AccMER: Accelerating Multi-Agent Experience Replay with Cache Locality-aware Prioritization

Kailash Gogineni*
The George Washington University
Washington, DC, USA
kailashg26@gwu.edu

Yongsheng Mei*
The George Washington University
Washington, DC, USA
ysmei@gwu.edu

Tian Lan
The George Washington University
Washington, DC, USA
tlan@gwu.edu

Peng Wei
The George Washington University
Washington, DC, USA
pwei@gwu.edu

Guru Venkataramani
The George Washington University
Washington, DC, USA
guruv@gwu.edu

***Abstract*—Multi-Agent Experience Replay (MER) is a key component of off-policy reinforcement learning (RL) algorithms. By remembering and reusing experiences from the past, experience replay significantly improves the stability of RL algorithms and their learning efficiency. In many scenarios, multiple agents interact in a shared environment during online training under centralized training and decentralized execution (CTDE) paradigm. Current multi-agent reinforcement learning (MARL) algorithms consider experience replay with uniform sampling or based on priority weights to improve transition data sample efficiency in the sampling phase. However, moving transition data histories for each agent through the processor memory hierarchy is a performance limiter. Also, as the agents' transitions continuously renew every iteration, the finite cache capacity results in increased cache misses.**

To this end, we propose AccMER, that repeatedly reuses the transitions (experiences) for a window of n steps in order to improve the cache locality and minimize the transition data movement, instead of sampling new transitions at each step. Specifically, our optimization uses priority weights to select the transitions so that only high-priority transitions will be reused frequently, thereby improving the cache performance. Our experimental results on the Predator-Prey environment demonstrate the effectiveness of reusing the essential transitions based on the priority weights, where we observe an end-to-end training time reduction of 25.4% (for 32 agents) compared to existing prioritized MER algorithms without notable degradation in the mean reward.

***Index Terms*—Multi-Agent Systems, Performance Optimization, Experience Replay Buffer, Reinforcement Learning, Hardware**

I. Introduction

Reinforcement Learning (RL) has been applied to solve many single-agent sequential decision-making problems [1]. RL frameworks optimize the control of agent behavior and its interactions with the environment by taking actions based on current observation/state space, assessing the quality of state-action pairs using a reward function, and then transitioning to a new state [1]. The function that determines the action is known as a policy. The agent strives to find the optimal policy to maximize the total cumulative (discounted) reward. The function representing the reward estimates is known as the value function.

Often times in practice, RL tasks involve multiple agents sharing the same environment, e.g., autonomous driving [2], [3], robotics and planning [4], [5], and aviation systems [6]. Multi-agent reinforcement learning (MARL) [1] helps to coordinate the decision-making among multiple agents and learn the desired joint behavior from collective experiences (transition data) and achieve their goals. In particular, joint actions among these agents could affect the environment dynamically. The transitions observed in the environment are usually stored as experience tuples in a memory replay buffer and repeatedly used to improve the sample efficiency and policy training. This phase in the MARL training is called *mini-batch sampling*. We note that the mini-batch sampling is compute-intensive in multi-agent systems, with each agent collecting a significant number of experience tuples of all other agents in every iteration to share information amongst the agents for collective decision making [7]. Consequently, the computational and memory bandwidth demands also increase exponentially with the number of agents, which limits the applicability of MARL in real-world decision-making situations [8].

Prior studies on experience replay buffers in RL have proposed various strategies to improve the transition data sampling efficiency. The simplest and most widely used experience replay method is uniform sampling, where the transition data stored in the replay buffer are sampled uniformly at random [9]. However, uniform sampling might often select unimportant transitions and slow down the learning efficiency. For this reason, prioritized experience replay (PER) [10] and its variants were introduced [11], [12]. However, most of these prior efforts focus on prioritization methods for the experience replay in single-agent settings, and they cannot be adapted readily to MARL scenarios.

Recent work [13] on collective priority optimization in Multi-Agent Experience Replay (MER) showed rigorous theoretical analysis in assigning optimal sampling weights to

*These authors contributed equally to this work.

DOI 10.1109/ASAP57973.2023.00041

achieve higher mean rewards. MAC-PO prioritization technique achieves better convergence than the existing multi-agent learning algorithms. Our preliminary experiments (Section II) show that this implementation would still be computationally expensive as MAC-PO has to move the transition data histories and update its transitions in every iteration, resulting in cache and memory bandwidth bottlenecks. Thus, realizing efficient MARL algorithms with prioritization schemes from the systems perspective is still an open research problem.

In this paper, we propose AccMER, a cache-aware transition data reuse strategy to improve the MER efficiency for MARL algorithms. Specifically, we design transition data-reuse optimization that improves the cache locality by efficiently reusing higher-priority transitions during the MARL training phase. To the best of our knowledge, this is the *first* work to focus on improving the end-to-end training time of cooperative MARL algorithms. We validate the effectiveness of AccMER on the Predator-Prey environment [14] through comparison with baseline MARL settings (QMIX, WQMIX [15], and QPLEX), decomposed policy gradient method (i.e., VDAC [16]). In our experiments, AccMER achieves an end-to-end training time reduction of 25.4% (for 32 agents) compared to MAC-PO without any significant degradation in the mean reward.

