write

9 Evaluation

Our evaluation answers the following questions. (1) How does SS perform on real NUMA machines with hybrid NVM memory? (2) How about SS's adaptability? (3) How does SS compare to the rule-based/traditional scheduling algorithms?

9.1 Workload

Pmemkv Scargall (2020) is a high-performance local/embedded key-value data repository optimized for persistent memory, widely used in persistent memory programming in current cloud computing systems. We use Pmemkv as the benchmark to generate workloads. Specifically, the experiment uses four access modes (Table 4). The key size is 16 bytes and the value size is 400 kB. 20 k keys are stored in the database. The number of threads is 48.

9.2 Effectiveness of SS

We compare SS with the following competing methods:

Rule-based Algorithms (Rule) (Yang et al. 2020): It is a recent rule-based thread scheduling method, which reduces the remote accesses to NVM on NUMA. The key mechanism is derived from the experience and learned knowledge.

Scheduling Domain (SD) (Siddha et al. 2005): This strategy relies on the original OS scheduler: Scheduling Domain. Its core idea is to balance the waiting queue length of processes on idle CPUs and busy CPUs.

We demonstrate the effectiveness of SS from the two perspectives as below.

9.2.1 Comparing SD, rule and SS

Figure 6 compares the program's execution time with the same configuration under three different thread scheduling approaches. The figure's x-axis represents four different working modes. These four situations cover the most commonly used access modes of the application. The y-axis represents the execution time. Fig. 6 has three sub-graphs. From top to bottom, they are *Real Time* (the real clock spent by the program from the start to the end), *User Time* (the program CPU time spent in the kernel mode), and *Sys Time* (the program CPU time spent in the user mode). *User+Sys* time represents how much CPU time the process uses, which

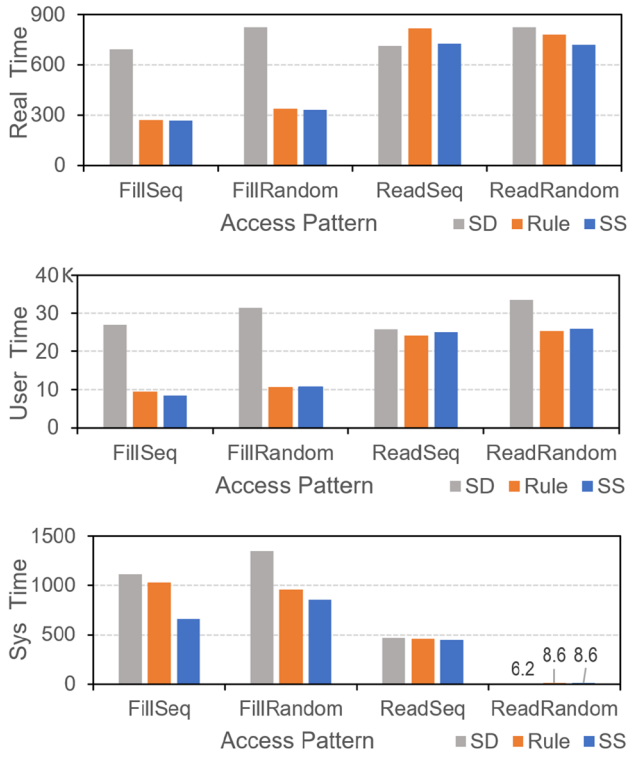


Fig. 6 Experimental results of comparison of SD, Rule, SS under different access modes

is longer than Real Time in cases where many threads are co-running.

Comparing the program's test results in Real Time, we find that compared with the traditional Scheduling Domain method, SS can reduce the program execution time to 59.9% at most with the write-oriented test set. The execution time is the same magnitude as the Rule-based algorithm. Besides, SS can reduce the program execution time by up to 12.4% in the test set where reading is the majority. SS can be applied to applications with different memory access patterns, especially for the applications that use NVM. In the worst case, the performance of SS is degraded by only 2.1% compared to the best method. For the read-majority benchmarks, the SS outperforms the Rule method by 10.7%. The percentage is obvious especially in the continuous read benchmark case. We summarize the reasons why SS performs better in these cases. In one sentence, SS is more intelligent than previous schedulers.

