Practical Online Reinforcement Learning for Microprocessors With Micro-Armed Bandit

Gerasimos Gerogiannis and Josep Torrellas, *University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA*

Although online reinforcement learning (RL) has shown promise for microarchitecture decision making, processor vendors are still reluctant to adopt it. There are two main reasons that make RL-based solutions unattractive. First, they have high complexity and storage overhead. Second, many RL agents are engineered for a specific problem and are not reusable. In this work, we propose a way to tackle these shortcomings. We find that, in diverse microarchitecture problems, only a few actions are useful in a given time window. Motivated by this property, we design* Micro-Armed Bandit *(or* Bandit *for short), an RL agent that is based on the low-complexity Multi-Armed Bandit algorithms. We show that Bandit can match or exceed the performance of more complex RL and non-RL alternatives in two different problems: data prefetching and instruction fetch thread selection in simultaneous multithreaded processors. We believe that Bandit's simplicity, reusability, and small storage overhead make online RL more practical for microarchitecture.

In recent years, machine learning (ML) has received widespread attention in the architecture community due to its ability to efficiently model complex patterns, optimize system operation, and adapt to dynamic workloads. ML has found many applications in processor microarchitecture, ranging from predictors, prefetchers, and cache replacement policies to resource management and control.

Reinforcement learning (RL) is a subclass of ML algorithms that targets *action selection* problems. Commonly, an RL agent interacts with its environment by trying different actions and receiving feedback in the form of a reward. The goal of the RL agent is to maximize the accumulated reward over the long term. During execution, the RL agent performs both exploration (i.e., trying new actions) and exploitation (i.e., running with the best action). Exploration is necessary because it helps the agent better understand the dynamics of its environment. The duration of an agent's interaction with its environment is called an *RL episode*.

0272-1732 © 2024 IEEE
Digital Object Identifier 10.1109/MM.2024.3408719
Date of publication 5 June 2024; date of current version 14 August 2024.

Compared to traditional ML, online RL eliminates the need for offline data collection. RL provides better adaptability and generalizes to environment configurations not encountered before. For these reasons, online RL is an attractive solution for action selection problems in microarchitecture.

Hardware agents that employ RL[2,7,8] typically decompose the environment into a set of states, where the RL agent tries to discover the best actions that maximize its reward. Such actions cause transitions between states. Although this approach is effective, it has high complexity as it introduces the need to track action values and transition probabilities for many different states. In the resource-constrained environment of core microarchitecture, this results in significant storage overhead.

A second shortcoming of most RL agents is that they are engineered for a specific problem and are not reusable. Designing, validating, and integrating a new RL hardware agent in a processor every time a new potential use case emerges is very costly. In summary, processor vendors have been reluctant to adopt RL-based microarchitecture agents due to their high complexity and lack of reusability.

To address these problems, our work introduces a *lightweight* and *reusable* hardware RL agent. It is based on our observation that, in some microarchitecture decision-making problems, only a small fraction of the action space is useful in a given time window. We refer to this property as *temporal homogeneity in the action space*. We show that, when this property is present, the problem can be roughly modeled using a simple RL model that has a single RL state.

Based on this insight, we explore the effectiveness of *Multi-Armed Bandit* (MAB) algorithms,[11] which are the RL flavor with the lowest complexity. Due to their simplicity, they are especially suitable for decision making in highly area-constrained settings like a processor's pipeline. We design an RL agent that implements the MAB algorithms in hardware. We call our agent *Micro-Armed Bandit* or *Bandit*.

We showcase Bandit for two decision-making problems that have sufficient temporal homogeneity in their action space to be tackled with MABs. First, we use Bandit to orchestrate simple and widely adopted non-RL level-2 (L2) data prefetchers, i.e., next-line, stride, and stream prefetchers. Second, we use Bandit to control the instruction fetch policy[3,12] in simultaneous multithreaded (SMT) processors.

Our evaluation shows that Bandit is able to match or outperform the performance of state-of-the-art RL and non-RL agents with a remarkably low storage overhead of only 100 B. We believe that Bandit's simplicity, reusability, and small storage overhead make online RL more practical for processor microarchitecture.

