Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching

Charles Block
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
coblock2@illinois.edu

Gerasimos Gerogiannis
University of Illinois at Urbana-Champaign
Urbana, Illinois, USA
gg24@illinois.edu

Josep Torrellas
University of Illinois Urbana-Champaign
Urbana, Illinois, USA
torrella@illinois.edu

Abstract

Online reinforcement learning (RL) holds promise for microarchitectural techniques like prefetching. Its ability to adapt to changing and previously-unseen scenarios makes it a versatile technique. However, when multiple RL-operated components compete for shared resources in multicore systems, they can often converge to sub-optimal policies due to conflicting incentives.

In this work, we identify key challenges that arise when scaling RL-based prefetchers to multi-core environments, and relate these to known problems from Multi-Agent Reinforcement Learning (MARL). In particular, we find that recent work using multi-armed bandit algorithms for prefetching can lead to inefficient systems when memory bandwidth is limited, as each agent attempts to claim a disproportionate share of the system's bandwidth.

To solve this problem, we present μMama, a light-weight supervisor of distributed multi-armed bandit agents, which learns performant joint-policies. In μMama, distributed local agents narrow the global joint-action search space, while a central agent with a global perspective learns system-wide policies. Additionally, μMama provides key local agents with a system perspective, encouraging them to avoid actions that would harm the others.

μMama exhibits high adaptability, which we show by evaluating it using multiple measures of performance. In our evaluation of an 8-core system, the policies learned by μMama outperform those of independently-operating agents by an average of 2.1% when optimizing for throughput, and by an average of 10.4% when optimizing for fairness. We also show that μMama performs better in systems that are more bandwidth constrained, as well as when profiles of the workloads are provided.

CCS Concepts

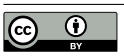
• **Computer systems organization → Multicore architectures**.

Keywords

Multi-Agent Reinforcement Learning, Microarchitecture, Machine Learning for Architecture, Prefetching

ACM Reference Format:

Charles Block, Gerasimos Gerogiannis, and Josep Torrellas. 2025. Micro-MAMA: Multi-Agent Reinforcement Learning for Multicore Prefetching. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756096



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756096

1 Introduction

Recently, machine learning (ML) methods for classic CPU microarchitecture techniques have been popular topics of research, as they promise new data-driven approaches to branch prediction [27, 59], data prefetching [11, 47], and more [10, 16, 45, 46]. Of particular interest are the works that use online reinforcement learning (RL) to learn solutions *at runtime* for problems ranging from prefetching [5, 17, 19, 35, 42] and cache replacement [35, 61] to branch prediction [62] and SMT thread management [17, 56]. Online RL promises to learn high-performing control policies for changing or unseen scenarios in real time. By learning from previous experiences, these algorithms are able to direct the actions of traditional microarchitectural elements that may otherwise rely solely on brittle hand-crafted heuristics.

Prior RL works have primarily considered systems of a single learner, such as a prefetcher for a single private cache [5, 17]. Yet multicore systems, in which several such learners may be located, introduce additional complexities. For example, an RL prefetcher may opt for an aggressive prefetch policy to increase its own reward, but this may harm the performance of other cores due to higher memory contention. Contention caused by prefetchers is a known problem [12, 13], but RL-based prefetchers often have conflicting incentives, *encouraging* them to increase memory contention, to the detriment of the whole system. Such behavior is well-known in the fields of Multi-Agent Reinforcement Learning (MARL) [2] and Game Theory [40].

To address this problem, we propose a lightweight system for coordinating a set of distributed RL-operated prefetchers in multicores. We start by identifying issues in the Micro-Armed Bandit prefetcher [17] that exacerbate competition between agents when operating in a multicore system. Based on the issues we find and prior theoretical work [39], we propose using an intelligent RL supervisor to coordinate distributed Micro-Armed Bandit agents, utilizing their local insights, while prioritizing actions that maximize the system's performance. We call this coordinator of microarchitectural Multi-Agent Multi-Armed bandits *μMama*.

Our simulations show the effectiveness of μMama. In eight-core μMama increases the system throughput by an average of 2.1% over uncoordinated Micro-Armed Bandit agents. On more bandwidth-constrained systems, μMama shows higher gains. In addition, μMama's RL-based design allows it to easily adapt to alternate optimization targets. For example, fairness is often critical in multi-tenant and latency-sensitive systems, such as cloud

environments. When provided with a reward that incorporates fairness, μMama improves the system's harmonic mean speedup by 10.4% compared to uncoordinated Micro-Armed Bandits. By simply changing the reward, μMama can be used to explore the fairness/throughput tradeoff of a system, increasing its flexibility and applicability.

Overall, the main contributions of the paper are:

- An analysis of issues faced by RL prefetchers in multicore systems and how they relate to known problems in MARL.
- The design of μMama: a flexible system architecture for coordinating contentious Micro-Armed Bandit agents.
- An evaluation of μMama on a range of system configurations, compared to multiple state-of-the-art prefetchers.
- A demonstration of μMama's configurability and how it can be modified for different targets, enabling flexible designs.

2 Background

In this section, we provide background on online RL. We discuss issues that arise when multiple online learners share a single environment.

2.1 Online Reinforcement Learning

RL [6, 55] is a type of ML where an *agent* takes various *actions* and receives *rewards* as feedback for those actions. The agent attempts to learn the correct actions to play in order to maximize its rewards. In *online* RL, the agent is thrust into its operating environment without prior training. An online RL agent must strike a balance between *exploring* its action space (to learn more about the impact of its actions on the environment) and *exploiting* what it has already learned (to maximize its rewards). There are numerous approaches to online RL, such as Q-learning [57], SARSA [43, 55], and policy gradient methods [55].

Some of the simplest approaches to RL are Multi-Armed Bandit algorithms [32]. In a Multi-Armed Bandit model, the agent's rewards are assumed to depend (maybe stochastically) only on the action (or "arm") that the agent chooses to excercise—not on some underlying system state. Therefore, an action that previously resulted in a high reward is expected to provide a high reward if chosen again. This does not fit every problem, but prior work has identified *temporal homogeneity* [17] present in multiple CPU microarchitectural problems that makes them amenable to such a model.

A popular Multi-Armed Bandit algorithm is the Upper Confidence Bound (UCB) algorithm. UCB begins with an *initial exploration step*, during which it plays each action once to acquire initial reward estimates. After testing each arm once, UCB will choose the action a_i with the greatest *value*, which is its average observed reward $\hat{r}_i$ plus a bonus term to encourage the selection of infrequently-used actions:

$$\text{value}\,(a_i) = \hat{r}_i + c\sqrt{\frac{\ln T}{n_i}} \tag{1}$$

The UCB algorithm only requires tracking an action's average reward, the number of times it has been used (n_i), and the global timestep (T). The exploration/exploitation tradeoff is controlled by the hyperparameter c. When one operates in time-varying environments, the *Discounted* UCB (DUCB) algorithm is used, which defines an additional parameter, $0 < \gamma < 1$, which controls the "forgetfulness" of the agent.

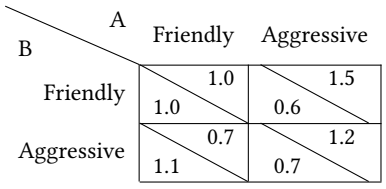


Figure 1: A general-sum game with two agents, where the rewards for the actions taken by A and B are shown in a grid. A's reward is more sensitive to changes than B's.

2.2 Multi-Agent RL and Games

Multi-Agent Reinforcement Learning (MARL) [2] is the extension of RL that deals with multiple agents acting within a single environment, and has deep ties to game theory [40]. In MARL, each agent's rewards depend on the actions of all agents (i.e., the *joint action*), not just its own. The agents play a "game," which may be *general-sum*—meaning that changes to one agent's reward are not always exactly counterbalanced by changes to the others.

2.2.1 Non-Cooperative Agents & Nash Equilibria. In general-sum games, the opportunity may arise for one agent to improve its own reward, but at the cost of worsening another agent's reward. In multicore prefetching, this may occur because a private prefetcher can adopt an aggressive policy, which boosts its core's performance but increases resource contention.

Consider a simplified two-player MARL game shown in Figure 1 (which is similar to the classic Prisoner's Dilemma [40]). Each agent can choose to adopt either a Friendly or an Aggressive policy. The numbers in the grid are the rewards for the actions taken by Agent A and Agent B. For example, if Agent A is Aggressive and Agent B is Friendly, the rewards for Agent A and Agent B are 1.5 and 0.6, respectively.

Choosing to be Aggressive will increase that agent's reward, but decrease the other's. Due to each agent's attempts to maximize its own reward, two non-cooperating agents will each adopt the Aggressive strategy, and wind up playing the {Aggressive, Aggressive} joint action. This is called a *Nash Equilibrium* [2, 38]: neither agent can unilaterally improve its reward by changing its own action. For example, if Agent B decided to be Friendly, its own reward would decrease from 0.7 to 0.6, so it will not do this. However, to maximize the sum of the rewards, Agent B must choose to be Friendly.

Note that there is room for ambiguity in Figure 1: although {Aggressive, Friendly} (top-right) provides the largest *total* reward, {Friendly, Friendly} distributes the reward *fairly* between the agents without giving up much of the total. The optimal strategy will therefore depend on what we value when defining the system's "performance." As we describe in Section 3.1, we discover similarities between multicore RL prefetching and the game presented here: individual agents prefer aggressive actions in multicore environments, where bandwidth is limited, even when it would be better for the system if some of them behaved in a friendlier fashion.

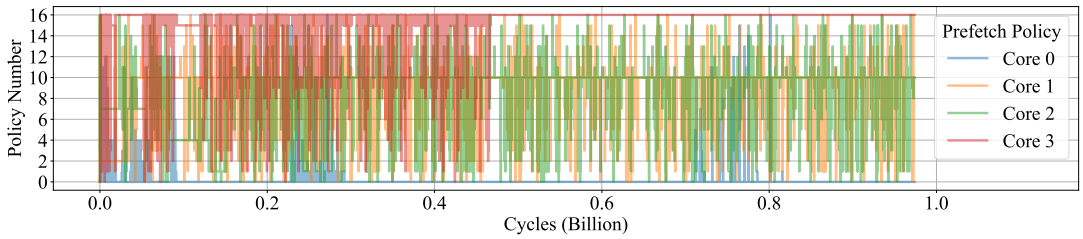


Figure 2: Prefetching policies chosen by four Micro-Armed Bandit agents in a four-core multicore running traces from SPEC and Ligra, as a function of time. The policies are sorted from most aggressive (top of the Y axis) to least (bottom).