- *SS schedules the read thread carefully.* Remote read operations for NVM data sources are less affected by NUMA, but the cost of thread migration is high. In this case, thread migration interrupts the sequential access process. After migration, re-heating is needed for the cache to load data, which ultimately hurts program performance. SS handle these cases carefully. It means that SS does not always migrate the read threads. Therefore, SS can have a better performance.
- *SS takes into account the imbalance of computing resources.* As mentioned before, the imbalance of computing resources is count for much. To reduce the intensity of competition for computing resources on the local node, SS migrates part of the read threads from the remote node to the local node (migrating more threads is not a good way), thereby the length of local CPU waiting queue is not very long.
- *SS makes scheduling decisions based on thread characteristics.* Compared with the rule-based algorithm, SS makes scheduling decisions based on the current OS event/thread memory access characteristics. This dynamically scheduling method fully considers the runtime thread characteristics, thus delivering better performance than the static rule-based algorithm.

9.2.2 Case studies: SS improves quality of service (QoS)

We use a typical example to show that SS can improve QoS. The workload used in the example is *fillrandom*; the number of requests is 960k; the single write size is 400kB; and the number of threads is 48. We run the workload and schedule threads with SD, Rule, and SS. Compared with SD and Rule, SS reduces P99 request latency by 19.1% and 4.1%, respectively. P99 request latency denotes the worst latency that was observed by 99% of all requests. SD has the worst request latency. This is because SD schedules threads evenly across all NUMA nodes, so it takes longer to migrate data for threads from remote nodes.

9.3 Studies on the convergence of the thread scheduling approaches

A key requirement of RL-based thread scheduling algorithm is its convergence speed. The algorithm is required to adapt quickly and provide correct scheduling decisions under realistic conditions where application behaviour changes frequently. The key challenge is whether the algorithm can complete the scheduling space exploration quickly. In this section, we show that SS converges quickly and can provide effective scheduling decisions for new cases.

We build four benchmarks that simulate real access mode on the experimental platform, and further test the convergence speed of SS with a learning rate of 0.3. In LinUCB, the confidence interval refers to the range of plausible values of the reward when performing a certain scheduling action. It is given by the Upper-Confidence-Bound (UCB) and can be used to judge whether the algorithm has converged. The larger the confidence interval is, the less certain our algorithm is about the potential benefits of scheduling actions, and vice versa. Each time the model makes a scheduling

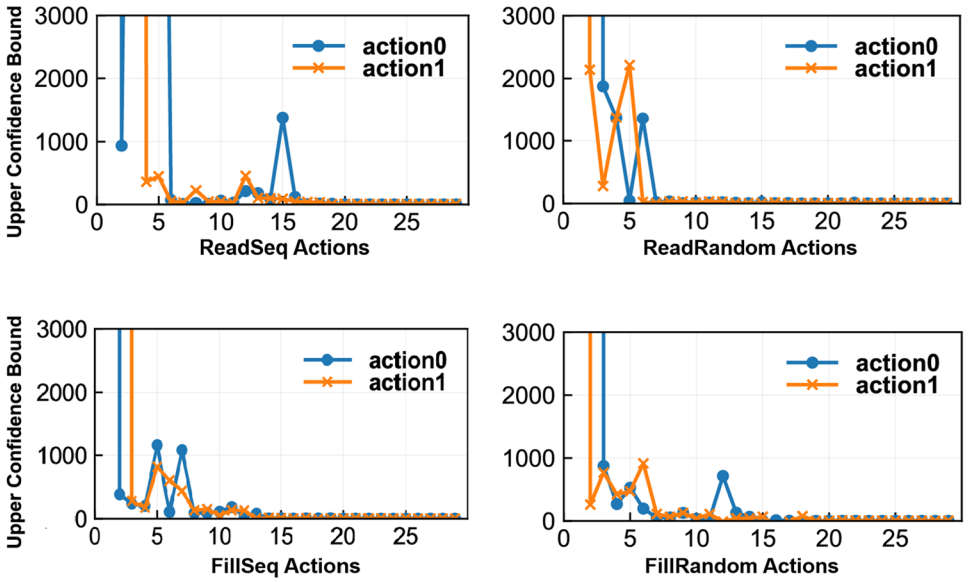


Fig. 7 SS convergence speed evaluation

decision, it chooses the action with the Max (Expected Value + UCB). When an action is performed, the UCB value is updated, thereby further eliminating the uncertainty in the scheduling space.