The main contributions of our paper are the following:

- We present AccMER, an experience data-reuse strategy that can be used in conjunction with MER to address MARL performance bottlenecks. In particular, we use experience prioritization to reuse high-priority transitions for future sampling.
- We adopt a hardware-software co-design approach where a cache-aware transition data-reuse strategy significantly reduces the last-level cache misses while ensuring the convergence levels to be on par with the best-performing state-of-the-art multi-agent algorithms.
- Our experimental results on the Nvidia Ampere systems demonstrate that: 1) AccMER reduces the end-to-end training time by about 25.4% (for 32 agents) on a cooperative multi-agent setting (Predator-Prey task with no punishment); and 2) Interestingly, as the number of agents increases, our optimization efforts lead to a better convergence while reducing the training time on the same hardware compared to the current multi-agent prioritized experience replay [13].

II. Motivation

In this section we motivate the need to understand the performance bottlenecks in MARL algorithms, where we profile various state-of-the-art MARL frameworks implemented using actor-critic methods with usually very large state spaces.

We characterize the training phases of various representative MARL algorithms, including MADDPG [9], MATD3 [17], and MASAC [17] using Predator-Prey environment [9]. These require multi-agent settings and learn the desired joint behavior from the collective experiences. These approaches employ memory replay buffers with a uniform sampling of transition history.

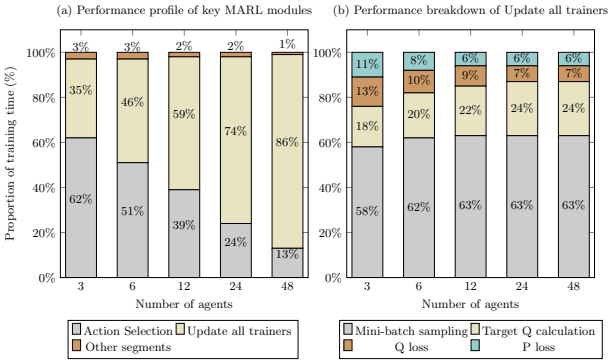


Fig. 1: Training time breakdown of various functions averaged acorss several MARL workloads under multi-agent settings on Ampere Architecture RTX 3090. The simulated multi-agent particle environment is Predator-Prey [9].

Figure 1a illustrates how the proportion of training time changes when the number of agents increases linearly on a CPU-GPU platform. We note that the *Update all trainers* phase contributes to ≈35% to ≈86% of the training time as the number of MARL agents grow from 3 to 48. The *Action selection* phase involves individual agents' policy networks using local observations to interact with the environment and this phase scales linearly. During *Update all trainers* phase, the actor network is updated using Q-values computed by the critic [9]. The target networks are created to achieve training stability. Note that the updating sequence of networks in the *Update all trainers* phase begin with that of critics, followed by the actors, and then the target networks. *Other segments* is a combination of reward collection, storing the present experiences and policy initialization and put together, they add a negligible performance overhead.

For a deeper analysis, as shown in Figure 1b, we divide the *Update all trainers* into multiple modules: *Mini-batch sampling, Target Q calculation*, and *Q loss & P loss* and present our results in the Predator-Prey environment. We observe that mini-batch sampling phase dominates the *Update all trainers* as it occupies 60% of training time in the update phase. This is because, each agent has to randomly sample a mini-batch of transition data (history) of all other agents from the replay buffer to update both the critic and actor networks. In real-world systems, this task may be lead to huge compute and memory requirements as the number of agents increase.

We also perform experiments to understand how the cache performance is affected as a consequence of batch size vs. buffer size trade-offs on the QMIX algorithm [18] on account of the random memory access patterns. From Table I, when the batch size increases from 64 to 256 in the difficulty-enhanced predator-prey environment, the LLC load misses increase to 120% in QMIX (for 8 agents). This is because of the uniform sampling, where the collected transitions will be continually renewed at every step, and finite cache capacity results in increased cache misses. As a result, conventional sampling

is impractical and may lead to computing bottlenecks in real-world systems, especially when the number of agents scale under MARL. Furthermore, even after MER prioritization is enabled, the global cache misses grow by $2.5\times$ when the number of agents scale from 16 to 32 in MAC-PO [13]. The dynamic memory requirements of observation and action spaces also grow quadratically due to each agent coordinating with other agents toward sharing their observations and actions [7], [19].

TABLE I: Cache miss profiles in the QMIX algorithm for different mini-batch and experience replay buffer sizes

Buffer size	Batch size	LLC load misses	global cache misses
100,000	256	10,228,039,764	30,727,770,917
10,000	128	7,253,378,442	24,608,100,772
1000	64	4,695,948,584	17,722,615,186
100	16	2,247,388,137	9,428,462,486

III. Background

In this work, we consider a multi-agent sequential decision-making task as a decentralized partially observable Markov decision process (Dec-POMDP) [20] consisting of a tuple $G = \langle S, U, P, R, Z, O, n, \gamma \rangle$, where $s \in S$ describes the global state of the environment. At each time-step, each agent $a \in A \equiv \{1, \ldots, n\}$ selects an action $u_a \in U$, and all selected actions combine and form a joint action $\mathbf{u} \in \mathbf{U} \equiv U^n$. Such a process leads to a transition in the environment based on the state transition function $P(s'|s, \mathbf{u}) : S \times \mathbf{U} \times S \to [0, 1]$. All agents share the same reward function $r(s, \mathbf{u}) : S \times \mathbf{U} \to \mathbb{R}$ with a discount factor $\gamma \in [0, 1)$.