2.2.2 Cooperative Agents & Credit Assignment. One seemingly-obvious "fix" for this kind of game is to provide every agent with the *same* reward: for example, the sum of the two components. The {Aggressive, Friendly} joint action would now provide a reward of 2.1 to both and the {Aggressive, Aggressive} joint action would provide 1.9. Now, the agents might converge to the right policy, namely, {Aggressive, Friendly}.

However, if each agent is unaware of the other's actions, this can lead to an exacerbated *credit-assignment problem* [2, 51, 60]. When Agent A chooses to be Aggressive, it might now see a reward of 1.9 (if B was Aggressive), and when Agent A chooses to be Friendly, it might now see a reward of 2.0 (if B was Friendly). After observing these rewards, Agent A may choose to be Friendly in the future, despite the optimal joint action requiring Agent A to be Aggressive. The problem is that Agent A did not properly assign credit for the rewards that it saw: it assumed responsibility for the +0.1 difference but, in reality, the reward was also impacted by the changing state of Agent B.

With more agents and more actions available to each, the credit-assignment problem worsens. Apparently obvious solutions often lead to other difficulties—e.g., allowing only one agent to explore at a time slows the learning process of all agents, and tracking all possible joint actions quickly grows intractable, since the joint action space scales exponentially with the number of agents.

2.2.3 Prior Theory. There has been much prior theoretical work on general-sum and cooperative multi-agent bandit problems, both with communication [8, 26, 36] and without [20, 21]. Other similar problems, such combinatorial bandits, and hierarchical MARL more generally, have also received attention [30–32, 39, 41]. Our ultimate solution resembles some of the latter works, which are lightweight enough to implement in hardware due to their simplicity.

2.3 Micro-Armed Bandit Prefetcher

The Micro-Armed Bandit prefetcher [17] ("Bandit") uses a DUCB algorithm to control an ensemble of prefetchers in a private L2 cache. Bandit controls a stride, stream, and next-line prefetcher. At each timestep, Bandit chooses from a set of actions specifying the degree of each prefetcher. For example, a possible action can be to disable the next-line and stride prefetchers, and set the streamer to degree 4. As the reward, Bandit is given the core's normalized IPC, which it attempts to maximize.

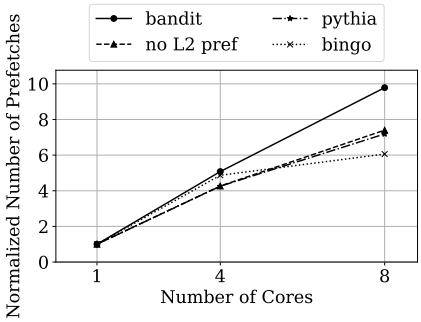


Figure 3: Normalized number of prefetches issued in four settings as the number of cores increases.

3 Motivation

To motivate the contribution of this paper, we now show that competing Bandit agents tend to converge to sub-optimal joint actions in multicores. We also consider a shared-reward solution and show that it quickly runs into the credit-assignment problem, limiting its utility. Finally, we comment on the challenges of using a single agent like Micro-Armed Bandit for "system-wide" control.

The experiments in this section use the setup of Section 5, with the Bandit prefetchers in the private L2 caches.

3.1 Competition Between Prefetchers

In multicore systems, each core and its associated prefetchers compete for resources such as memory bandwidth. This competition influences the learning process of RL prefetchers like Bandit that aim to maximize their own core's IPC.

Figure 2 shows the prefetching policies chosen by four Bandit agents in a 4-core multicore as a function of time. The workload is traces from SPEC06 [52], SPEC17 [53], and the Ligra graph processing suite [48]. The policies are sorted from most aggressive at the top of the Y axis to least aggressive at the bottom (see Section 5).

In this experiment, each agent prefers one policy over all others, but still regularly explores, as can be clearly seen by the noisy pattern. The agent in Core 0 turns its prefetcher off (Policy 0) for most of the time; Cores 1 and 2, which are running two different simpoints from 607.cactuBSSN_s, mostly use Policy 10, which is moderately aggressive; and Core 3 mostly uses the most aggressive policy (Policy 16). These configurations are similar to the actions that Bandit chooses in a single-core environment. However, in the multicore setting, better configurations exist. For example, enabling

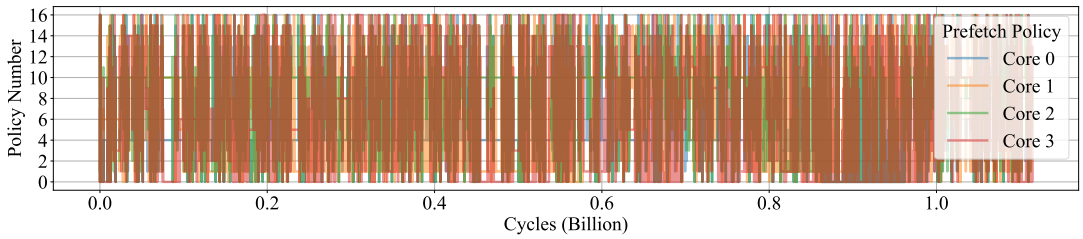


Figure 4: Prefetching policies chosen on the workload mix shown in Figure 2 when using a shared reward.

only the next-line prefetcher (Policy 1) for all cores on this workload reduces the demand on the memory system and results in 5% better system performance.

Figure 3 hints at the more general nature of this problem. The figure shows the normalized number of prefetches issued in four settings as the number of cores increases. The settings have the following prefetchers in L2: Bandit [17], Pythia [5], Bingo [3], and no prefetcher (no L2 pref). They all have a stride prefetcher in L1. The curves are normalized to the number of prefetches with 1 core.

We see that Bandit, unlike the other prefetchers, becomes *more* aggressive as the core count increases. In an eight-core system, the agents collectively issue almost ten times the number of prefetches as in the single core system, which is 25% more than the 8x we might expect. Although this is not a problem *per se*, it indicates that the multicore environment is affecting Bandit's learning process, and rather than dialing back the prefetchers, the results are policies that *increase* memory contention. Just as in Section 2.2, agents are unlikely to reduce their prefetcher's agressiveness when not forced to cooperate, since any core that deviates from this equilibrium policy will likely incur a slowdown.

3.2 The Credit Assignment Problem

As discussed in Section 2.2, a naïve approach to solving the problem of competition involves providing a shared reward (e.g., system performance) to these agents. This should encourage the agents to adopt actions that help the whole system, and not just themselves. However, such a scheme faces the credit assignment problem.

This problem occurs when agents can no longer distinguish the effects of their own actions from the effects of others'. A shared reward is suboptimal for an agent because it is affected by the noise introduced by the other agents. We described an example of this problem in Section 2.2.2. In here, we describe another example. Specifically, for a prefetch-sensitive workload, the actions that *most* affect a particular core's performance are often those of its own prefetcher. However, when a system is experiencing only light or moderate resource contention, a shared reward will just dilute the signal most relevant to the agent: the performance of the local core.

This can be seen in Figure 4, which shows the same workload as in Figure 2, but this time providing a shared reward to the agents. Although it is hard to see, Cores 0 and 2 manage to converge mostly to Policies 4 and 10, respectively, but the other two agents are unable to converge to a stable policy. On average, across the same 52 4-core workloads evaluated in Section 6, simply supplying a shared reward does not result in any meaningful performance gain.

3.3 A Very Large Search Space

To sidestep the credit assignment problem, one could treat the multicore system as a single ensemble of prefetchers and use only one Micro-Armed Bandit to optimize over their combined action spaces. Such an agent could directly optimize for system-wide performance without ever mis-assigning credit.

However, the action space for that agent would be far too large. In the 4-core system we consider, where each L2 prefetcher has 17 possible configurations, there are a total of $17^4 = 83{,}521$ joint actions available to the system. Just the initial exploration step of a DUCB system-level agent might take almost an second to complete, which is too long to exploit temporal homogeneity [17]. With 8 cores, this initial exploration step may take almost a day, and with 16, several millennia. A table for storing so many rewards would also be too costly.

In practice, even if we cannot directly iterate over this joint action space, we can still track a limited number of joint actions to try to distinguish the good from the bad, and use this to coordinate the prefetchers.

4 µMama System Design

To solve the problems just outlined, we introduce the µMama system, which builds upon Micro-Armed Bandit to provide intelligent system-aware prefetching. This section describes how µMama searches for high-performing joint actions with the assistance of a distributed set of local Bandit agents. µMama leverages these local agents to gain insights into promising joint actions, while guiding them towards cooperative policies. We provide a high-level algorithmic description and discuss the design of this system.

4.1 Local Exploration, Global Exploitation

µMama takes a balanced approach, utilizing both greedy local agents and system-level joint action tracking. In a similar fashion as prior theoretical work [39], locally-greedy Bandit agents in each private prefetcher explore promising actions, while a system-level agent monitors and overrides these agents to dictate system policy when it believes this will improve performance. In most cases, these local agents receive their local core's IPC as reward, whereas the system-level µMama supervisor is rewarded with an estimation of the system performance.

Algorithm 1 provides a high-level description of the µMama action selection algorithm. At each timestep, an RL µMama "arbiter" agent (Line 2) determines whether to dictate a known joint action to the n private prefetchers or to allow the n local agents to independently select actions. If dictating, µMama greedily selects

Algorithm 1 The μMama action selection algorithm

```
1: procedure GetSystemAction
2:     selectJointAction ← Arbiter()        ▷ Choose local or joint
3:     if selectJointAction then
4:         a ← arg max_k {r̂_k^sys}          ▷ JAV cache lookup
5:     else
6:         a ← {LocalAction_0, ..., LocalAction_{n-1}}
```

the previously-observed joint action that has the highest average reward (Line 4). Instead, if μMama's arbiter decides to allow the n local agents to choose their own actions, they do so based on their own local value estimates (Line 6).