Figure 7 shows the UCBs corresponding to the two scheduling actions of SS when scheduling the four workloads. In general, the upper bound of the confidence interval can converge to a stable range within 20 scheduling epochs, indicating that our algorithm can accurately assess the potential benefits of different thread scheduling actions quickly. In practice, our algorithm can quickly adapt to new environments and provide correct scheduling decisions with negligible learning costs.

10 Discussions and future work

In our work, since a few NVM adaptation applications are currently available, we use Pmemkv to generate the benchmarks. Although the Pmemkv is widely used in persistent memory programming, it is limited to simulate the complex characteristics of commercial applications. The platform used in our work has a dual CPU socket with Intel Xeon Gold 6240M CPU@2.60GHz, and 512G Optane DCPMMs on each NUMA node, which still has a gap with the large-scale commercial cloud platform. The performance loss of commercial cloud platforms due to remote memory access is heavier than that on our platform. Therefore, we think our smart scheduler can be a promising approach. Artificial Intelligence has made significant achievements in many aspects, but open-source schedulers based on artificial intelligence are still rare. We hope to make our work publicly available and hope our work can benefit the future researchers and engineers. In the future work, we plan to carry out the following work: (1) Test more applications to fit the virtual application environments; (2) Open source our SS; (3) Collaborate with vendors that provide cloud services, and deploy the scheduler on the sizeable commercial cloud platform.

We further test workloads with different read and write rates, i.e., read rate varies from 50%, 70% to 90%. SS performs slightly better than SD (the OS original scheduler), but the Rule-based scheduler outperforms both SD and SS in these cases. For the case with 50% write and 50% read, SS and SD take 25 and 27 minutes, respectively; by contrast, the Rule-based scheduler only takes around 9 minutes to provide an ideal solution. For the case with 70% read and 30% write, SD and SS take about 18 minutes, but the rule-based scheduler only takes 7 minutes. For the case with 90% read and 10% write, all the three schedulers perform similarly and can converge within around 4 minutes. The results show that SS does not exhibit the advantage for these workloads with mixed read/write operations. The reasons behind this are multi-folded. (1) In some cases, using the simple linear regression (Chen et al. 2022; Liu 2019; Lv et al. 2014, 2015) to make the decision can be a good approach with low overheads. Therefore, it will be a promising idea that devises an adaptive approach that combines ML models and other rule-based approaches. Our future work will further study the problem along this thought line. (2) ML models perform differently (Chen et al. 2022; Liu 2019), and the ML model used in this work might not handle complicated mixture cases well. (3) Only using RL might not be a good idea in some cases, and using multiple ML models to work cooperatively can be an ideal approach (Liu 2019). As RL models' performance relies on the starting point in the scheduling exploration space (Liu 2019), proactively employing well-trained ML models to provide the first step solutions for

them (Liu 2019; Liu et al. 2016) can be an ideal alternative. Moreover, our future work will further study how the ML models perform in reality and will also have more efforts on how to deploy our work on NUMA systems with more CPU nodes.

11 Conclusion

In the context of the hybrid memory system on NUMA, we propose a new reinforcement learning approach to improve the overall system performance. We introduce Smart Scheduler (SS), an online, lightweight thread scheduler to schedule threads across NUMA nodes equipped with NVM. SS employs linUCB to evaluate the best thread scheduling actions continuously. We test SS on a real machine equipped with Intel Optane NVM. SS can reduce the execution time of multi-threaded programs by up to 59.9% compared with the previous thread scheduling strategy. It also outperforms rule-based scheduling and scheduling domain methods by 4.1% and 19.1% for quality of service in some cases. We show that ML can be a promising approach for OS scheduling on new architecture. Moreover, we also show that using ML models for system optimizations should be careful, as they don't consistently outperform other approaches. Our future work will further study how to leverage ML/AI technologies for system optimization and design.