RL MODELING AND TEMPORAL HOMOGENEITY

RL Modeling

There are multiple RL problem formulations, which differ in how they model the environment. They have different levels of complexity. Three of the most important ones are Markov decision process-based RL (MDP-RL), contextual bandits (CBs), and MABs.[10] The first two decompose the environment into a large number of states; the third one uses a single state and, is the one with the least complexity. The level of complexity of a formulation is directly correlated with the implementation complexity of the RL agent in hardware. In this work, we focus on MABs.

MABs[4,11] use a single state to model the whole environment. Their goal is to identify the single best action that yields the highest reward while minimizing the time spent trying suboptimal actions. In contrast to heuristic-based approaches, MAB algorithms try to balance exploration and exploitation in a statistically optimal way. MABs have significantly lower complexity than MDP-RL and CB because there is a single state and, therefore, only a single value needs to be tracked per action. Although MABs' simplicity makes them attractive for resource-constrained settings, their application to microarchitecture has not been investigated.

Temporal Homogeneity in the Action Space

MABs can be applied to the microarchitecture domain if a certain property is present. Consider a scenario where the same action is repeatedly optimal for a large-enough time period within an RL episode. We refer to this property as *temporal homogeneity in the action space*. In this case, as the same action is temporally optimal regardless of the environment's state, the whole state space can collapse into a single state. This property enables the effective use of the simple MAB RL flavor.

Figure 1 provides an example of a (fully) temporal homogeneous and a temporal heterogeneous action space. In the figure, the whole action space consists of two actions: *Ax* and *Ay*. Figure 1(a) shows a fully temporal homogeneous action space. It shows the sequence of optimal actions over time for two different RL episodes (e.g., the execution of two different benchmarks). In episode 1, action *Ax* is always optimal, while in episode 2, *Ay* is always optimal. On the other hand, Figure 1(b) shows an environment with a temporal heterogeneous action space. The optimal action changes rapidly with time during the same episode.

For MABs to be effective for action selection in a microarchitecture problem, the following two conditions should be met:

1) The problem must be characterized by *temporal homogeneity in the action space*. This means that the same action should be repeatedly optimal for a large-enough time period inside an RL episode.
2) The problem must offer *adaptation opportunities*. This means that different actions should be optimal across different RL episodes or during different phases of the same RL episode (e.g., across different benchmarks or across benchmark phases).

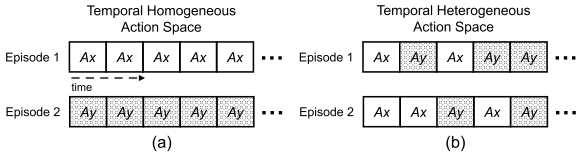

FIGURE 1. (a) Temporal homogeneous and (b) heterogeneous action spaces.

APPLICABILITY OF MABs TO TWO DIFFERENT MICROARCHITECTURE PROBLEMS

As a proof of concept for the applicability of MABs across diverse microarchitecture decision-making problems, we focus on two different use cases. These problems 1) offer adaptation opportunities and 2) are characterized by sufficient temporal homogeneity to be tackled with MABs. We briefly describe them next.

Data Prefetching

In data prefetching, it is well known that different prefetching degrees and prefetching offsets work best for different scenarios. Thus, prefetching offers adaptation opportunities. However, characterizing the temporal homogeneity of the prefetching action space requires identifying the optimal prefetching action at every point in the program. Unfortunately, this is a nontrivial task. Consequently, we use as a proxy the actions taken by Pythia,[2] a spatial MDP-RL prefetcher that shows high performance. Pythia supports 16 different offsets and four different degrees, for a total of 64 actions. By profiling the frequency of the Pythia actions in SPEC traces for a duration of 1 billion instructions, we found that the most selected action in each application accounts for 60% of the total selections, while the second-most selected accounts for 15% of the total action selections. In other words, for a duration of 1 billion instructions, 3% of the action space accounts for 75% of the total action selections. Note that the most selected action is different in each application.