This algorithm can be broken down into three distinct RL agent roles. First, there are the local agents located in each private L2, which operate almost exactly as in Bandit [17]. Second, there is a system-level agent that tracks high-performing joint actions and always chooses to exploit its best joint action. We call this agent the *Joint Action Value (JAV) cache*, which will be described Section 4.2. Finally, the arbiter agent controls which of the other two agents dictates system policy at each time step: either the local agents independently choose the actions to take, or a joint action from the JAV cache is used. The arbiter also strongly resembles Bandit [17], but it only has these two possible actions.

In this system, the local agents are primarily intended to support exploration of new joint actions. Rather than randomly sampling or iterating over the joint action space, local agents can help to focus the system's exploration on joint actions that have the potential to improve their core's performance. However, in some cases, increasing one core's performance may not matter much to the system (e.g., other cores may have more potential for speedup), but its prefetcher could still create interference. In this case, μMama rewards this local agent based on the system performance, rather than on its local core's performance (more details in Section 4.2.4).

A distinct advantage of using RL with performance-based rewards is flexibility. By changing the reward calculation, the same hardware can support different tradeoffs. As an example, increasing throughput in multicore systems often involves treating cores unfairly by prioritizing those with the greatest potential for speedup—and treating cores fairly often results in reduced throughput. To target a different point in this tradeoff, one only needs to change μMama's reward calculation, opening up the opportunity for flexible run-time configuration. We exploit this idea later.

4.2 Microarchitecture of μMama

We now present the detailed design of μMama, which coordinates local prefetcher controllers to optimize for system-level performance. In order to do this, μMama has to first be able to quantify the system's performance and determine which joint actions are high-performing. Once this is done, μMama can direct the local agents according to a system-level policy. Additionally, μMama should provide feedback to key local prefetchers in order to encourage them to explore policies that benefit the whole system. In this section, we present the details of how μMama accomplishes each of these to maximize system throughput, and then examine extending the design to other performance targets.

4.2.1 Computing System-Level Rewards. To optimize for system-level performance, μMama needs to be able to quantify it. A common multicore throughput metric is the Weighted Speedup (WS) [12–14, 49], defined in Equation 2. First, we compute the speedup of a core running on the system with its L2 prefetcher (*opt,MP* for optimized multiprocessor) relative to when it is running alone on the system without L2 prefetcher (*base,SP* for baseline single processor). Then, we take the sum of all these individual speedups.

$$WS = \sum_{i=0}^{n-1} S_i = \sum_{i=0}^{n-1} \frac{\text{IPC}_i^{\text{opt,MP}}}{\text{IPC}_i^{\text{base,SP}}} \tag{2}$$

To provide an accurate reward, we must estimate WS at runtime. The numerator, $\text{IPC}_i^{\text{opt,MP}}$, is the IPC of core i in the optimized system, which can be measured by μMama. However, the denominator is more difficult to measure: we must know how the workload behaves when it runs on a core alone on the system.

One way to estimate $\text{IPC}_i^{\text{base,SP}}$ is to profile each workload alone on the system, either offline or by periodically interrupting the others, and provide these profiles to μMama. Alternatively, one could heuristically approximate $\text{IPC}_i^{\text{base,SP}}$ in a way that can be computed at runtime. μMama adopts this latter approach, which does not require software support (see Section 6.6 for an evaluation of the former).

To estimate WS, first note that the speedup terms in Equation 2 are equivalent to the right-hand side of Equation 3.

$$S_i = \frac{\text{IPC}_i^{\text{opt,MP}}}{\text{IPC}_i^{\text{base,SP}}} = \underbrace{\frac{\text{IPC}_i^{\text{base,MP}}}{\text{IPC}_i^{\text{base,SP}}}}_{S_i^{\text{MP}}} \times \underbrace{\frac{\text{IPC}_i^{\text{opt,MP}}}{\text{IPC}_i^{\text{base,MP}}}}_{S_i^{\text{opt}}} \tag{3}$$

$\text{IPC}_i^{\text{base,MP}}$ is the IPC of Workload i when run in the presence of the other applications in the multicore, without prefetching. Then, S_i^{MP} is Workload i's "speedup" (more typically, slowdown) caused by the multicore environment, and S_i^{opt} is the speedup provided by L2 prefetching. The local agents already track something similar to S_i^{opt} (i.e., their normalized rewards, r_i).

Consider now how to estimate S_i^{MP}. We expect that workloads that miss more in their L2 will suffer a greater slowdown from the multicore environment. Let δ_i be the number of accesses issued by Workload i that would have missed the L2, per instruction executed, in the absence of an L2 prefetcher. The expression $\frac{S_i^{\text{MP}}}{\sum_{j=0}^{n-1} S_j^{\text{MP}}}$ is the fraction of the multicore-induced speedup contributed by Workload i. The same workload contributes a fraction of total L2 misses equal to $\frac{\delta_i}{\sum_{j=0}^{n-1} \delta_j}$, and $\left(1 - \frac{\delta_i}{\sum_{j=0}^{n-1} \delta_j}\right)$ is the fraction of total L2 misses contributed by all the workloads except i. Therefore, we estimate that $\frac{S_i^{\text{MP}}}{\sum_{j=0}^{n-1} S_j^{\text{MP}}} \approx \alpha \left(1 - \frac{\delta_i}{\sum_{j=0}^{n-1} \delta_j}\right)$ for some α. Re-arranging,

$$S_i^{\text{MP}} \approx \alpha \left(\sum_{j=0}^{n-1} S_j^{\text{MP}}\right)\left(1 - \frac{\delta_i}{\sum_{j=0}^{n-1} \delta_j}\right) \tag{4}$$

Because WS is a homogeneous function (i.e., $WS(c \times S_0, c \times S_1) = c \times WS(S_0, S_1)$), we can maximize it while ignoring the multiplicative terms common to all cores (in particular, $\alpha \times \sum_{j=0}^{n-1} S_j^{\text{MP}}$). This yields

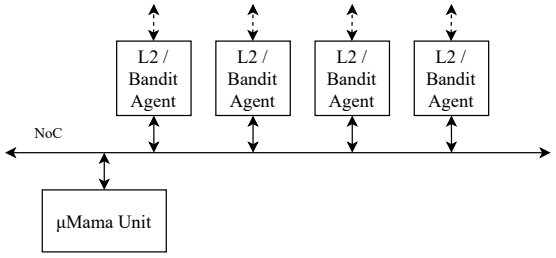


Figure 5: Local agents communicate with µMama Unit.

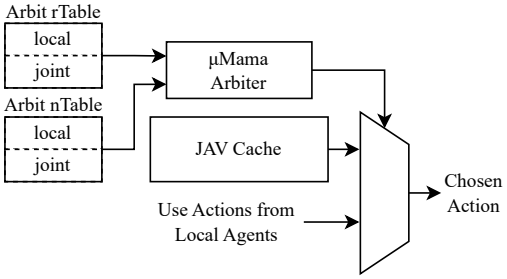


Figure 6: The µMama Arbiter chooses between a joint action taken from the JAV cache or actions taken by local agents.

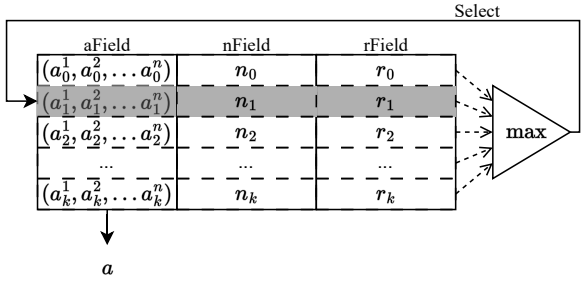
the following approximation for S_i, which we use to estimate WS at runtime.

$$S_i \approx \tilde{S}_i = \underbrace{\left(1 - \frac{\delta_i}{\sum_{j=0}^{n-1} \delta_j}\right)}_{\tilde{S}_i^{\text{MP}}} \times \underbrace{r_i}_{\tilde{S}_i^{\text{opt}}} \tag{5}$$

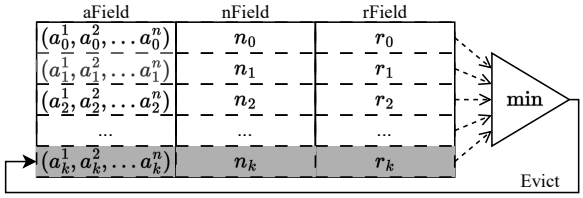
4.2.2 Disambiguating Credit Assignment with the JAV Cache. Now that we have a global reward signal, µMama must associate joint actions with their expected rewards. This is done with the central µMama Unit, which communicates with the private-cache Bandit agents over an on-chip network, as shown in Figure 5. The µMama Unit contains both the µMama *Arbiter* and the *Joint Action-Value* (JAV) cache, shown in Figure 6.

The µMama Arbiter is a DUCB Multi-Armed Bandit agent, similar to the local agents, but with only two possible actions (shown in the figure as the two inputs of a multiplexer): use a joint action from the JAV or use actions chosen by local agents. To make its decision, the µMama Arbiter uses two tables, namely, *Arbit nTable* and *Arbit rTable*. *Arbit nTable* tracks the number of times each of the arbiter's possible actions (*local* or *joint*) has been taken; *Arbit rTable* tracks the average reward for each of the two actions. The arbiter is queried once every T_{arbit} timesteps. For example, if it chooses to allow the local agents to operate freely, they explore for T_{arbit} time steps before the arbiter is queried again. Otherwise, µMama enforces joint actions from the JAV cache for T_{arbit} steps.

The JAV cache is a hardware cache where each entry corresponds to a previously applied joint action. An entry has three fields: the joint action description (*aField*), the number of times the joint action has been played (*nField*), and the average reward for the joint action (*rField*). The table is indexed by the joint action. A simple design would implement the cache in a fully-associative manner.