Acknowledgements We thank the reviewers for their valuable comments. This work is supported by the Key Research and Development Program of Guang Dong (No. 2021B0101310002), NSFC under grant No. 62072432. Yuetao Chen is a student member in Sys-Inventor Lab led by Lei Liu.

References

Agrawal, S., Goyal, N.: Thompson sampling for contextual bandits with linear payoffs (2012)

Arulraj, J., Pavlo, A.: How to build a non-volatile memory database management system. Proceedings of the 2017 ACM International Conference on Management of Data. SIGMOD '17, pp. 1753–1758. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3035918.3054780

Azimi, R., Tam, D.K., Soares, L., Stumm, M.: Enhancing operating system support for multicore processors by using hardware performance monitoring. SIGOPS Oper. Syst. Rev. **43**(2), 56–65 (2009). https://doi.org/10.1145/1531793.1531803

Auer, P.: Using confidence bounds for exploitation-exploration tradeoffs. J. Mach. Learn. Res. **3**(Nov), 397–422 (2002)

Ban, A.N.: Spearman correlation. (2019)

Bera, R., Kanellopoulos, K., Nori, A.V., Shahroodi, T., Subramoney, S., Mutlu, O.: Pythia: A customizable hardware prefetching framework using online reinforcement learning. 2021 54rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2021). IEEE

Blagodurov, S., Fedorova, A., Zhuravlev, S., Kamali, A.: A case for numa-aware contention management on multicore systems. 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 557–558 (2010)

Bonett, D., Wright, T.: Sample size requirements for estimating pearson, kendall and spearman correlations. Psychometrika **65**(1), 23–28 (2000)

Chen, S., Jin, A., Delimitrou, C., Martınez, J.F.: ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud. The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA-28), (2022)

Chen, Y., Peng, I.B., Peng, Z., Liu, X., Ren, B.: Atmem: Adaptive data placement in graph applications on heterogeneous memories. Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. CGO 2020, pp. 293–304. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368826.3377922

Intel: Intel Optane Technology. Website. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html (2022)

Intel: Scaling MySQL with Intel Optane Persistent Memory (2019). https://www.intel.com/content/www/us/en/architecture-and-technology/scaling-mysql-with-optane-persistent-memory.html

Ipek, E., Mutlu, O., Martínez, J.F., Caruana, R.: Self-optimizing memory controllers: A reinforcement learning approach. 2008 International Symposium on Computer Architecture, pp. 39–50 (2008). https://doi.org/10.1109/ISCA.2008.21

Intel: https://github.com/opcm/pcm. On-line Resources (2018)

Intel-UPI (2017). https://en.wikipedia.org/wiki/Intel_Ultra_Path_Interconnect

Kiefer, T., Schlegel, B., Lehner, W.: Experimental evaluation of numa effects on database management systems. Markl, V., Saake, G., Sattler, K.-U., Hackenbroich, G., Mitschang, B., Härder, T., Köppen, V. (eds.) Datenbanksysteme Für Business, Technologie und Web (BTW) 2025, pp. 185–204. Gesellschaft für Informatik e.V., Bonn (2013)

Kernel, L.: https://www.kernel.org/doc/Documentation/filesystems/dax.txt. On-line Resources

Levy, S., Yao, R., Wu, Y., Dang, Y., Huang, P., Mu, Z., Zhao, P., Ramani, T., Govindaraju, N., Li, X., *et al.*: Predictive and adaptive failure mitigation to avert production cloud {VM} interruptions. 14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20), pp. 1155–1170 (2020)

Lepers, B., Quema, V., Fedorova, A.: Thread and memory placement on NUMA systems: Asymmetry matters. 2015 USENIX Annual Technical Conference (USENIX ATC 15), pp. 277–289. USENIX Association, Santa Clara, CA (2015). https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

Li, T., Baumberger, D., Koufaty, D.A., Hahn, S.: Efficient operating system scheduling for performance-asymmetric multi-core architectures. SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1–11 (2007). https://doi.org/10.1145/1362622.1362694