The data suggest that the prefetching action space has high temporal homogeneity but not a perfect one. Also recall that MABs cannot distinguish between environment states (e.g., between cache line addresses or between PCs). Hence, using an MAB agent to directly select a single best degree and offset for all the cache lines would often not work. Instead, we leverage the following observation: conventional and widely adopted non-RL prefetchers such as stride and stream prefetchers can already distinguish between environment states to some extent. For example, the PC-based stride prefetcher can concurrently support different offsets for different PCs. Thus, instead of using the MAB agent to directly determine the prefetch degrees and offsets, we use it as a *coordinator* of various conventional lightweight prefetchers.

SMT Instruction Fetch Thread Selection

An important factor that affects performance in SMT processors is the policy that determines from which thread to fetch instructions. In Tullsen et al.,[12] different *fetch-priority* policies were investigated. The policies give priority to threads based on different microarchitectural metrics. For example, branch count prefers to fetch from threads that have fewer branch instructions in the reorder buffer (ROB), while ICount (IC) favors threads with fewer occupied entries in the instruction queue (IQ).

Choi and Yeung[3] introduced an adaptive mechanism for the *fetch gating* of threads. The mechanism dynamically sets a per-thread occupancy threshold for hardware structures using a hill climbing algorithm. If the occupancy of a thread in the IQ, the integer register file, or the ROB exceeds this threshold (which is the same for all the structures) the thread is fetch gated. The best threshold is determined by trial epochs, dynamically increasing the allowed entries for one thread by δ units at the expense of the other threads. An interesting property of the optimal thresholds is that they are mostly *temporally stable*, as shown in Choi and Yeung.[3]

In our 2023 symposium paper,[5] we adapt both the fetch-gating mechanism and the fetch-priority mechanism. In particular, we examine different hardware structures whose high use can trigger fetch gating, and different priority policies to use for the remaining non-fetch-gated threads. We refer to the resulting compound mechanism for instruction fetch thread selection as *fetch-priority-gating* (*PG*) *policy*. We tested 64 of these policies, each with different pipeline structures affecting fetch gating and/or a different priority policy for the non-fetch-gated threads, for different two-threaded SPEC mixes. We observed that different fetch PG policies work best in different application mixes. Specifically, a static oracle that predicts the optimal fetch PG policy for each benchmark mix can offer speedups up to 30% over the policy proposed by Choi and Yeung.[3] This reveals that SMT instruction fetch thread selection offers adaptation opportunities, which can be exploited with MABs.

MAB ALGORITHMS

In MAB algorithms, an *arm* refers to a specific action available to the MAB agent, while a *bandit step* is

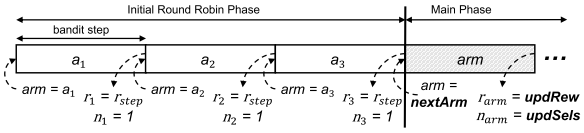

FIGURE 2. An overview of the MAB algorithm.

defined as the time duration for which the agent is idle waiting to observe the outcome of its previous arm selection. Further, r_{step} is the reward received at the end of a bandit step. For every arm i, two variables are collected: the average reward r_i that previous selections of this arm have yielded, and the number of times n_i that this arm has been selected in the past.

Figure 2 provides a general overview of an MAB algorithm. It begins with an initial round-robin phase, during which all the arms are tried once. The figure assumes a total of three arms. For each arm i, r_i is set to the r_{step} received during that arm's first step, and n_i is set to one. Then, the main phase of the algorithm begins, which lasts for as long as the agent keeps interacting with the environment. It consists of three basic functionalities, which depend on the specific MAB algorithm used. Those are *nextArm()*, which selects the next arm to be tried; *updSels(arm)*, which updates the number of selections n_i for the currently selected arm i and potentially other arms; and *updRew*(r_{step}), which updates the reward r_i for the currently selected arm i after the bandit step is over and the r_{step} has been collected.

In this work, we consider three different MAB algorithms.[10] First, we consider ϵ-greedy, which implements a random exploration. This means that, when it explores, it does not consider the prior reward or the selection frequency of different arms. Second, we consider the *upper confidence bound* (*UCB*) algorithm, which implements an intelligent exploration that accounts for both past rewards and selection frequencies of different arms. Finally, we consider the *discounted UCB* (*DUCB*) algorithm, which is a UCB extension that gradually forgets past rewards. Thus, it is especially useful for dynamic scenarios.