In the partially observable environment, the agents' individual observations $z \in Z$ are generated by the observation function $O(s, u) : S \times A \to Z$. Each agent has an action-observation history $\tau_a \in T \equiv (Z \times U)^*$. Conditioning on the history, the policy becomes $\pi^a(u_a|\tau_a) : T \times U \to [0, 1]$. The joint policy $\boldsymbol{\pi}$ has a joint action-value function: $Q^\pi(s_t, \mathbf{u}_t) = \mathbb{E}_{s_{t+1:\infty}, \mathbf{u}_{t+1:\infty}}[R_t|s_t, \mathbf{u}_t]$, where t is the timestep and $R_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$ is the discounted return. The learning algorithm has access to all local action-observation histories $\boldsymbol{\tau}$ and global state s during training, yet every agent can only access its individual history in execution. The learning algorithm we use in this work is an actor-critic method, MAC-PO, a multi-agent prioritized experience replay variant of QMIX.

In QMIX [18], the learner is designed for multi-agent cooperative tasks with global reward. Specifically, QMIX solves credit assignment problems using additional information during training. The core idea of QMIX is value decomposition. A parameterized mixer function is proposed to combine the Q-functions of agents into a centralized Q-function that is trained on the global reward. Further, QMIX demonstrates that the mixer network provides monotonicity in its inputs, which makes the argmax of the agents' Q-functions consistent with the argmax of the centralized Q-function. This property is called Individual-Global-Max (IGM), and it is important for factorizing the global Q-function to agents' Q-functions.

IV. Methodology

AccMER trains with the cache-aware transition data-reuse optimization on top of the MER prioritization scheme to improve the MARL performance. AccMER aims to reduce the number of last-level cache misses by reusing transition data, that ultimately improves the training time. The rest of this section delves deeper into the MER prioritization scheme and the transition data-reuse optimization of AccMER's design.

A. Prioritization Optimization for Experience Replay

Recent work [21] shows that the design of prioritized sampling methods influences the loss function. On the contrary, the expected gradient of a loss function with non-uniform sampling is equivalent to that of a weighted loss function with uniform sampling, which provides a recipe for transforming a regular loss function L_1 with a non-uniform sampling scheme into an equivalent weighted loss function L_2 with uniform sampling. Based on this equivalence, MAC-PO [13] further explores the optimal weighting scheme for prioritized experience in MARL, given by a weighting factor $w_k(s, \mathbf{u})$ when computing the loss function. In this paper, we adopt this MER prioritization scheme as the baseline while investigating the hardware-aware optimization that can improve learning efficiency regarding the overall training time and cache usage. Given the prioritization weight w_k, we use the following loss function during the learning, which is:

$$L_{\text{AccMER}} = \sum_{i=1}^{b} w_k(s, \mathbf{u})(Q_k - y_i)^2(s, \mathbf{u}), \tag{1}$$

where b is the batch size. In the loss function (1), $y_i = \mathcal{B}^* Q_{k-1}$ denotes a fixed target that can be obtained through a target network, where $\mathcal{B}^*$ is the Bellman operator satisfying $\mathcal{B}^* Q(s, \mathbf{u}) \stackrel{\text{def}}{=} r(s, \mathbf{u}) + \gamma \arg\max_{\mathbf{u}'} \mathbb{E}_{s'} Q(s', \mathbf{u}')$. The following lemma for deciding optimal weights is proposed in [13].

Lemma 1 (Optimal prioritization weight)**.** *The optimal weight in* (1) *is proportional to:*

$$w_k(s, \mathbf{u}) \propto |Q_k - \mathcal{B}^* Q_{k-1}| \exp(-|Q_k - Q^*|) f(\pi_k^a), \tag{2}$$

where Q^* denotes the optimal action value function and the function $f(\cdot)$ is defined as:

$$f(\pi_k^a) \stackrel{\text{def}}{=} 1 + \sum_{i=1}^{n} \prod_{\substack{j=1 \\ j \neq i}}^{n} \pi_k^j - n \prod_{i=1}^{n} \pi_k^i. \tag{3}$$

The optimal weight in (2) consists of three main terms, which are Bellman error term $|Q_k - \mathcal{B}^* Q_{k-1}|$, value enhancement term $\exp(-|Q_k - Q^*|)$, and joint action probability function $f(\pi_k^a)$. The Bellman error term measures the distance between the estimation of the action value function and the Bellman target, in which the significant difference means higher hindsight Bellman error and will lead to higher sampling weight assignment. The value enhancement term indicates that any transitions with more accurate action values compared to the optimal value estimation after the Bellman

update should be assigned with higher weights. Considering the relationship between agents' individual policies, the joint action probability function shows the counter-intuitive fact that the higher weights will be assigned to transitions with one's action differentiated from the others, as the maxima of function (3) can be reached if and only if one agent's action probability is small in the transition while all other agents' action probabilities are large. We adopt this optimal weighting scheme in our work with the necessary normalization. To thoroughly exploit transitions with higher weights, we propose AccMER, that leverages data reuse strategy to efficiently remember and recapture the highly-weighed transitions in the replay buffer and utilize them for a specific number of steps for more efficient and performance-wise better learning.