Lin, T.-R., Penney, D., Pedram, M., Chen, L.: A deep reinforcement learning framework for architectural exploration: A routerless noc case study. 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 99–110 (2020). https://doi.org/10.1109/HPCA47549.2020.00018

Liu, L., Yang, S., Peng, L., Li, X.: Hierarchical hybrid memory management in os for tiered memory systems. IEEE Trans. Parallel Distributed Syst. **30**(10), 2223–2236 (2019)

Liu, L.: QoS-Aware Machine Learning-based Multiple Resources Scheduling for Microservices in Cloud Environment. arxiv, (2019). https://arxiv.org/abs/1911.13208

Liu, L., Li, Y., Ding, C., Yang, H., Wu, C.: Rethinking memory management in modern operating system: horizontal, vertical or random? IEEE Trans. Comput. **65**(6), 1921–1935 (2016)

Li, L., Chu, W., Langford, J., Schapire, R.E.: A contextual-bandit approach to personalized news article recommendation. Proceedings of the 19th International Conference on World Wide Web. WWW '10, pp. 661–670. Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1772690.1772758

Lv, F., Liu, L., Cui, H., Wang, L., Liu, Y., Feng, X., Yew, P.C.: Wise-Throttling: a new asynchronous task scheduler for mitigating I/O bottleneck in large-scale datacenter servers. J. Supercomput. **71**(8), 3054–3093 (2015)

Lv, F., Cui, H., Wang, L., Liu, L., Wu, C., Feng, X., Yew, P.C.: Dynamic I/O-aware scheduling for batch-mode applications on chip multiprocessor systems of cluster platforms. J. Comput. Sci. Technol. **29**(1), 21–37 (2014)

McCurdy, C., Vetter, J.: Memphis: Finding and fixing numa-related performance problems on multi-core platforms. 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pp. 87–96 (2010). https://doi.org/10.1109/ISPASS.2010.5452060

Nishtala, R., Petrucci, V., Carpenter, P., Sjalander, M.: Twig: Multi-agent task management for colocated latency-critical cloud services. 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 167–179 (2020). https://doi.org/10.1109/HPCA47549.2020.00023

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M.A., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. Nat. **518**(7540), 529–533 (2015). https://doi.org/10.1038/nature14236

Piggin, N.: Scheduling Domains. On-line Resources (2002). https://lwn.net/Articles/80911/

PMDK: https://github.com/pmem/pmemkv. On-line Resources (2019)

Siddha, S., Pallipadi, V., Mallick, A.: Chip multi processing aware linux kernel scheduler. In: Linux Symposium, vol. 193 (2005). Citeseer

Scargall, S.: pmemkv: A Persistent In-Memory Key-Value Store, pp. 141–153. Apress, Berkeley, CA (2020). https://doi.org/10.1007/978-1-4842-4932-1_9

Van Riel, R., Chegu, V.: Automatic numa balancing. Red Hat Summit (2014)

Vermorel, J., Mohri, M.: Multi-armed bandit algorithms and empirical evaluation. Gama, J., Camacho, R., Brazdil, P.B., Jorge, A.M., Torgo, L. (eds.) Machine Learning: ECML 2005, pp. 437–448. Springer, Berlin, Heidelberg (2005)

Virouleau, P., Broquedis, F., Gautier, T., Rastello, F.: Using data dependencies to improve task-based scheduling strategies on numa architectures. Dutot, P.-F., Trystram, D. (eds.) Euro-Par 2016: Parallel Processing, pp. 531–544. Springer, Cham (2016)

Wang, Y., Jiang, D., Xiong, J.: Numa-aware thread migration for high performance nvmm file systems

Wang, Z., Liu, X., Yang, J., Michailidis, T., Swanson, S., Zhao, J.: Characterizing and modeling non-volatile memory systems. 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 496–508 (2020). IEEE

Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., Swanson, S.: An Empirical Guide to the Behavior and Use of Scalable Persistent Memory (2019)

Yang, S., Li, X., Dou, X., Gong, X., Liu, H., Chen, L., Liu, L.: Monitoring memory behaviors and mitigating numa drawbacks on tiered nvm systems

Yu, S., Park, S., Baek, W.: Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. Proceedings of the International Conference on Supercomputing, pp. 1–10 (2017)