In our 2023 symposium paper,[5] we provide full details of the *nextArm()*, *updSels(arm)*, and *updRew*(r_{step}) functions for the three different MAB algorithms we consider. In addition, we provide some microarchitecture-inspired modifications to the algorithms that result in higher performance in practical scenarios.

THE MICRO-ARMED BANDIT

Microarchitecture

We design a lightweight microarchitectural agent that implements in hardware the three MAB algorithms discussed earlier. We call the agent *Micro-Armed Bandit* or *Bandit* for short. Bandit has a simple microarchitecture, as shown in Figure 3. It consists of two tables, an arithmetic unit, and some control logic. The two tables are the *nTable* and the *rTable*, and each has as many entries as the number of arms. For each arm i, the *nTable* contains the number of times that i has been selected so far (n_i), while the *rTable* contains the current value of its reward (r_i). The arithmetic unit executes the arithmetic operations in the *nextArm*, *updSels*, and *updRew* functions in hardware.

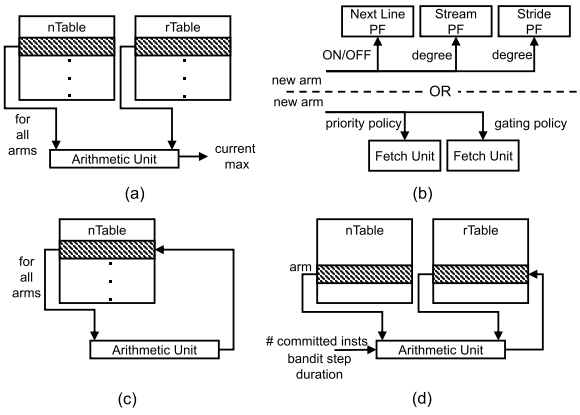

FIGURE 3. The Micro-Armed Bandit microarchitecture. (a) A next arm selection, (b) communicate arm selection, (c) update selections, and (d) update reward. PF: prefetcher.

Figure 3(a) shows the implementation of *nextArm*. The hardware reads the nTable and rTable entries for all the arms, calculates the corresponding potentials, and picks one arm as the new arm. Then, in Figure 3(b), Bandit control logic communicates the arm selection to the controlled entity—in the figure, the L2 data prefetcher or the instruction fetch unit of an SMT core. In the background, Bandit updates the nTable [Figure 3(c)] according to the logic of function *updSels*. Note that, depending on the algorithm, one or potentially all entries are updated. Once the bandit step is over, the Bandit arithmetic unit receives the appropriate hardware performance counters, computes the step's reward (r_{step}), and accumulates it into the rTable entry of the corresponding arm [Figure 3(d)]. The figure assumes that the reward is the core's average instructions per cycle. This process repeats continuously.

Practicality of the Micro-Armed Bandit Design

There are three main reasons that make Bandit a practical addition to CPUs: its low storage overhead, low latency, and modular design.

Bandit has a small storage overhead, which scales linearly with the number of arms. Assuming that r is stored using a single-precision floating-point data type and n is stored using an unsigned integer data type, the storage overhead per arm is 8 B. For the maximum number of arms in our evaluation, which is 11, the total storage overhead is less than 100 B. In comparison, Pythia, which is an MDP-RL agent, requires 24 KB just to store its state-action values.

While the bandit step is in progress, the agent can implement most of the required computations in the background. We estimate that the resulting latency on the critical path is only about 50 cycles. This number is negligible compared to the total duration of a bandit step in our evaluation, which corresponds to 1000 L2 demand accesses for prefetching and 128 K cycles for SMT fetch priority and gating policy selection.

Finally, one difficulty preventing the wider use of RL and ML agents in core microarchitecture is that, typically, a new agent needs to be designed, validated, and integrated in the core every time that a new use case emerges. To overcome this limitation, Bandit is designed in a modular and reusable manner. We envision a future-proof design where the core initially includes a Bandit module for a particular microarchitecture enhancement. In subsequent generations of processors, the design can be reused for other enhancements by only changing the architectural interface of the Bandit module: which specific counters it should read and which modules it should actuate on (e.g., the data prefetcher or the cache controller). There is no need to redesign, revalidate, or reintegrate the module.