(a) Selecting the highest-reward joint action for play.



(b) Selecting the lowest-reward joint action for eviction.

Figure 7: Two operations supported by the JAV cache.

The JAV cache supports two operations. First, it can look up the action with the highest *rField* (Figure 7a), which µMama uses whenever the arbiter overrides the local agents. Second, it can insert a new action or update an existing action's *nField* and *rField* with new system-level rewards. When the JAV cache updates its tables at the end of each timestep, it may need to evict an entry to make room for a new joint action entry. It will always choose to evict the worst-performing action, based on its *rField*, as shown in Figure 7b. The JAV cache does not evict any entry if the incoming action appears less rewarding than every currently-tracked action.

As Section 4.3 discusses, µMama is very tolerant of the latencies involved in querying and updating the JAV cache. This tolerance allows the JAV cache to be implemented as a single-ported structure, with max/min operations implemented by sequentially scanning over the entries, rather than requiring costly reduction trees.

It is best to keep the size of the JAV cache relatively modest since, unlike local agents, the JAV cache does not explore lower-performing actions. Therefore, during program phase changes, a large JAV cache may take a long time to stop picking actions that were highly rewarding in the past, but are not anymore.

4.2.3 Managing the Complexity of the JAV Cache. In our evaluation, we use a small, low-overhead JAV cache. However, other systems may need larger JAV caches, especially if the agent step sizes are small or their action spaces are large. This could lead to high area and energy requirements if we make the JAV cache fully associative or if action selection involves comparisons between every entry's *rField* (Figure 7a). Moreover, since the *aField* size scales linearly with core count, lookup operations required for updating the JAV cache may also grow more expensive in large systems. To manage this, two simple changes can be made.

First, the JAV cache can be made set-associative. The sets would be indexed by a hashes of the *aField*, and each entry would still be tagged by the *aField*. To minimize collisions, these hashes should

mix bits from throughout the *aField* so that the set depends on the policies of all the cores. The tradeoffs involved in choosing the associativity of the JAV cache are the same as for any cache: decreasing the associativity leads to more evictions of data, but reduces the resource requirements. It also reduces the number of comparisons required for eviction (Figure 7b).

Second, the number of comparisons required for action selection (Figure 7a) can be reduced. By maintaining a copy of the best-performing joint action and its reward, we eliminate the need to compare every entry. When the JAV cache updates the reward of an entry, it must only check to see whether the updated entry now surpasses the previous best, and if so, update the copy.

4.2.4 Providing Global Rewards to Local Agents. As discussed previously, shared rewards often exacerbate the credit-assignment problem, especially since local actions *usually* affect the local core more than the rest of the system. However, we observe that, often, some workloads in a system contribute much less to WS than others. Providing these "low importance" cores with global rewards can prove useful: rather than maximizing their own performance, which has negligible impact on the system as a whole, they can focus on reducing the negative impacts that their policies might have on the rest of the system.

To see how this works, consider two cores in a larger system, where Core 0 is slowed down a lot by multicore contention, but Core 1 is not (i.e., $S_0^{\text{MP}} \ll S_1^{\text{MP}}$). Since a core's total contribution to WS is determined by $S_i^{\text{MP}} \times S_i^{\text{opt}}$, it may be profitable reduce prefetching aggressiveness in Core 0 (potentially decreasing S_0^{opt}) and free up enough resources for Core 1 to prefetch aggressively (increasing S_1^{opt}). This tradeoff becomes more promising the greater the difference between S_0^{MP} and S_1^{MP} is.

With this in mind, if $\tilde{S}_i^{\text{MP}}$ (Equation 5) is below some small threshold θ_{global}, μMama will provide to agent i the system-level ("global") reward, rather than a local reward. This approach is expected be more helpful than having agent i optimize its own performance, since its own performance is expected to be poor anyways.

4.2.5 Extending to Other Performance Metrics. So far, we have used WS as our system-level performance metric. A commonly-used alternative is the Harmonic Mean Speedup (HS):

$$HS = \frac{n}{\sum_{i=0}^{n-1} \frac{1}{S_i}} \tag{6}$$

where S_i is the same as in Equation 2. Unlike WS, HS emphasizes fairness: there are quickly-diminishing returns for improving the performance of only one core. To obtain high HS, speedups need to be balanced between cores. Thanks to μMama's design, it is trivial to change the system reward calculation for HS: μMama just uses the same approximations of S_i as in Equation 5.

With HS, determining which local agents receive system-level rewards (because it is relatively unimportant to increase their core's performance, as discussed in Section 4.2.4) is only slightly more complex. To see why, recall that, when using WS, μMama focuses on estimating S_i^{MP}, since the slope of WS with respect to S_i^{opt} is S_i^{MP}, and thus a small value of S_i^{MP} limits the importance of S_i^{opt}. Similarly, when using HS, μMama focuses on estimating $\frac{\partial HS}{\partial S_i^{\text{opt}}}$, which is equal to $\frac{S_i^{MP}}{n} \times \left(\frac{HS}{S_i}\right)^2$. Hence, μMama compares this value to the threshold θ_{global} and, if it is less than θ_{global}, provides the system-level reward to local agent i. Further, since θ_{global} is an arbitrary constant, with the appropriate choice of θ_{global}, it is only necessary to compare it to $S_i^{MP} \times \left(\frac{HS}{S_i}\right)^2$.

In practice, the desired system-level performance metric may not be known at design time: the same chips may be sold to both high-throughput supercomputing facilities and cloud datacenters, which may have different requirements. For this reason, designers can include many possible optimization targets, each with different tradeoffs, to be selected at installation- or run-time.

4.3 Communication and Latency

We now consider the communication requirements of μMama and how this communication is scheduled to make the system highly tolerant of network and computation latencies.

4.3.1 Need for Communication in μMama. Micro-Armed Bandit [17] defines its timesteps in terms of the number of demand accesses to the L2, rather than as a fixed cycle count. This is so that all types of workloads, fast or slow, can observe the effects of their prefetching policy before the agent must determine the next step's policy.

However, when coordinating several agents, it is beneficial to have a *common timestep*. To accomplish this, in our design, when a local agent reaches its threshold number of L2 demand accesses (*step*), it sends a message to μMama to mark itself as ready to advance. Once the majority of local agents do so (or once one local agent reaches k_{step} times such threshold), μMama broadcasts instructions to begin the next timestep. Thus, advancing one timestep requires one first round-trip to the μMama unit for most local agents.

Additionally, each agent must send the μMama unit its local r_i and δ_i (Section 4.2.1) at the end of each timestep so that μMama can compute system-level rewards. δ_i is estimated by recording the number of useful prefetches and the number of L2 misses. The μMama unit updates its tables based on this information and sends the local agents instructions for the next timestep, including any global rewards. This incurs another round-trip per local agent and timestep.

4.3.2 Hiding Latency: Planning One Step Ahead. These two communication round-trips, along with the computations involved in updating the μMama unit's state, would create extra latency in each timestep. This would not stall any part of the system, since the prefetchers would continue to operate according to the policies of the previous step. However, this could cause additional variations in timestep sizes, which should be avoided.

To minimize this effect, μMama hides most of this communication and computation by *planning one timestep ahead*. Figure 8 illustrates this approach. After enough "finished" messages for timestep n arrive ①, μMama immediately broadcasts instructions to start timestep $n+1$ ②. Local agents then select their actions and begin timestep $n+1$, while in parallel sending their r_i and δ_i from step n to μMama ③. μMama then computes the system-level reward, updates its JAV cache and arbiter, queries both as appropriate, and determines the system's policy for timestep $n+2$. μMama

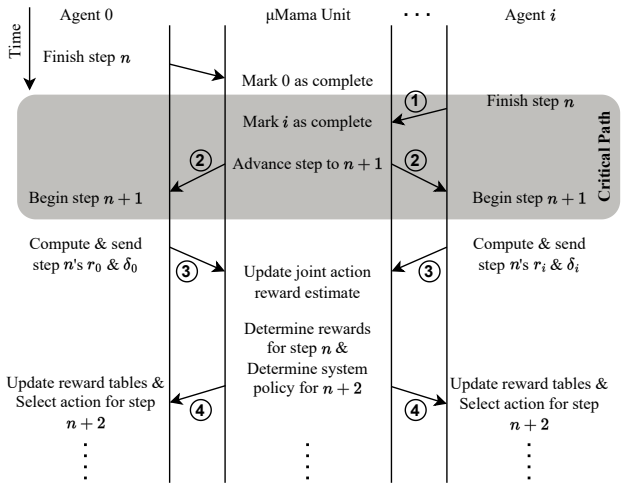


Figure 8: The μMama unit communicates regularly with the local agents to synchronize timesteps and coordinate actions.

sends this system policy information for timestep $n+2$, as well as any necessary global rewards for timestep n, to the local agents ④, who will update their tables and buffer the system policy for timestep $n+2$ until the latter begins.

Let Agent i be the one that completes the majority and ends timestep n. Using the above schedule, the "critical path" between Agent i becoming ready for the next timestep and the system beginning step $n+1$ consists only of Agent i sending a message to the μMama unit ①, followed by a broadcast from the μMama unit to all local agents ②. All latencies involved in sending individual updates between the agents and the μMama unit, as well as the latency of updating and querying the JAV cache and arbiter, are completely hidden behind the ongoing timestep.

4.4 Design overheads

Along with additional communication, μMama introduces minimal storage and logic requirements.

4.4.1 Storage and Logic Overheads. The JAV cache is the central storage structure of the μMama unit. This is a small fully-associative structure. The tag (i.e., the *aField* bits) size grows linearly with the number of cores and logarithmically with the number of actions available to the local prefetcher agents. The reward estimates (*rField* bits) and counters (*nField* bits) are double-precision floats in our experiments. In our evaluation, we use a 2-entry JAV cache. For our 8-core system with 17 local arms, the *aField* has 40 bits. Overall, the JAV cache has a total size of 336 bits, or 42 bytes, counting both tags and data. This is only about 2% the size of the combined storage used by the local Bandit agents in this system.

