Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making

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

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

ABSTRACT

Online Reinforcement Learning (RL) has been adopted as an effective mechanism in various decision-making problems in microarchitecture. Its high adaptability and the ability to learn at runtime are attractive characteristics in microarchitecture settings. However, although hardware RL agents are effective, they suffer from two main problems. First, they have high complexity and storage overhead. This complexity stems from decomposing the environment into a large number of states and then, for each of these states, bookkeeping many action values. Second, many RL agents are engineered for a specific application and are not reusable.

In this work, we tackle both of these shortcomings by designing an RL agent that is both lightweight and reusable across different microarchitecture decision-making problems. We find that, in some of these 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*. Motivated by this property, we design an RL agent based on Multi-Armed Bandit algorithms, the simplest form of RL. We call our agent *Micro-Armed Bandit*.

We showcase our agent in two use cases: data prefetching and instruction fetch in simultaneous multithreaded (SMT) processors. For prefetching, our agent outperforms non-RL prefetchers Bingo and MLOP by 2.6% and 2.3% (geometric mean), respectively, and attains similar performance as the state-of-the-art RL prefetcher Pythia—with the dramatically lower storage requirement of only 100 bytes. For SMT instruction fetch, our agent outperforms the Hill Climbing method by 2.2% (geometric mean).

CCS CONCEPTS

• **Computer systems organization**; • **Computing methodologies** → **Reinforcement learning**;

KEYWORDS

Reinforcement Learning, Microarchitecture, Machine Learning for Architecture, Multi-Armed Bandits, Prefetching, Simultaneous Multithreading

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3623780

ACM Reference Format:
Gerasimos Gerogiannis and Josep Torrellas. 2023. Micro-Armed Bandit: Lightweight & Reusable Reinforcement Learning for Microarchitecture Decision-Making. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3623780

1 INTRODUCTION

Over 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 a myriad of applications in processor microarchitecture, ranging from predictors [10, 37, 72], prefetchers [27, 62], and cache replacement policies [59, 61] to resource management and control [15, 19]. Online Reinforcement Learning (RL) [12, 69] is a subclass of ML algorithms that is particularly well-suited for microarchitecture decision-making problems. In online RL, agents interact with their environment without having to pre-train with an offline dataset. This enables better adaptability and generalization to previously unseen environment configurations.

Hardware agents that employ RL [11, 34, 35, 50, 64, 83–85] typically decompose the environment into a complex 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 processor microarchitecture, this results in significant storage overhead. In addition, most of these RL agents 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.

To address these problems, our work introduces a lightweight and reusable hardware RL agent. We find 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 single RL state.

Based on this insight, we explore the effectiveness of *Multi-Armed Bandit* (MAB) algorithms [54, 73], 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. Specifically, we design an RL agent that implements the *Discounted Upper Confidence Bound* (DUCB)

algorithm [24] 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. Firstly, we use Bandit to orchestrate simple and widely adopted non-RL L2 data prefetchers—i.e., next-line, stride, and stream prefetchers. Secondly, we use it to control the instruction fetch policy in simultaneous multithreaded (SMT) processors [75]. In particular, in the second use case, we introduce a novel design space for SMT instruction fetching that customizes both the thread *fetch priority* [74] and the thread *fetch gating* [17] policies.

Our simulation-based evaluation suggests that Bandit is effective. In the data prefetching use case, for single-core experiments, Bandit outperforms non-RL prefetchers Bingo [7] and MLOP [60] by 2.6% and 2.3% (geometric mean), respectively, and attains similar performance as the state-of-the-art RL prefetcher Pythia [11]—with the dramatically lower storage requirement of only 100 bytes. In addition, Bandit attains good performance in four-core experiments and outperforms Pythia by 2.5% in bandwidth-constrained scenarios. In the SMT instruction fetch use case, Bandit outperforms the Hill Climbing method [17] by 2.2% (geometric mean).