Algorithm 1 AccMER

1: Initialize step t, experience replay buffer $\mathcal{D}$ with size d, mini-batch size b, reuse ratio α, and weights
2: **for** $t = 1 : t_{max}$ **do**
3: Initialize the state (observation vetor)
4: **while** goal state is not reached **do**
5: for each agent a, select action u_a w.r.t. the policy π^a
6: Compute the reward and next state
7: Store the current trajectory (current state, next state, rewards, action) into replay buffer $\mathcal{D}$
8: **end while**
9: for every $\lfloor d/b \rfloor$ steps, select the $\mathcal{S}^-$ = $\{\alpha \cdot b\}$ transitions from $\mathcal{D}$ ranked based on the optimal weights
10: Sample $(1-\alpha)\cdot b$ transitions as $\mathcal{S}^+$ from the complement of $\mathcal{S}^-$ in $\mathcal{D}$ following the uniform distribution
11: Update the mini-batch $\mathcal{S} = \mathcal{S}^- \cup \mathcal{S}^+$
12: **for** each time-step k in $\mathcal{S}$ **do**
13: Compute the optimal weights w_k according to the prioritization scheme in Lemma 1
14: Update the weights for transitions
15: **end for**
16: Update the network parameters
17: **end for**

B. Cache-aware transition data-reuse optimization

From our analysis, we note that the sampling phase is one of the compute-intensive phases, as each agent has to sample all other agents' transition data sequentially. Figure 2 shows an example layout of conventional sampling and the priority-guided transition data-reuse optimization.

Uniform sampling suffers from random memory access patterns, often leading to low cache line utilization, meaning the arrays' transitions are indexed randomly. Because of this, the number of cache misses and memory bandwidth demands of the program increase with the number of agents.

To perform the cache-aware transition data-reuse optimization, AccMER first partitions two different micro-batches according to the mini-batch size b as shown in Algorithm 1. We initialize a weight lookup table $\mathcal{W}$ mapping to transitions data addresses in the replay buffer $\mathcal{D}$. Both lookup table $\mathcal{W}$ and replay buffer $\mathcal{D}$ have the size of d. The initial weights for the weight lookup table will be the same for all transitions. Depending on the reuse ratio $\alpha \in [0, 1]$ (if α is 0, all the transitions are sampled uniformly, where as if α is 1, all the transitions will be reused), the micro-batch $\mathcal{S}^-$ ranks and selects the transitions data according to the weight lookup table $\mathcal{W}$ from the replay buffer $\mathcal{D}$ (line 9). For every $\lfloor d/b \rfloor$ steps, we reuse the same transitions according to the priority weights. In this way, we map the addresses of the weight lookup table $\mathcal{W}$ and transitions in the replay buffer $\mathcal{D}$ to choose high-priority transitions. Unlike conventional sampling, which might select random transitions and often select unimportant transitions, our optimization leverages priority-guided transition data-reuse to improve the data availability in upper-level caches for better memory locality.

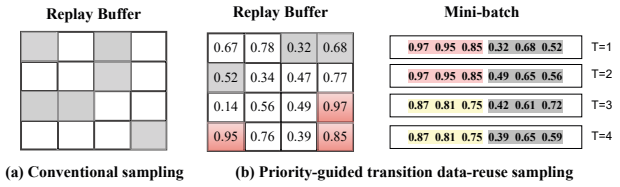


Fig. 2: Illustration of (a) conventional sampling, where gray filled boxes denote the uniformly sampled transitions from the replay buffer, and (b) data reuse sampling, we sample 50% and reuse the rest of the transitions (reuse ratio = 0.5, batch size $b = 6$, replay buffer $\mathcal{D}$ is a $4 * 4$ matrix) and the number of reuses (time-steps) n is computed by $\lfloor d/b \rfloor = 2$. At step $T = 1$, as the data reuse ratio is 0.5, AccMER selects the first three transitions with the highest weights, and in the next phase, uniform sampling is performed on the remaining three transitions. At step $T = 2$, the same transition data will be reused, whereas a new set of transitions is sampled randomly. Since $n = 2$, for $T = 3$ and $T = 4$, the reuse-based transition data updates every two steps.

Note that reuse ratio, α for the transition data partition is input by the user. We sample the remaining transitions from the complement of $\mathcal{S}^-$ in replay buffer $\mathcal{D}$ following the uniform distribution and store it in $\mathcal{S}^+$ (line 10). In the next step, we concatenate the $\mathcal{S}^-$ and $\mathcal{S}^+$ and update the batch $\mathcal{S}$ (line 11). By Lemma 1, we calculate the optimal prioritization weights for the transitions, and then we update the weight lookup table $\mathcal{W}$ with the optimal weight w_k. In order to avoid the bias towards reusing the same transitions, we apply a discount factor γ to all the weights in the weight lookup table $\mathcal{W}$. To maximize the reuse of transitions with high priority, we sort the weight lookup table $\mathcal{W}$ so that high-priority transitions will be reused for multiple steps.

V. Experimental Evaluation

In this section, we evaluate the performance of AccMER using difficulty-enhanced Predator-Prey task [13] at various punishment levels. Section V-A introduces the implementation details and hyper-parameters that we used in the experimental

evaluation. In Section V-B, we demonstrate the effectiveness of AccMER by comparing the mean reward with several state-of-the-art MARL baselines. In Section V-B, we show that AccMER is able to improve the end-to-end training time and reduce a significant number of last-level cache misses with no significant loss in the mean episode reward. Finally, in Section V-D, we conduct the additional scalability tests to demonstrate the benefits of AccMER[1].

A. Evaluation Setup

For evaluation, we implemented the key components of AccMER: a) Optimal prioritization [13], and b) Cache-aware transition data-reuse optimization on the baseline QMIX code base. We use epsilon greedy for action selection with annealing from ϵ = 0.995 decreasing to ϵ = 0.05 in 100K training steps in a linear way [1], [13]. The performance for each algorithm is evaluated for 32 episodes every 1000 training steps.