Following the methodology in [17], we estimate the area of a larger JAV cache using CACTI [4], and the area of the floating-point logic from the numbers in [44]. We then scale to 10nm using the approach in [54]. In a 10nm 40-core system with a larger JAV cache of 64 entries, the μMama unit would only use an area of 0.00712 mm^2. Relative to a server-class 40-core Intel Icelake [9], which has a total die area of 628 mm^2, this is an area overhead of about 0.001%.

Table 1: Some prefetcher parameters used in the evaluation.

Prefetcher	Parameters
Bandit [17]	$c = 0.01$; $\gamma = 0.9995$; step = 800 accesses; with 64-entry stride/streamer prefetchers
μMama	step = 800 accesses, $\theta_{\text{global}} = 1 - \frac{1.4}{n}$, $k_{step} = 5$ **Local Agents:** $c = 0.01$; $\gamma = 0.995$ **Arbiter:** $c = 0.1$; $\gamma = 0.995$; $T_{arbit} = 5$ **JAV Cache:** 2 entries; $\gamma = 0.999$

4.4.2 Communication Overheads. The μMama system requires regular communication between the local agents and the μMama unit, as described in Section 4.3. However, this communication is very light: each agent exchanges only 27 bytes of data with the μMama unit during each timestep, and only 2 bytes during the "critical path" highlighted in Figure 8. During our evaluation, the average timestep length on an eight-core system is about 150,000 cycles, or 38 μs. In a 40-core system with the same timestep length, this would correspond to a total data rate of about 28 MB/s, or 710 kB/s per prefetcher agent. Such a low data rate is inconsequential to modern NoCs, which reach dozens of GB/s or more, even when crossing chiplet boundaries [37]. Due to this low communication overhead and μMama's latency tolerance, μMama can reuse any existing NoC with a suitable interface.

A typical timestep of 38 μs also means that network latencies should not significantly affect μMama. The latency between a local agent and the μMama unit in a machine should be similar to the core-to-core latency in that machine, which range from dozens to hundreds of nanoseconds in modern systems [15]. This is no more than a few percent of a timestep. If a system has much larger latencies than this, then the system architect may increase the timestep size, at the cost of making μMama's learning process somewhat slower.

5 Methodology

5.1 Architecture

We evaluate μMama using a recent version[1] of Champsim [18], which is a trace-based microarchitectural simulator. We port Micro-Armed Bandit [17], Bingo [3], and Pythia [5] to this version of Champsim. For each prefetcher, we consider several hyperparameters and choose the set that performs best for a single core. We also provide some comparisons using a stride prefetcher shipped by Champsim. All of these prefetchers are placed at the L2, with an additional low-degree stride prefetcher located in the L1D.

Some key parameters for Bandit and μMama are shown in Table 1. Since many of μMama's hyperparameters are similar to those found in Bandit [17], we perform only a limited parameter space search. Specifically, we set *step* to 800 (as in Bandit) and restrict the exploration of c and γ to values within about one order of magnitude of the values known to work with Bandit. We determine the μMama-specific parameters (i.e., T_{arbit}, θ_{global}, k_{step}, and JAV size) by scanning within ranges determined to be feasible. We use 25 four-core workloads to guide these explorations. The final parameters have certain intuitiveness. For example, μMama's local

[1] Commit 9fc6b604ce56f9e0f39a8fe3ed9f1e8eab4435a9

Table 2: Bandit arms used in our experiments.

Arm ID	0	1	2	3	4	5	6	7	8
Next-Line On?	✗	✓	✗	✗	✗	✗	✗	✗	✗
Stride Degree	0	0	0	0	2	0	2	0	0
Streamer Deg.	0	0	2	3	2	4	3	5	6

Arm ID	9	10	11	12	13	14	15	16	
Next-Line On?	✗	✓	✗	✗	✗	✗	✗	✗	
Stride Degree	0	0	4	4	8	0	8	15	
Streamer Deg.	7	6	4	5	6	15	7	15	

Table 3: Default system used in the evaluation.

Component	Configuration
CPU	1, 2, 4, or 8 CPUs; 4GHz; ROB=352; LQ=192; SQ=114; Fetch Buffer: 64; Fetch Width: 16; Decode Width: 6; Dispatch Width: 6; Exec Width: 10; Retire Width: 10; BP: hashed_perceptron
L1I Cache	32 KB (64 sets x 8 ways); 4 cycle hit; no prefetcher; 64B line
L1D Cache	48 KB (64 sets x 12 ways); 5 cycle hit; 24-entry ip_stride prefetcher; 64B line
L2 Cache	1MB (1024 sets x 16 ways); 10 cycle hit; experiment-specific prefetcher; 64B line
Shared LLC	6 MB (8,192 sets x 12 ways); 40 cycle hit; no prefetcher; 64B line
DRAM	2400 MT/s; 1 channel; 4 ranks; 8 banks; 65536 rows; 1024 columns

agents and arbiter use lower γ values than Bandit, since their roles are to explore changing environments, but the JAV cache uses a higher γ to better remember high-performing joint-actions. We use a wider selection of arms for Bandit than in the original work, shown ordered by total degree in Table 2.

We model the latency of µMama's "critical path" identified in Figure 8 as a constant 200 cycles, which is similar to the same-socket core-to-core round-trip latencies in recent server-class processors [15]. The shared-reward threshold θ_{global} is parameterized on the number of active cores, n, and is the same for WS and HS.

Unless otherwise specified, experiments use the system configuration presented in Table 3. With 4 cores active, this configuration provides a per-core memory bandwidth of 4.8 GB/s, and with 8 cores active, it provides 2.4GB/s per core. These are similar to the per-core memory bandwidths supported by Sapphire Rapids [22] and Sierra Forest [50] systems, respectively.

In our simulations, we use a brief warmup of 1 million instructions per core, and simulate until each core executes at least 250 million more. We report statistics from these latter 250 million. If any core reaches the end of its trace before the simulation is finished, the trace is restarted.

5.2 Workloads

We use a collection of prefetch-sensitive traces for our evaluation, taken from the SPEC CPU 2006 [52], SPEC CPU 2017 [53], Ligra [48], and PARSEC 2.1 [7] benchmark suites. These traces were released by the 3rd Data Prefetching Championship [1] and Pythia [5]. We identify "prefetch-sensitive" traces as those that observe a greater than 10% performance change when run alone with one of our prefetchers, compared to without L2-prefetching. Out of 324 tested traces, 148 meet this criteria, comprised of 50% from Ligra, 22% from SPEC06, 20% from SPEC17, and 8% from PARSEC. This set includes scientific programs, datacenter workloads, compilers, and more. Such workloads may be commonly co-located on high-throughput or multi-tenant systems such as supercomputers or cloud nodes. We randomly sample from traces to generate 52 multicore workload mixes for 4- and 8-core systems.

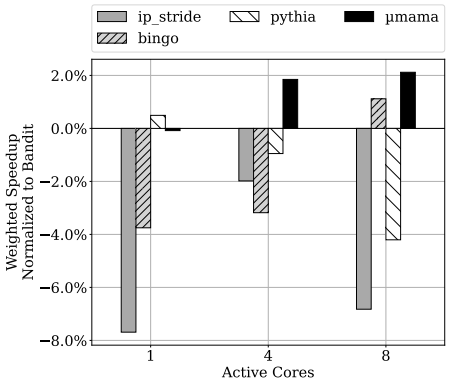


Figure 9: Average Weighted Speedup of four prefetchers, normalized to that of uncoordinated Micro-Armed Bandits.

5.3 Evaluation Metrics

Our evaluation involves an analysis of both *throughput* and *fairness*, each of which is important in its own context [14]. To measure system throughput, we use the Weighted Speedup (WS) [12–14, 49], introduced in Equation 2. In our fairness evaluation, we use the Harmonic Mean Speedup (HS) from Equation 6. HS is commonly used to jointly quantify speedup and fairness [12–14]. In addition, we also report the *unfairness* of the various prefetcher configurations, which quantifies the maximum degree to which one workload is prioritized over another in a given workload mix. We define the unfairness as in [12].

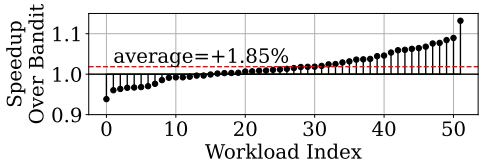
$$Unfairness = \frac{MAX\{S_0, S_1, \ldots, S_{N-1}\}}{MIN\{S_0, S_1, \ldots, S_{N-1}\}} \tag{7}$$

6 Evaluation

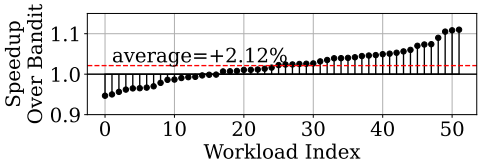
In this section, we evaluate µMama. First, we show that µMama achieves higher performance than Micro-Armed Bandit on multicore systems with a per-core memory bandwidth similar to modern server-class processors. Next, we show that µMama tends to use less aggressive prefetcher configurations than Bandit, and leads to a fairer system. Finally, we show that µMama's advantage over Bandit increases when the system has limited memory bandwidth.

6.1 Throughput

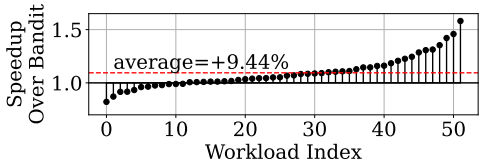
Figure 9 compares the average normalized Weighted Speedup of four prefetchers, compared to the performance of uncoordinated Bandit prefetchers, when 1, 4, or 8 cores are active. When 4 or 8 cores are active, µMama shows speedups of 1.9% and 2.1%, respectively, over the system using the independent Bandit agents. The



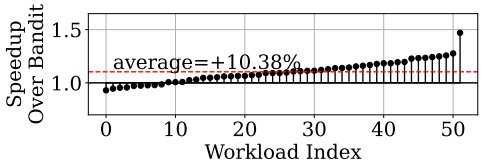
(a) Weighted Speedups of µMama normalized to Bandit (4-core).



(b) Weighted Speedups of µMama normalized to Bandit (8-core).