PERFORMANCE EVALUATION

We evaluate Bandit for two distinct use cases. First, we use it to control the degree and type of a set of lightweight L2 data prefetchers: next line, stream, and PC-based stride. Second, we use it to control the fetch priority and gating policies of SMT threads.

Our simulation-based evaluation across a wide range of benchmark suites (SPEC, PARSEC, Ligra, and CloudSuite) suggests that Bandit is effective. In the data prefetching use case, Bandit slightly outperforms non-RL prefetchers Bingo[1] and multi-lookahead offset prefetcher[9] by 2.3%–2.6%, and attains similar performance as the state-of-the-art RL prefetcher Pythia,[2] but does so with the dramatically lower storage requirement of only 100 bytes.

For the SMT use case, Bandit offers a 2.2% geometric mean speedup over the fixed policy proposed by Choi and Yeung[3] and a 7% geometric mean speedup over IC.[12]

FUTURE RESEARCH DIRECTIONS AND OPPORTUNITIES

We believe that Bandit opens up a variety of future research directions and opportunities. In our 2023 symposium paper,[5] we describe potential Bandit extensions that could trade off a slightly higher storage overhead for higher performance. In this section, we outline some additional research directions.

Applying Bandit to Other Use Cases and Architectures

Bandit's effectiveness is not limited to data prefetching and instruction fetching in SMT cores. For example, Bandit can be used to control, with low overhead, a hybrid predictor (i.e., a predictor composed of multiple predictors of different characteristics) for potentially many microarchitectural mechanisms. Examples are branch, value, or off-chip load prediction. Further, it can be used to control, concurrently, multiple microarchitecture mechanisms, e.g., the combination of data prefetching and cache power management. Bandit can inspire researchers to replace heuristic approaches by the lightweight RL approach of Bandit. In "Guidelines for Using Bandit in Microarchitecture Problems," we give some guidelines for using Bandit in other microarchitecture problems.

In addition, Bandit is not limited to online decision making in CPUs. Investigating potential use cases in GPUs or in domain-specific architectures with configurable knobs[6] is an interesting direction.

Discovering New Use Cases

In this work, we present one new microarchitecture mechanism that can be optimized with RL: the combination of instruction fetch gating and prioritization in SMT. Other researchers can adopt Bandit for other microarchitecture problems. Bandit can trigger a search for more opportunities for customization in microarchitecture problems that have not received attention, due mainly to the prohibitive cost of high-complexity RL agents.

Formalizing the Design of ML Agents for Microarchitecture

In this work, we introduce the property of temporal homogeneity in the action space of microarchitectural mechanisms such as data prefetching and instruction fetching. We showed that this property is key to determining the complexity of RL-based designs. Mechanisms with high temporal homogeneity can be optimized with a low-complexity RL agent. To the best of our knowledge, this is the first work that tackles ML agent design for microarchitecture in a principled

manner. Our work can inspire analogous principled approaches for other microarchitecture problem classes such as classification and modeling. Overall, we believe that our work can guide researchers to design new ML-based agents or rethink the design of existing ones for simplicity.

CONCLUDING REMARKS

In this work, we proposed the Micro-Armed Bandit, the first microarchitecture design based on MAB algorithms, the simplest form of RL. We introduced the concept of temporal homogeneity in the action space, a property necessary for tackling microarchitecture problems with low-cost RL. With a storage overhead of only 100 B, Bandit can match or exceed the performance of more complex RL and non-RL alternatives in two different problems: data prefetching and SMT instruction fetch thread selection.

In addition, we provided guidelines for applying Bandit to other problems, and discussed how our findings can guide the design of future ML agents for microarchitecture. Overall, we believe that Bandit's simplicity and modular design can make online RL more attractive to microarchitects.

GUIDELINES FOR USING BANDIT IN MICROARCHITECTURE PROBLEMS

Here we provide guidelines to use Bandit in microarchitecture problems. We discuss 1) a method for determining Bandit's applicability for a specific problem, 2) practical ways to maximize Bandit's performance, and 3) an example of a problem ill-suited for Bandit.

PREREQUISITES

Initially, the microarchitecture decision-making problem should be expressed as an RL problem, and its action space (e.g., the possible combinations of prefetching degree and offsets for prefetching) should be defined. After that, a representative set of traces or simpoints from phases of different benchmarks should be collected. This set will be used to 1) characterize the applicability of Bandit and 2) tune Bandit to maximize its performance. We refer to this set of traces as the tune set. In our 2023 paper,[5] we used roughly 10% of our traces/simpoints as the tune set.