TABLE II: Hyper-parameters for the Predator-Prey task with no punishment.

Hyper-parameter	Value
Batch size	256
Replay buffer size	100000
Target network update interval	Every 200 episodes
Learning rate	0.001
TD-lambda	0.6

Target Platform: We train and profile AccMER on the Nvidia GeForce RTX 2080TI architecture connected with Intel(R) Core(TM) i9-7920X CPU, which has 12 cores with 16 MiB of Last-Level Cache, 128 GiB Gigabytes of main memory and the CPU's clock speed of 2.90GHz. The server runs on Ubuntu Linux 20.04.5 LTS operating system with CUDA 11.3, cuDNN 8.2, PCIe Express® v3.0 with NCCL v2.5.7 communication library. The machine supports python 3.8.16, PyTorch (v1.8.2), pyyaml (v5.3.1) and OpenAI GYM (v0.11). We use Perf [22] tool to evaluate the hardware efficiency.

TABLE III: Hyper-parameters for the Predator-Prey task with punishment= -1.5.

Hyper-parameter	Value
Batch size	128
Replay buffer size	10000
Target network update interval	Every 200 episodes
Learning rate	0.001
TD-lambda	0.6

Multi-agent environment: A partially observable environment on a grid-world Predator-Prey task [13] where 8 agents have to catch 8 prey in a 10 × 10 grid. Each agent can either move in one of the 4 compass directions, remain still, or try to catch any adjacent prey. In this task, a successful capture with the positive reward of 1 must include two or more predator agents surrounding and catching the same prey simultaneously, requiring a high level of cooperation. A failed coordination between agents to capture the prey, which happens when only one predator catches the prey, will receive a negative punishment reward. We select the punishments of 0 and -1.5 in the experiments, with more punishment representing higher difficulty.

[1]https://github.com/kailashg26/AccMER

B. Comparison with MARL baselines

We select multiple state-of-the-art MARL algorithms for comparison, which include value-based factorization MARL algorithm (i.e., QMIX [18], WQMIX [15], and QPLEX), decomposed policy gradient method (i.e., VDAC [16]), and decomposed actor-critic approaches (i.e., FOP [23] and DOP [24]). All of the baselines have demonstrated their convergence properties in handling various multi-agent tasks.

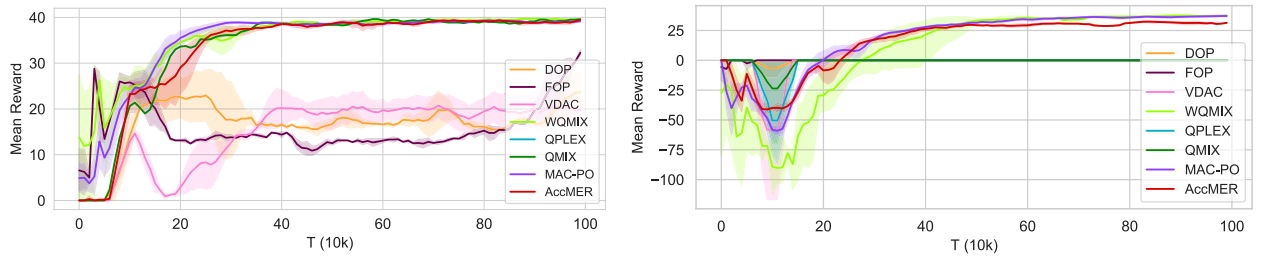
We validate the efficacy of AccMER on Predator-Prey environment for two punishment levels: 0 and -1.5, and the hyper-parameters are shown in Table II and Table III. The transition data-reuse ratio α for all the experiments is 0.5, and the batch size b for Predator-Prey: 0 and -1.5 is 256 and 128, respectively. Additionally, we set the discount factor γ as 0.8 for the Predator-Prey: -1.5 hard settings and γ as 1.0 for Predator-Prey (no punishment).

Figure 3 shows the performance of eight algorithms with different punishments, where all results show the effectiveness of AccMER. We note that AccMER's convergence levels are on par with best-performing state-of-the-art MARL algorithms in finding the optimal policy. In Figure 3b, AccMER significantly outperforms other state-of-the-art algorithms like QMIX and WQMIX and performs on par with MAC-PO in a hard setting that requires a higher level of coordination among agents in order to learn optimal policy. Most of the MARL algorithms learn a sub-optimal policy where agents learn to work together with limited coordination. Although the algorithmic performance (reward) of AccMER and MAC-PO are almost similar, compared to the latter, Figure 4 shows that AccMER achieves a performance speedup by reducing the training time by about 17% for Predator-Prey (no punishment) task compared to MAC-PO. This demonstrates that efficiently recapturing the prioritized transitions with higher weights and smart cache data-reuse strategies, AccMER can learn the optimal policy and improve the training efficiency.

C. Impact of AccMER

We perform experiments with a transition data-reuse ratio, $\alpha = 0.5$, meaning 50% of the prioritized transitions are being reused between the iterations.