(c) Harmonic Speedups of µMama normalized to Bandit (4-core).



(d) Harmonic Speedups of µMama normalized to Bandit (8-core).

Figure 10: Weighted and Harmonic Speedups of µMama normalized to Bandit when using 4 cores and 8 active cores.

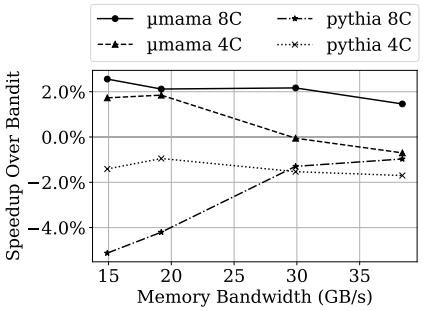


Figure 11: Weighted Speedups of µMama and Pythia normalized to Bandit for different values of the memory bandwidth.

other prefetchers generally underperform Bandit, with the exception of Pythia in the single-core case, which performs comparably, and Bingo in the 8-core case, probably due to Bingo's relatively conservative prefetching and high accuracy.

It can be shown that the total number of prefetches issued to all L2 caches when using µMama is about 23.9% lower than when using uncoordinated Bandits in the four-core case, and about 15.5% lower in the eight-core case. This reduction in prefetches in µMama brings µMama in line with the other prefetchers shown in Figure 3. Yet, µMama does not lower prefetcher aggressiveness across the board: on average, about 1.5 agents *increase* their prefetching aggressiveness in the 4-core workloads, and 3.5 do so in the eight-core workloads. This suggests that µMama successfully increases overall throughput by choosing to prioritize some cores over others.

Figures 10a and 10b show µMama's speedups normalized to Bandit for each trace mix used in the 4- and 8-core experiments, respectively. µMama shows a performance improvement in the majority of workload mixes in both cases, with multiple workloads observing speedups above 10%.

6.2 Scaling Memory Bandwidth

Along with our baseline system, described in Table 3, which uses a single channel of memory resembling DDR4-2400, we evaluate µMama on a system with one channel of DDR4-1866, and on systems with two channels of DDR4-1866 or DDR4-2400. Figure 11 shows how the Weighted Speedup of µMama and Pythia normalized to independent Bandit agents varies with memory bandwidth when using 4 cores (*4C*) and 8 cores (*8C*). The figure shows that µMama provides the greatest advantage over Bandit when memory bandwidth is low. This is expected since, with low bandwidth, the uncoordinated prefetchers in Bandit create high contention. In the most bandwidth-constrained system, µMama shows a speedup of 2.56% with eight cores. In contrast, Pythia's relative speedup can decrease with tighter memory bandwidth, as can be seen for the 8-core curve.

6.3 Understanding Workload Characteristics

As Figures 10a and 10b show, some workloads benefit from µMama more than others. The four-core workload with traces from SPEC06's 429.mcf & 435.gromacs, SPEC17's 649.fotonik3d_s, and Ligra's PageRankDelta slows by 4%, but the workload with (different) traces from 435.gromacs & 649.fotonik3d_s, along with Ligra's BC & Bellman-Ford, speeds up by 13.2%. Of the eight traces in these two workloads, seven are present in workloads with speedups and also in workloads with slowdowns. To understand why, we need a system-level view.

When the workloads benefiting most from µMama are run without an L2 prefetcher, they tend to either have a relatively low average number of L2 misses per thousand instructions ($\mu_{\text{L2-MPKI}}$), a relatively high variance ($\sigma^2_{\text{L2-MPKI}}$), or both. The high variance allows µMama to identify "low-importance" cores to receive a system-level reward more often (Section 4.2.4). The relatively low average L2-MPKI tends to mean that conservative prefetcher configurations can still achieve good performance: indeed, these workloads tend to see larger reductions in prefetcher aggressiveness than the others when using µMama. If we restrict our analysis to the 56% of workloads for which $\mu_{\text{L2-MPKI}} - \sigma_{\text{L2-MPKI}} < 2.5$ MPKI, µMama's speedup is 2.7% on four cores and 3.4% on eight.

One characteristic that is common to all the workloads that we consider is that they are sensitive to prefetching. This sensitivity is what drives competition between Bandits (Section 3.1), and managing this competition is an important reason for µMama's

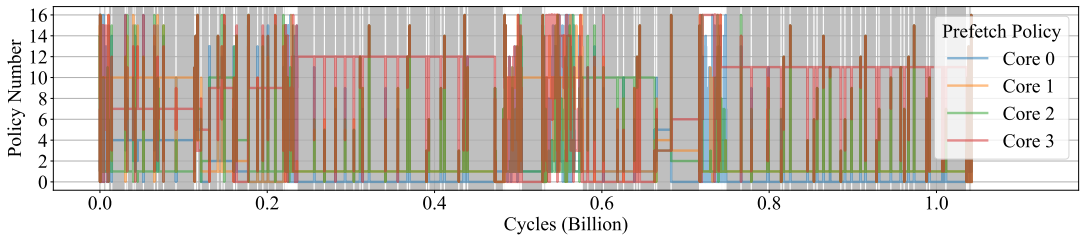
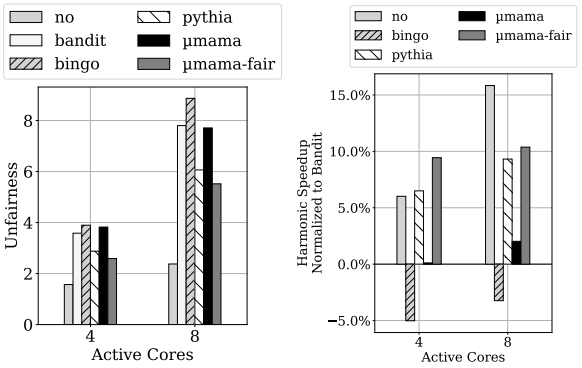


Figure 12: Prefetching policies chosen on the workload mix shown in Figure 2 when using the throughput-oriented µMama. The gray-shaded parts of the plot are where the arbiter enforces a joint action from the JAV cache.



(a) Prefetcher Unfairness (lower is fairer).

(b) Harmonic Speedups (higher is better).

Figure 13: Prefetcher Unfairness and Harmonic Speedups normalized to Bandit for different prefetchers.

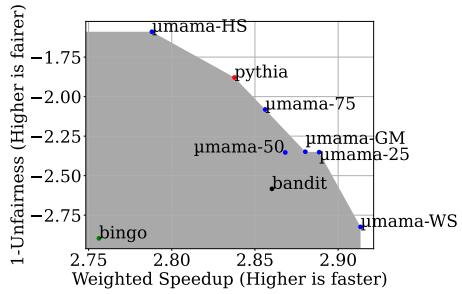


Figure 14: Tradeoff between performance and fairness for µMama designs with different reward functions and other prefetchers.

speedup improvements. The non-sensitive workloads (defined using the criteria from Section 5.2), in addition to simply not being as affected by prefetching policies, have less of a competition problem for µMama to fix. For these workloads, µMama only delivers an average speedup of 0.4% over Bandit for eight-core workloads.

6.4 µMama for Fairness

In user-facing or micro-service environments, fairness between cores may be valued as much as, or more, than throughput [14]. In this section, we evaluate how µMama can be used to improve the fairness of the system as described in Section 4.2.5. Specifically, we compare µMama, which uses WS, to µMama-Fair, which uses HS. We also show how more precise tradeoffs can be made by changing the reward function. No hyperparameter beyond the reward function is changed. Recall that Equation 7 defines *Unfairness*.

Figure 13a shows the Unfairness measure for several prefetchers. We see that, in both 4- and 8-core workloads, µMama-Fair shows a ≈30% reduction in Unfairness compared to Bandit. Figure 13b shows that this translates into µMama-Fair's 9.44% and 10.38% higher harmonic speedup normalized to Bandit in the 4- and 8-core workload mixes, respectively. Figure 13b also shows that µMama-Fair has a significantly higher harmonic speedup compared to the throughput-oriented µMama, which in turn achieved higher WS than Bandit with a similar unfairness.

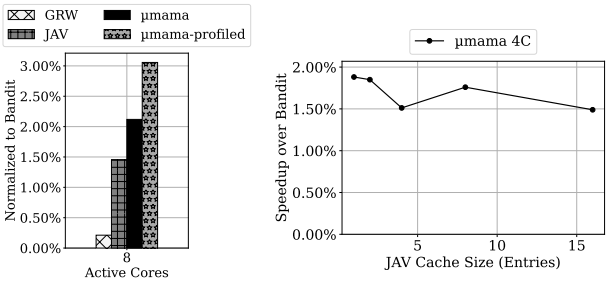
Figures 10c and 10d show µMama-Fair's harmonic speedups normalized to Bandit for each trace mix used in the 4- and 8-core experiments, respectively. µMama-Fair shows large harmonic speedups in the majority of the workload mixes in both cases.

Yet the tradeoff between throughput and fairness does not end with these two configurations. System engineers may want to further tune the prefetching to get the best performance for their particular application. With µMama, this means fine-tuning the reward function. Specifically, we consider configurations that use rewards equal to $(1-\alpha)\cdot WS + \alpha \cdot HS$, $\alpha \in \{0, 0.25, 0.50, 0.75, 1\}$, and we call these µMama-WS, µMama-25, µMama-50, µMama-75, and µMama-HS. We also consider µMama-GM, a configuration that uses the geometric mean of core speedups.

Figure 14 shows the tradeoff between performance and fairness for these µMama designs and other prefetchers. In the figure, the Y axis shows 1 minus the average Unfairness, so that higher numbers are better. A perfectly fair system has a (1-Unfairness) value of 0. The X axis shows the absolute average Weighted Speedup. We see that these µMama configurations together form a clear performance frontier, allowing system designers the opportunity to easily analyze the tradeoffs involved for their particular applications. On the other hand, Bandit is non-Pareto-optimal. Although Pythia lies on the Pareto frontier, µMama maintains more flexibility, as its various configurations cover a wide swath of throughput/fairness tradeoffs.