Overall, this paper's contributions are:

- The concept of temporal homogeneity in the action space, which enables low overhead RL for microarchitecture decision-making.
- The adoption of Multi-Armed Bandits as an efficient and reusable learning mechanism for such problems.
- The Micro-Armed Bandit, an RL agent that implements the Discounted Upper Confidence Bound algorithm in hardware and a simulation-based evaluation of its performance for data prefetching and SMT instruction fetching.
- A new design space for SMT instruction fetching that customizes both the thread fetch priority and thread fetch gating policies.

2 BACKGROUND

2.1 Online Reinforcement Learning

RL [12, 69] 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 is to learn the optimal balance between *exploration* (trying new actions) and *exploitation* (selecting the best-known action) to maximize the accumulated reward in the long term. Exploration is necessary for an RL algorithm 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*.

In online RL, agents interact with their environment without having to pre-train with an offline dataset. As a result, online RL has several attractive characteristics [11, 18] for microarchitecture decision-making problems compared to other methods such as Supervised Machine Learning [15, 19, 33, 39, 41, 43, 46] and Formal Control [29, 45, 51–53]. In comparison with traditional ML, online RL eliminates the need for offline data collection. It provides better adaptability and generalizes to environment configurations not encountered before. As an example, consider an agent that is trained offline using the SPEC benchmarks [66, 67]. Ensuring high performance in unseen workloads stemming from graph analytics or cloud applications is non-trivial. Even if the offline dataset is large enough to include a diverse set of benchmarks, there are no guarantees that the agent will adapt successfully to unforeseen system conditions such as power fluctuations. When compared to Formal Control, online RL does not depend on an offline static model of the system and does not require the actions to be linearly correlated with their outcomes [18]. For these reasons, online RL is an attractive solution for action selection problems in microarchitecture.

2.2 RL Problem Formulations

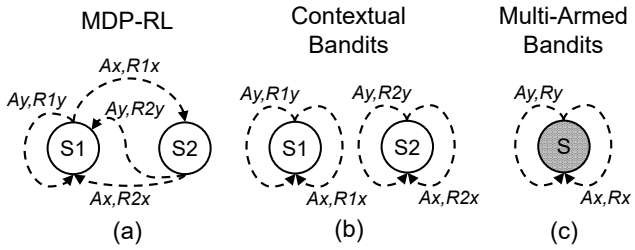

Figure 1: Examples of states and actions in different RL problem formulations. S represents states, A represents actions, and R represents rewards. The arrows illustrate state transitions.

There are multiple RL problem formulations, which differ in how they model the environment. They have different levels of complexity. The level of complexity of a formulation is directly correlated with the implementation complexity of the RL agent in hardware. In this section, we describe three alternative formulations. In addition, we identify a common property of microarchitectural problems that enables the use of the RL flavor with the lowest complexity.

(1) Markov Decision Process-based Reinforcement Learning (MDP-RL): Modeling the environment as a Markov Decision Process (MDP) [9] is the most general and highest-complexity problem formulation. The environment is decomposed into many states and, in each state, different actions yield different rewards. The actions cause non-deterministic transitions between states. Before deciding on an action, the agent considers its current state as well as the states it might transition to after taking that action. Figure 1(a) shows a simple MDP environment with 2 states (*S1* and *S2*). In each state, two actions are possible (*Ax* and *Ay*). After the agent selects an action, it receives a reward (Rij), which depends on both the current state and the selected action.

The high complexity of MDP-RL stems from the requirement of bookkeeping action values and transition probabilities for each distinct state of the environment. Popular algorithms in this category include Q-Learning [80] and SARSA [55, 69]. Numerous works propose hardware agents that model the environment as an MDP. For example, Ipek et al. [34] propose a SARSA-based memory controller that uses the number and characteristics of reads and writes in the transaction queue to encode the state. Cohmeleon [85] proposes a Q-Learning-based agent to orchestrate accelerator coherence in heterogeneous SoCs. The Pythia L2 prefetcher [11] is based on SARSA and can be customized to use different data flow and control flow features to encode the state.