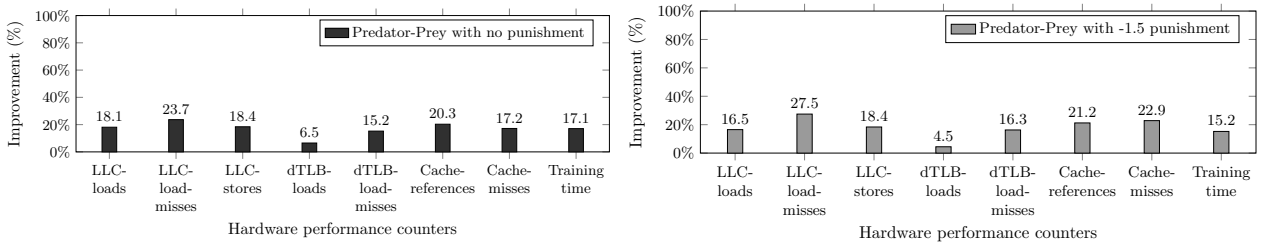
Figure 4 shows the training time savings with respect to the wall-clock training time and the profiles of certain key hardware performance counters. Specifically, in a Predator-Prey environment with no punishment, compared to MAC-PO, AccMER reduces the end-to-end training time by about 17%, which is almost $1.2\times$ faster than MAC-PO (Figure 4a) and there is no noticeable degradation in the mean episode reward. In fact, by reusing prioritized transition data, we reduced the LLC-load misses by about 23.7% and the global cache misses by about 17.2%. As the batch size is 128 for the



(a) No punishment

(b) Punishment $= -1.5$

Fig. 3: Average reward per episode on the Predator-Prey tasks for AccMER and other MARL algorithms under different punishment levels. AccMER shows almost the same convergence speed as MAC-PO while reducing the total training time.



(a) No punishment

(b) Punishment $= -1.5$

Fig. 4: Hardware performance analysis of AccMER for Predator-Prey environment with punishment= 0 and punishment= -1.5.

hard setting (Predator-Prey with punishment=-1.5), AccMER shows 15.2% improvement in training time, compared to the 17.1% when the batch size is 256. This indicates that larger batch sizes and replay buffers can boost further performance gains (Figure 4b). Interestingly, when a high level of coordination is required among the agents, data-reuse optimization demonstrates *higher* effectiveness by reducing 27.5% LLC load misses, and 16.3% dTLB load misses compared to MAC-PO. These experimental results confirm that priority-guided transition data-reuse is highly effective in multi-agent scenarios. We further note that excluding the environment interactions phase from the MARL training time will give further speedups, as the environment interactions grow dramatically when more agents are involved in the MARL training phase.

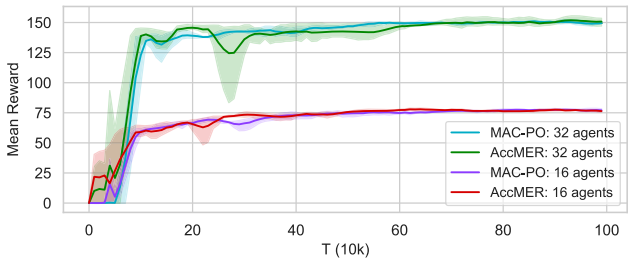


Fig. 5: Average reward per episode on the Predator-Prey task for AccMER and MAC-PO algorithms with no punishment.

D. Scalability Tests

We conduct the scalability tests and profile AccMER on the Nvidia GeForce RTX 3090 Ampere Architecture connected with AMD Ryzen Threadripper PRO 3975WX CPU, which has 32 cores with 128 MiB of Last-Level Cache, 512 Gigabytes of main memory and the CPU's clock speed of 3.5GHz. The server runs on Ubuntu Linux 20.04.5 LTS OS with PCIe Express® v4.0 and NCCL v2.8.4 communication library.

We use the following hyper-parameters for the scalability tests: The transition data-reuse ratio α is 0.5, and the batch size b for Predator-Prey: 0 is set to 128, with $10,000$ as the buffer size. Additionally, we set the discount factor γ as 0.8 for the Predator-Prey: 0 task.

Figure 5 shows the mean episode reward curves of AccMER and MAC-PO [13]. While we can see that when the number of agents increases linearly (from 16 to 32), AccMER shows a slightly faster convergence than MAC-PO. Moreover, since AccMER reuses the prioritized transition data for multiple future steps, it reduces the end-to-end training time by about 19.8% and 25.4% for 16 and 32 agents, respectively (Figure 6). By scaling the number of agents, we observe that the global cache misses gradually improve (18.4% improvement in cache misses for 32 agents over the baseline), which indicates that AccMER can achieve higher speedups and improve the hardware efficiency for large-scale cooperative MARL settings.

VI. Related Work

Prior works have demonstrated how experience replay and its variants can achieve better convergence for RL workloads. However, to our knowledge, no prior research presents insights into end-to-end performance improvement that involves multiple agents from the systems perspective. We provide an overview of related efforts in hardware-software acceleration and experience replay in RL.

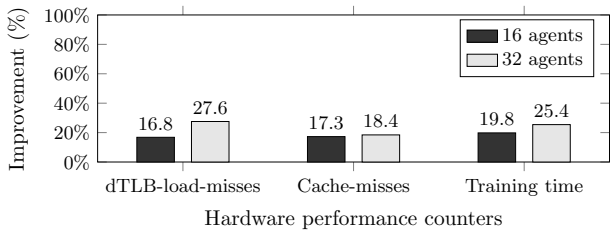


Fig. 6: Performance analysis of AccMER for Predator-Prey task with no punishment in multi-agent settings.