6.5 Arbiter Decisions

Figure 12 shows the prefetching policies chosen on the workload mix shown in Figure 2 when using the throughput-oriented µMama.



(a) Component breakdown. **(b) Speedup vs. JAV cache size.**

Figure 15: µMama's sensitivity to its various components.

The gray-shaded sections of the trace show when the arbiter enforces a joint action from the JAV cache—as opposed to letting local agents explore individually. We see that this workload primarily uses actions from the JAV cache, with occasional breaks to allow the local agents to explore other actions. In contrast to Bandit's actions on this same workload (Figure 2), µMama disables the streamer for Cores 1 and 2, while decreasing the degrees of Core 3's prefetchers. It can be shown that the result is higher performance for Cores 1 and 2, similar performance for Core 0, and only slightly reduced performance for Core 3. In total, the *WS* improves by almost 10% over Bandit for this workload.

Some workloads use the arbiter more than others but, on average, the four-core workloads and eight-core workloads use actions from the JAV cache 64% and 67% of the time, respectively.

6.6 Sensitivity to Design Components

µMama contains several important components, from the JAV cache to heuristics used to estimate speedups. In this section, we evaluate µMama's sensitivity to several of these.

6.6.1 JAV and Global Rewards. The µMama algorithm consists of two major components: the JAV cache and the global reward (GRW) assignment to low-impact local prefetchers. Figure 15a shows how each of these two components contributes to µMama's improvement in weighted speedup normalized to Bandit in an eight-core system. We see that, alone, the global reward does not improve performance much, while the JAV cache alone provides about 1.5% uplift over Bandit. The two features combine synergistically, delivering a 2.1% improvement over Bandit.

6.6.2 Sensitivity to the JAV Cache Size. In our evaluation, we use a JAV cache of size 2, which requires very few resources and quickly adapts to phase changes in the workload thanks to rapid entry eviction. Figure 15b shows µMama's weighted speedup normalized to Bandit on 4-core workloads for different JAV cache sizes. The speedup degrades slightly as the JAV cache grows, probably due to the additional time required to update reward estimates after program phase changes. However, as the size of the multicore grows, we expect that so too will the optimal JAV cache size.

6.6.3 Profile-Guided Reward Estimation. Equation 4 introduced an estimate of S_i^{MP}, the speedup (or slowdown) of a trace as it moves from running on a core in an unloaded multicore to running on a core in a multicore where all the other cores are busy. This estimate

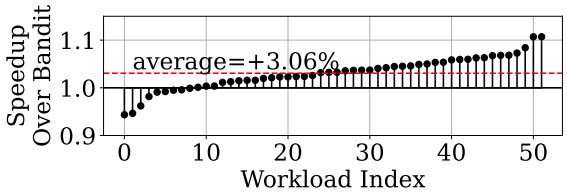


Figure 16: Weighted Speedups of µMama-Profiled normalized to Bandit for 8 core.

is easy to compute, and using it as part of the reward function helps to attain good performance. However, it is only approximate. To show this, we computed the relative values of the S_i^{MP} estimations across cores, since these are the weights given to the observed rewards (r_i) of the different cores. Then, we compare these values to their ground truth. We find that, for our eight-core experiments, the Pearson correlation coefficient between these two is only 0.51.

A more accurate approach involves profiling the performance of each trace alone in the unloaded multicore, and then providing these profiled numbers to µMama at run-time. We call the resulting system *µMama-Profiled*. Figure 15a shows the weighted speedup normalized to Bandit for µMama-Profiled. We see that µMama-Profiled achieves a 3% speedup over Bandit, which is 0.9% higher than when using µMama with the estimated values of S_i^{MP}.

Figure 16 shows µMama-Profiled's weighted speedups normalized to Bandit for each trace mix used in the 8-core experiments. We see that, compared to µMama in Figure 10b, µMama-Profiled improves the average Weighted Speedup. Further, relative to µMama, µMama-Profiled reduces the number of workloads with slowdowns over Bandit by 47%. This is because it eliminates cases where µMama prioritizes the wrong cores. For data-center applications that frequently run at large scale, this improvement may be worth the cost of profiling [29]. However, for more general workloads, this is probably less useful, given the up-front cost and software involvement.

7 Future Work

In this work, we focused on a system of Bandit-based L2 prefetchers. This scenario has been helpful for understanding the problems with multiple RL agents. Other scenarios, such as systems with Bandit-based L1 *and* L2 prefetchers, may require changes to µMama's design. If the two levels are controlled by separate agents, the JAV will need to contain {L1 pref, L2 pref} pairs, instead of just the L2 actions, and the global timestep may need to be revised to account for the different miss frequencies of the two levels. If, however, the L1 and L2 prefetchers are jointly controlled by a *single* Bandit agent, µMama's design will hardly change.

µMama's applicability to non-Bandit prefetchers depends on their particular design. For example, Pythia [5] and RLOP [19] use RL to determine offsets instead of degrees. Without any control knob for aggressiveness, µMama may not be directly applicable. On the other hand, it should be easier to apply µMama to designs such as RL-CoPref [58] which, like Bandit [17], use RL to control the degrees of a set of prefetchers. The larger state space of RL-CoPref would still require some modifications: the actions stored in the JAV

cache may need to represent limits on prefetcher aggressiveness, rather than a particular prefetcher configuration as done here.

Other non-RL prefetchers can also be managed via μMama in the same way as the next-line, stream, and stride prefetchers are used in this work. In systems that already incorporate control knobs for prefetcher aggressiveness (e.g., Arm Neoverse V2 [33]), a design like μMama can be used to control these knobs in a way that accounts for the characteristics of different workloads in the system.

8 Related Work

Prior works have also investigated the use of reinforcement learning in multicore environments. Jain et al. [24] use hierarchical Q-Learning to optimize system EDP by controlling core DVFS, uncore DVFS, and LLC partitioning. In addition to the differences in optimization targets and action spaces, their learning algorithm restricts itself to only changing one variable at a time, which can slow down the learning process substantially in large systems. Jalili and Erez [25] use a deep learning approach that combines a system-level shared network frontend with per-prefetcher "branches." Their deep network requires offline training, and its network is fixed at runtime, limiting adaptability in unseen environments.

More broadly, there has been a wide range of works targeting the problem of managing resources in multicore systems, especially in regards to prefetching. Ebrahimi et al. [13] used a global throttling mechanism to reduce the bandwidth used by inaccurate local prefetchers and, in a later work [12], showed how prioritizing certain demand accesses over others can help to overcome performance and fairness issues caused by over-eager prefetching in multicores. The second of these works is orthogonal to ours, and the former is largely heuristic. Although well-reasoned, they do not directly optimize for application performance like μMama's reward function does, meaning they must be reformulated if a different optimization target is required. Similarly, Arm's Neoverse V2 processors limit the number of outstanding transactions when the network reports high load, but do not directly account for the tradeoff between local and system-level performance [33, 34]. AREP [28] and Limoncello [23] both use runtime monitors to determine when to enable or disable hardware prefetchers, and provide software prefetching to compensate for lost hardware prefetching. Both of these works require that workloads be compiled with particular toolchains to ensure compatibility with the software prefetching. This limits their utility on public systems, where users may be unwilling to change their development processes.

9 Conclusion

We observed that, in multicore systems, uncoordinated RL prefetchers motivated by their own cores' performance will naturally converge to contentious, and often globally-suboptimal, prefetching policies. Based on this finding, we proposed μMama, a light-weight supervisor of distributed multi-armed bandit agents that learns and applies globally-optimized prefetching actions. μMama's simple RL design allows it to be used to target a range of system-level objectives, simplifying the process of exploring performance tradeoffs.

We evaluated μMama's effectiveness on multicore systems with various memory bandwidths. In an 8-core multicore, the prefetching policies learned by μMama outperform those of independently-operating agents by an average of 2.1% when optimizing for throughput, and by an average of 10.4% when optimizing for fairness. μMama performs better in systems that are more bandwidth constrained, as well as when profiles of the workloads are provided. Finally, we showed the flexibility of μMama under alternative reward functions. We conclude that μMama is a significant improvement over Bandit for prefetching in multicore systems.

Acknowledgments

This work was supported by NSF with grants CCF 2107470, CCF 2316233, and Graduate Research Fellowship DGE 21-46756; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by the IBM-Illinois Discovery Accelerator Institute. This work made use of computing resources of Illinois Computes, which is supported by the University of Illinois Urbana-Champaign (UIUC); and the Illinois Campus Cluster, operated in conjunction with NCSA.

References

[1] Alaa R. Alameldeen, Seth Pugsley, Michael Ferdman, and Mina Abbasi Dinani. 2019. The 3rd Data Prefetching Championship. https://dpc3.compas.cs.stonybrook.edu/

[2] Stefano V. Albrecht, Filippos Christianos, and Lukas Schäfer. 2024. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press. https://www.marl-book.com

[3] Mohammad Bakhshalipour, Mehran Shakerinava, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Bingo Spatial Data Prefetcher. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 399–411. https://doi.org/10.1109/HPCA.2019.00053

[4] Rajeev Balasubramonian, Andrew B. Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Trans. Archit. Code Optim.* 14, 2, Article 14 (June 2017), 25 pages. https://doi.org/10.1145/3085572

[5] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. 2021. Pythia: A Customizable Hardware Prefetching Framework Using Online Reinforcement Learning. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1121–1137. https://doi.org/10.1145/3466752.3480114

[6] Dimitri Bertsekas. 2019. *Reinforcement learning and optimal control*. Vol. 1. Athena Scientific.

[7] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[8] Ilai Bistritz and Nicholas Bambos. 2020. Cooperative multi-player bandit optimization. *Advances in Neural Information Processing Systems* 33 (2020), 2016–2027.

[9] Intel Corporation. 2021. Intel® Xeon® Platinum 8380 Processor. https://www.intel.com/content/www/us/en/products/sku/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz/specifications.html

[10] Yi Ding, Nikita Mishra, and Henry Hoffmann. 2019. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 39–52. https://doi.org/10.1145/3307650.3326633

[11] Quang Duong, Akanksha Jain, and Calvin Lin. 2024. A New Formulation of Neural Data Prefetching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1173–1187.