Finally, we need a state-of-the-art RL or non-RL decision-making agent for the problem of interest to assist in assessing the effectiveness of a Bandit-based solution. We refer to this agent as the baseline agent. In our 2023 paper,[5] we used the Pythia prefetcher[2] and the fixed SMT policy proposed by Choi and Yeung[3] as our baseline agents.

DETERMINING IF BANDIT IS APPLICABLE TO A MICROARCHITECTURE DECISION-MAKING PROBLEM

After the *action space*, *tune set*, and *baseline agent* have been defined, it is time to determine whether Bandit can be a good choice for the problem of interest. As explained in the "RL Modeling and Temporal Homogeneity" section, for Bandit to be a good choice, the problem should be characterized by *temporal homogeneity in the action space*, and it should offer *adaptation opportunities*.

Characterizing Temporal Homogeneity

Characterizing the temporal homogeneity in the action space of the problem of interest is not trivial because it would require a perfect agent that selects the optimal action at every time step. One could observe the actions of the baseline agent and determine whether they are temporal homogeneous. If they are, this is an indication that a Bandit agent could potentially be used as a low-cost alternative to the baseline agent. Otherwise, one cannot yet declare that the problem lacks temporal homogeneity. This is because the baseline agent is not perfect.

In cases of high but imperfect temporal homogeneity, one can also consider redefining the action space. For example, a possible approach is to use Bandit as a coordinator of lower-level agents or predictors in the same manner that we used it as a coordinator of lightweight prefetchers.

Characterizing Adaptation Opportunities

For Bandit to offer benefits for a given problem, different arms should be optimal for different benchmarks or for different phases of the same

benchmark. In this case, we say that the problem offers *adaptation* opportunities.

To determine whether the problem of interest offers adaptation opportunities, one should execute, for each trace (or simpoint), as many runs as the number of actions (i.e., arms) available to Bandit. In each run, a given arm is applied for the whole duration of the run (i.e., *constant arm run*). After all the *constant arm runs* are completed, one should identify the arm that performed the best on average across all traces of the tune set (i.e., *best constant arm*). In addition, one can compute the resulting performance if the best arm for each trace was selected by an oracle before the run (i.e., *per-trace arm oracle*).

If the performance of the *per-trace arm oracle* is not higher than the one of the *best constant arm*, and assuming that no phase changes occur during the execution of a trace, one can conclude that Bandit is not a good fit for the problem of interest.[a] In addition, if the *per-trace arm oracle* does not outperform the baseline agent, then Bandit will likely not offer performance improvements over the baseline agent.

MAXIMIZING BANDIT PERFORMANCE

After deciding to employ Bandit for a microarchitecture decision-making problem, to get the most out of it, one needs to choose 1) the appropriate MAB algorithm, 2) an effectively pruned action space, 3) and appropriate values for the algorithm hyperparameters (e.g., the bandit step).

Algorithm Selection

In our experience, DUCB performed better than ϵ-greedy and UCB in all our use cases. Therefore, we recommend it as a starting point. Note that MAB algorithms are an active research topic and we could not evaluate all possible options in our paper.[5] We leave the evaluation of Bandit with alternative MAB algorithms for future work.

Arm Pruning

An optimization that we employed to increase Bandit's performance is *arm pruning*. Simply put, we removed arms that were not expected to contribute much to the performance at runtime. To achieve this, we iteratively removed (pruned) one arm and recomputed the performance of the per-trace arm oracle. If the pruning of the arm resulted in performance degradation below a threshold, we discarded that arm. Otherwise, we reinserted the arm in the arm pool. Decreasing the number of arms to be tried at runtime decreases Bandit's exploration cost and can boost performance. However, one limitation of arm pruning is that it is performed based on the contents of the tune set. Thus, there is the danger of removing arms that could potentially be very useful in circumstances not captured in the tune set.