A. Quantization

Low-precision training for neural networks reduces the neural network weights, enables faster compute operations, and minimizes the memory transfer computation time. Quantization aware training [25], post-quantization training [26], and mixed precision [27] demonstrated that neural networks may be quantized to a lower precision without significant degradation in rewards. Furthermore, to speedup the training, prior works have proposed algorithmic modifications (e.g., compound loss scaling, storing the hypotenuse in Adam, etc.) that leaves the underlying agent and its hyper-parameters unchanged but improves the numerical stability and reduces the memory and compute requirements [28]. QuaRL [29] demonstrates how to accelerate single-agent RL, where quantization is applied to speedup the RL training and inference. All the prior works differ from our work as they apply quantization to single-agent RL algorithms or neural networks. In contrast, we seek to improve the performance of MARL by reusing the prioritized transitions for a certain number of time-steps.

B. Software-Hardware Acceleration for RL

Distributed training has been widely adopted to reduce the training time of single-agent RL algorithms [30]–[35]. Another strategy for MARL acceleration is to show the training efficiency via theoretical analysis by restricting the agent interactions to one-hop neighborhoods and adopting a distributed training strategy to simulate the state transitions of only a small subset of agents on each compute node [36]. However, training on VM-based approaches still requires extensive management of the cluster and deploying the training jobs. FA3C [31] studies how to accelerate multiple parallel worker scenarios, where each agent is controlled independently within their own environments using single-agent RL algorithms. iSwitch [32] reduces the end-to-end network latency for synchronous training, but also improves the convergence with faster weight updates for asynchronous training. In contrast, our work focuses on multi-agent learning frameworks, where the agents operate in a single shared environment.

C. Experience Replay Buffer

Many RL algorithms adopt prioritization to increase the learning efficiency, initially originating from prioritized sweeping for value iteration [37], [38]. Prioritized experience replay (PER) [39] is one of the key advancements in the DQN algorithm [40], [41] and has been included in many RL algorithms combining multiple improvements [42], [43]. Variants of PER have been proposed for considering sequences of transitions [11], [44] or optimizing the prioritization function [45]. Discor re-weights updates to reduce variance [46]. ReMERN uses the regret minimization method to design the prioritized experience replay scheme in the single-agent environment [12]. So far, most prior works about experience replay are designed for single-agent RL algorithms.

A recently proposed multi-agent experience replay framework, MAC-PO [13], can find the optimal prioritized sampling scheme by computing the optimal sampling weights for experience replay when the environment involves multiple agents. We adopt MAC-PO as one of the baselines in our studies. However, different from MAC-PO, this paper seeks to improve the actual run-time (system) performance of the multi-agent experience replay prioritization, such as training efficiency and cache utilization. Specifically, AccMER adopts the MER prioritization scheme in MAC-PO and repeatedly reuses the selected transitions with high prioritization weights, achieving performance speedup while retaining the MAC-PO's learning performance advantages. Apart from methods that focus on improving the training time of cooperative problems, other mechanisms use a *neighbor sampling strategy* to improve the locality and training efficiency of competitive tasks [47].

VII. Conclusion

We presented AccMER, a cache-aware transition data reuse strategy for multi-agent experience replay. Our experimental results demonstrate that AccMER reduces the overall training time by about 25.4% (for 32 agents) over prior multi-agent prioritized replay schemes. Additionally, we show that the proposed data reuse optimization alleviates the performance issues posed by random memory access patterns by optimizing the transition data sampling phase for better hardware cache locality, thereby improving the overall MARL performance.

Acknowledgment

This research is based on work supported by the National Science Foundation under grant CCF-2114415.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Y. Cao, W. Yu, W. Ren, and G. Chen, "An overview of recent progress in the study of distributed multi-agent coordination," *IEEE Transactions on Industrial informatics*, vol. 9, no. 1, pp. 427–438, 2012.

[3] Y. Hu, A. Nakhaei, M. Tomizuka, and K. Fujimura, "Interaction-aware decision making with adaptive strategies under merging scenarios," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019, pp. 151–158.

[4] L. Matignon, L. Jeanpierre, and A.-I. Mouaddib, "Coordinated multi-robot exploration under communication constraints using decentralized markov decision processes," in *Twenty-sixth AAAI conference on artificial intelligence*, 2012.

[5] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1334–1373, 2016.

[6] P. Razzaghi, A. Tabrizian, W. Guo, S. Chen, A. Taye, E. Thompson, A. Bregeon, A. Baheri, and P. Wei, "A survey on reinforcement learning in aviation applications," *arXiv preprint arXiv:2211.02147*, 2022.

[7] K. Gogineni, P. Wei, T. Lan, and G. Venkataramani, "Scalability Bottlenecks in Multi-Agent Reinforcement Learning Systems," *arXiv preprint arXiv:2302.05007*, 2023.

[8] S. Iqbal and F. Sha, "Actor-attention-critic for multi-agent reinforcement learning," in *International conference on machine learning*. PMLR, 2019, pp. 2961–2970.

[9] R. Lowe, Y. I. Wu, A. Tamar, J. Harb, O. Pieter Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *Advances in neural information processing systems*, vol. 30, 2017.

[10] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[11] M. Brittain, J. Bertram, X. Yang, and P. Wei, "Prioritized sequence experience replay," *arXiv preprint arXiv:1905.12726*, 2019.

[12] X.-H. Liu, Z. Xue, J. Pang, S. Jiang, F. Xu, and Y. Yu, "Regret minimization experience replay in off-policy reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 17 604–17 615, 2021.