[12] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. 2011. Prefetch-aware shared-resource management for multi-core systems. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. 141–152. https://doi.org/10.1145/2000064.2000081

[13] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd*

Annual IEEE/ACM International Symposium on Microarchitecture (New York, New York) *(MICRO 42)*. Association for Computing Machinery, New York, NY, USA, 316–326. https://doi.org/10.1145/1669112.1669154

[14] Stijn Eyerman and Lieven Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28, 3 (2008), 42–53. https://doi.org/10.1109/MM.2008.44

[15] Alessandro Fogli, Bo Zhao, Peter Pietzuch, Maximilian Bandle, and Jana Giceva. 2024. OLAP on Modern Chiplet-Based Processors. *Proc. VLDB Endow.* 17, 11 (July 2024), 3428–3441. https://doi.org/10.14778/3681954.3682011

[16] Abdoulaye Gamatié, Xin An, Ying Zhang, An Kang, and Gilles Sassatelli. 2019. Empirical model-based performance prediction for application mapping on multicore architectures. *Journal of Systems Architecture* 98 (2019), 1–16. https://doi.org/10.1016/j.sysarc.2019.06.001

[17] Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 698–713. https://doi.org/10.1145/3613424.3623780

[18] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] https://arxiv.org/abs/2210.14324

[19] Yan Huang and Zhanyang Wang. 2024. RLOP: A Framework Design for Offset Prefetching Combined with Reinforcement Learning. In *Proceedings of the 13th International Conference on Computer Engineering and Networks*, Yonghong Zhang, Lianyong Qi, Qi Liu, Guangqiang Yin, and Xiaodong Liu (Eds.). Springer Nature Singapore, Singapore, 90–99.

[20] Zhiming Huang and Jianping Pan. 2023. A near-optimal high-probability swap-regret upper bound for multi-agent bandits in unknown general-sum games. In *Proceedings of the Thirty-Ninth Conference on Uncertainty in Artificial Intelligence* (Pittsburgh, PA, USA) *(UAI '23)*. JMLR.org, Article 86, 11 pages.

[21] Zhiming Huang and Jianping Pan. 2024. Distributed Learning of Unknown Games for HetNet Selection. *IEEE Transactions on Network Science and Engineering* (2024), 1–13. https://doi.org/10.1109/TNSE.2024.3354792

[22] Intel Corporation. 2023. Intel® Xeon® Platinum 8490H Processor. https://www.intel.com/content/www/us/en/products/sku/231747/intel-xeon-platinum-8490h-processor-112-5m-cache-1-90-ghz/specifications.html

[23] Akanksha Jain, Hannah Lin, Carlos Villavieja, Baris Kasikci, Chris Kennelly, Milad Hashemi, and Parthasarathy Ranganathan. 2024. Limoncello: Prefetchers for Scale. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 577–590. https://doi.org/10.1145/3620666.3651373

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

[25] Majid Jalili and Mattan Erez. 2022. Managing Prefetchers With Deep Reinforcement Learning. *IEEE Computer Architecture Letters* 21, 2 (2022), 105–108. https://doi.org/10.1109/LCA.2022.3210397

[26] Haozhe Jiang, Qiwen Cui, Zhihan Xiong, Maryam Fazel, and Simon S. Du. 2023. A Black-box Approach for Non-stationary Multi-agent Reinforcement Learning. arXiv:2306.07465 [cs.LG]

[27] D.A. Jimenez and C. Lin. 2001. Dynamic branch prediction with perceptrons. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 197–206. https://doi.org/10.1109/HPCA.2001.903263

[28] Muneeb Khan, Michael A. Laurenzanoy, Jason Marsy, Erik Hagersten, and David Black-Schaffer. 2015. AREP: Adaptive Resource Efficient Prefetching for Maximizing Multicore Performance. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 367–378. https://doi.org/10.1109/PACT.2015.35

[29] Tanvir Ahmed Khan, Muhammed Ugur, Krishnendra Nathella, Dam Sunwoo, Heiner Litz, Daniel A Jiménez, and Baris Kasikci. 2022. Whisper: Profile-guided branch misprediction elimination for data center applications. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 19–34.

[30] Tejas D. Kulkarni, Karthik R. Narasimhan, Ardavan Saeedi, and Joshua B. Tenenbaum. 2016. Hierarchical deep reinforcement learning: integrating temporal abstraction and intrinsic motivation. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3682–3690.

[31] Saurabh Kumar, Pararth Shah, Dilek Hakkani-Tur, and Larry Heck. 2017. Federated Control with Hierarchical Multi-Agent Deep Reinforcement Learning. arXiv:1712.08266 [cs.AI]

[32] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press.

[33] Arm Limited. 2022. Arm® Neoverse™ V2 Core Technical Reference Manual. https://developer.arm.com/documentation/102375/latest/

[34] Arm Limited. 2025. Use of CBusy. https://developer.arm.com/documentation/109252/0101/MPAM-System-Guidance-for-Infrastructure/Use-of-CBusy

[35] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. 2024. CHROME: Concurrency-Aware Holistic Cache Management Framework with Online Reinforcement Learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture*. https://doi.org/10.1109/HPCA57654.2024.00090

[36] Anup Menon and John S Baras. 2013. A distributed learning algorithm with bit-valued communications for multi-agent welfare optimization. In *52nd IEEE Conference on Decision and Control*. IEEE, 2406–2411.

[37] Samuel Naffziger, Kevin Lepak, Milam Paraschou, and Mahesh Subramony. 2020. 2.2 AMD Chiplet Architecture for High-Performance Server and Desktop Products. In *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*. 44–45. https://doi.org/10.1109/ISSCC19947.2020.9063103

[38] John F Nash Jr. 1950. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences* 36, 1 (1950), 48–49.

[39] Santiago Ontañón. 2017. Combinatorial multi-armed bandits for real-time strategy games. *J. Artif. Int. Res.* 58, 1 (Jan. 2017), 665–702.

[40] Martin J Osborne. 2004. An Introduction to Game Theory. *Oxford University Press* 2 (2004), 672–713.

[41] Marie Ossenkopf, Mackenzie Jorgensen, and Kurt Geihs. 2019. Hierarchical multi-agent deep reinforcement learning to develop long-term coordination. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing* (Limassol, Cyprus) *(SAC '19)*. Association for Computing Machinery, New York, NY, USA, 922–929. https://doi.org/10.1145/3297280.3297371

[42] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. *SIGARCH Comput. Archit. News* 43, 3S (jun 2015), 285–297. https://doi.org/10.1145/2872887.2749473

[43] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK.

[44] Soheil Salehi and Ronald F. DeMara. 2015. Energy and area analysis of a floating-point unit in 15nm CMOS process technology. In *SoutheastCon 2015*. 1–5. https://doi.org/10.1109/SECON.2015.7132972

[45] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 291–303. https://doi.org/10.1109/HPCA51647.2021.00033

[46] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying Deep Learning to the Cache Replacement Problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 413–425. https://doi.org/10.1145/3352460.3358319

[47] Zhan Shi, Akanksha Jain, Kevin Swersky, Milad Hashemi, Parthasarathy Ranganathan, and Calvin Lin. 2021. A hierarchical neural model of data prefetching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 861–873. https://doi.org/10.1145/3445814.3446752

[48] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. *SIGPLAN Not.* 48, 8 (Feb. 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[49] Allan Snavely and Dean M. Tullsen. 2000. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.* 35, 11 (Nov. 2000), 234–244. https://doi.org/10.1145/356989.357011

[50] Don Soltis and Stephen Robinson. 2023. The Next Generation of High Performance, Energy-Efficient Computing: Intel® Xeon® Processors Built on Efficient-Core . In *2023 IEEE Hot Chips 35 Symposium (HCS)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–16. https://doi.org/10.1109/HCS59251.2023.10254709

[51] Kyunghwan Son, Daewoo Kim, Wan Ju Kang, David Hostallero, and Yung Yi. 2019. QTRAN: Learning to Factorize with Transformation for Cooperative Multi-Agent Reinforcement Learning. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 5887–5896. http://proceedings.mlr.press/v97/son19a.html

[52] Standard Performance Evaluation Corporation. 2006. SPEC CPU 2006. https://www.spec.org/cpu2006/

[53] Standard Performance Evaluation Corporation. 2017. SPEC CPU 2017. https://www.spec.org/cpu2017/

[54] A. Stillmaker and B. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIntegration.TechScale/.

[55] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning, An Introduction*. Cambridge University Press.

[56] Wenjun Wang and Wei-Ming Lin. 2018. Real-time physical register file allocation with neural networks for simultaneous multi-threading processors. *Int. J. High Perform. Syst. Archit.* 8, 3 (January 2018), 146–158. https://doi.org/10.1504/ijhpsa.

2018.100714

[57] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[58] Huijing Yang, Juan Fang, Xing Su, Zhi Cai, and Yuening Wang. 2024. RL-CoPref: a reinforcement learning-based coordinated prefetching controller for multiple prefetchers. *The Journal of Supercomputing* (2 2024). https://doi.org/10.1007/s11227-024-05938-9

[59] Siavash Zangeneh, Stephen Pruett, Sangkug Lym, and Yale N. Patt. 2020. BranchNet: A Convolutional Neural Network to Predict Hard-To-Predict Branches. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 118–130. https://doi.org/10.1109/MICRO50266.2020.00022

[60] Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. 2020. Learning Implicit Credit Assignment for Cooperative Multi-Agent Reinforcement Learning. *arXiv preprint arXiv:2007.02529* (2020).

[61] Yang Zhou, Fang Wang, Zhan Shi, and Dan Feng. 2022. An End-to-End Automatic Cache Replacement Policy Using Deep Reinforcement Learning. *Proceedings of the International Conference on Automated Planning and Scheduling* 32, 1 (Jun. 2022), 537–545. https://doi.org/10.1609/icaps.v32i1.19840

[62] Anastasios Zouzias, Kleovoulos Kalaitzidis, and Boris Grot. 2021. Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. arXiv:2106.13429 [cs.LG]