Hyperparameter Tuning

MAB algorithms involve hyperparameters that need to be set to appropriate values to maximize performance. They include the bandit step and a few algorithm-specific parameters (e.g., the exploration probability ϵ in ϵ-greedy).[10] We tuned the hyperparameters using a simple multidimensional grid search, performing tune set runs for each point of the grid (which encodes a combination of hyperparameter values). An indicator of the quality of a specific hyperparameter value set is whether Bandit's average performance gets close, or even exceeds, the one of the per-trace arm oracle.

AN ILL-SUITED PROBLEM FOR BANDIT

To help the reader better understand the limitations of Bandit, we now give an example of a microarchitecture problem that is ill-suited for Bandit. Imagine a scenario where Bandit is used to determine, for a given cache set, the way that should be replaced in the event of an eviction. The action space has as many actions as the number of ways of the set, and each arm encodes the ID of the way to be replaced. It is easy to show that this problem is not characterized by temporal homogeneity. If we assumed temporal homogeneity, we would repeatedly select the same way of the set for replacement during a phase of execution. This would lead to a last-in, first-out replacement policy, which is known to deliver subpar performance. It is well known that determining the optimal way to evict requires contextual information that conveys future knowledge (which cache line will be used further in the future). Thus, using Bandit for this problem is suboptimal.

[a]Note that this could change by redefining the action space.

ACKNOWLEDGMENTS

We thank Shreya Seth and Argyrios Gerogiannis for their help in making this work possible. This research was funded in part by an Intel Transformative Server Architecture gift; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation program sponsored by DARPA; and by U.S. National Science Foundation Grant CCF 2107470, Grant CNS 1956007, and Grant CNS 1763658.

REFERENCES

1. M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proc. IEEE Int. Symp. High Performance Comput. Archit. (HPCA)*, Washington, DC, USA, 2019, pp. 399–411, doi: 10.1109/HPCA.2019.00053.
2. R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA: ACM, 2021, pp. 1121–1137, doi: 10.1145/3466752.3480114.
3. S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing," in *Proc. 33rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2006, pp. 239–251.
4. A. Garivier and E. Moulines, "On upper-confidence bound policies for non-stationary bandit problems," 2008, *arXiv:0805.3415*.
5. G. Gerogiannis and J. Torrellas, "Micro-armed bandit: Lightweight & reusable reinforcement learning for microarchitecture decision-making," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, New York, NY, USA: Assoc. Comput. Machinery, 2023, pp. 698–713, doi: 10.1145/3613424.3623780.
6. G. Gerogiannis, S. Yesil, D. Lenadora, D. Cao, C. Mendis, and J. Torrellas, "SPADE: A flexible and scalable accelerator for SpMM and SDDMM," in *Proc. 50th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA: Assoc. Comput. Machinery, 2023, pp. 1–15, doi: 10.1145/3579371.3589054.
7. E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proc. 35th Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, 2008, pp. 39–50.
8. L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic locality and context-based prefetching using reinforcement learning," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA: Assoc. Comput. Machinery, 2015, pp. 285–297, doi: 10.1145/2749469.2749473.
9. M. Shakerinava, M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Multi-lookahead offset prefetching," in *Proc. Third Data Prefetching Championship*, 2019.
10. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
11. W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, nos. 3–4, pp. 285–294, 1933, doi: 10.1093/biomet/25.3-4.285.
12. D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor," in *Proc. 23rd Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA: Assoc. Comput. Machinery, 1996, pp. 191–202.

GERASIMOS GEROGIANNIS is a Ph.D. candidate of electrical and computer engineering at the University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA. His research interests lie in the area of computer architecture, with an emphasis on modern aspects of domain-specific systems, such as flexibility, integration, scalability, scheduling, and decision making. Gerogiannis received his diploma degree (M.Sc.) in electrical and computer engineering from the University of Patras. Contact him at gg24@illinois.edu.

JOSEP TORRELLAS is the Saburo Muroga Professor of Computer Science at the University of Illinois at Urbana-Champaign, Urbana, IL, 61801, USA, and director of the Semiconductor Research Corporation/DARPA JUMP 2.0 ACE Center for Evolvable Computing. His research interests include parallel computer architectures for performance, energy efficiency, programmability, and security. Torrellas received his Ph.D. degree in electrical engineering from Stanford University. Contact him at torrella@illinois.edu.