[13] Y. Mei, H. Zhou, T. Lan, G. Venkataramani, and P. Wei, "MAC-PO: Multi-agent experience replay via collective priority optimization," *arXiv preprint arXiv:2302.10418*, 2023.

[14] W. Böhmer, V. Kurin, and S. Whiteson, "Deep coordination graphs," in *International Conference on Machine Learning*. PMLR, 2020, pp. 980–991.

[15] T. Rashid, G. Farquhar, B. Peng, and S. Whiteson, "Weighted qmix: Expanding monotonic value function factorisation for deep multi-agent reinforcement learning," 2020.

[16] J. Su, S. Adams, and P. Beling, "Value-decomposition multi-agent actor-critics," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 13, 2021, pp. 11 352–11 360.

[17] J. Ackermann, V. Gabler, T. Osa, and M. Sugiyama, "Reducing overestimation bias in multi-agent domains using double centralized critics," *NeurIPS Deep Reinforcement Learning Workshop*, 2019.

[18] T. Rashid, M. Samvelyan, C. Schroeder, G. Farquhar, J. Foerster, and S. Whiteson, "Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2018, pp. 4295–4304.

[19] H. U. Sheikh and L. Bölöni, "Multi-agent reinforcement learning for problems with combined individual and team reward," in *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2020, pp. 1–8.

[20] F. A. Oliehoek and C. Amato, *A concise introduction to decentralized POMDPs*. Springer, 2016.

[21] S. Fujimoto, D. Meger, and D. Precup, "An equivalence between loss functions and non-uniform sampling in experience replay," *Advances in neural information processing systems*, vol. 33, pp. 14 219–14 230, 2020.

[22] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[23] T. Zhang, Y. Li, C. Wang, G. Xie, and Z. Lu, "Fop: Factorizing optimal joint policy of maximum-entropy multi-agent reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2021, pp. 12 491–12 500.

[24] Y. Wang, B. Han, T. Wang, H. Dong, and C. Zhang, "Dop: Off-policy multi-agent decomposed policy gradients," in *International Conference on Learning Representations*, 2020.

[25] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[26] T. Tambe, E.-Y. Yang, Z. Wan, Y. Deng, V. J. Reddi, A. Rush, D. Brooks, and G.-Y. Wei, "Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[27] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[28] J. Björck, X. Chen, C. De Sa, C. P. Gomes, and K. Weinberger, "Low-precision reinforcement learning: running soft actor-critic in half precision," in *International Conference on Machine Learning*. PMLR, 2021, pp. 980–991.

[29] S. Krishnan, M. Lam, S. Chitlangia, Z. Wan, G. Barth-Maron, A. Faust, and V. J. Reddi, "Quarl: Quantization for fast and environmentally sustainable reinforcement learning," 2022.

[30] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "GA3C: GPU-based A3C for deep reinforcement learning," *CoRR abs/1611.06256*, 2016.

[31] H. Cho, P. Oh, J. Park, W. Jung, and J. Lee, "Fa3c: Fpga-accelerated deep reinforcement learning," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 499–513.

[32] Y. Li, I.-J. Liu, Y. Yuan, D. Chen, A. Schwing, and J. Huang, "Accelerating distributed reinforcement learning with in-switch computing," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019, pp. 279–291.

[33] M. W. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, N. Momchev, D. Sinopalnikov, P. Stańczyk, S. Ramos, A. Raichuk, D. Vincent *et al.*, "Acme: A research framework for distributed reinforcement learning," *arXiv preprint arXiv:2006.00979*, 2020.

[34] A. Stooke and P. Abbeel, "Accelerated methods for deep reinforcement learning," *arXiv preprint arXiv:1803.02811*, 2018.

[35] A. V. Clemente, H. N. Castejón, and A. Chandra, "Efficient parallel methods for deep reinforcement learning," *arXiv preprint arXiv:1705.04862*, 2017.

[36] B. Wang, J. Xie, and N. Atanasov, "Darl1n: Distributed multi-agent reinforcement learning with one-hop neighbors," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2022, pp. 9003–9010.

[37] A. W. Moore and C. G. Atkeson, "Prioritized sweeping: Reinforcement learning with less data and less time," *Machine learning*, vol. 13, no. 1, pp. 103–130, 1993.

[38] H. Van Seijen and R. Sutton, "Planning by prioritized sweeping with small backups," in *International Conference on Machine Learning*. PMLR, 2013, pp. 361–369.

[39] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *ICLR*, 2016.

[40] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.

[41] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1995–2003.

[42] D. Horgan, J. Quan, D. Budden, G. Barth-Maron, M. Hessel, H. van Hasselt, and D. Silver, "Distributed prioritized experience replay," in *International Conference on Learning Representations*, 2018.

[43] G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, T. Dhruva, A. Muldal, N. Heess, and T. Lillicrap, "Distributed distributional deterministic policy gradients," in *International Conference on Learning Representations*, 2018.

[44] B. Daley and C. Amato, "Reconciling λ-returns with experience replay," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[45] D. Zha, K.-H. Lai, K. Zhou, and X. Hu, "Experience replay optimization," in *IJCAI*, 2019.

[46] A. Kumar, A. Gupta, and S. Levine, "Discor: Corrective feedback in reinforcement learning via distribution correction," *Advances in Neural Information Processing Systems*, vol. 33, pp. 18 560–18 572, 2020.

[47] K. Gogineni, P. Wei, T. Lan, and G. Venkataramani, "Towards efficient multi-agent learning systems," *arXiv preprint arXiv:2305.13411*, 2023.