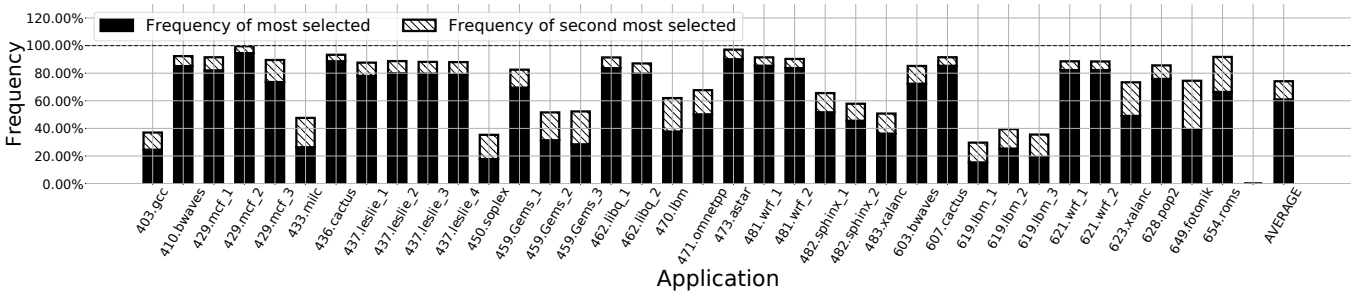

Figure 2: Frequency of the top 2 most selected actions by Pythia in SPEC applications during a trace of 1B instructions.

(2) Contextual Bandits: Modeling a decision-making problem with Contextual Bandits [5, 58] abstracts away the state transition probabilities. State transitions can still occur but they are not influenced by the agent's actions. Figure 1(b) shows a Contextual Bandit environment. The agent can take actions. However, any transition between states is not a consequence of selecting an action. Instead, the system *randomly* transitions between states due to other effects. The agent observes the context associated with each state and tries to determine the best action for a given context to maximize its accumulated reward. Contextual Bandits are a lower complexity problem formulation than MDP-RL but still require the bookkeeping of action values for each state. The most notable application of Contextual Bandits in microarchitecture is the Context Prefetcher [50], which utilizes both hardware and compiler-injected software context to decide on the appropriate prefetching actions.

(3) Multi-Armed Bandits (MABs): Multi-Armed Bandits (MABs) [54, 73] collapse the environment into a single state as displayed in Figure 1(c). The goal is to identify the single best action that yields the highest reward while minimizing time spent trying sub-optimal actions. In MAB terminology, the available actions are referred to as *arms*. MABs have significantly lower complexity than the two previous methods because there is a single state and, therefore, only a single value needs to be tracked per action. Although MABs' low complexity makes them attractive for resource-constrained settings, their application to microarchitecture has not been investigated.

MABs are less powerful than the other two RL problem formulations. However, they can still be applied to the microarchitecture domain. Consider a scenario where the same action is repeatedly optimal for a large enough time period inside an RL episode. In this case, the action is *temporally* optimal regardless of the environment's state and the whole state space can collapse into a single state. We refer to this property as *temporal homogeneity in the action space*.

Figure 3 provides examples of (fully) temporal homogeneous and temporal heterogeneous action spaces. In the figure, the whole action space consists of two actions: *Ax* and *Ay*. Figure 3(a) shows an environment with 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 2 different benchmarks). We see that, in Episode 1, action *Ax* is always optimal, while in Episode 2, *Ay* is always optimal. On the other hand, Figure 3(b) shows an environment with a temporal heterogeneous action space, where the optimal action changes rapidly with time during the same episode. Note that temporal homogeneity does not require the same action to be optimal during the whole episode. Instead, it imposes optimality requirements for a time period long enough so that the MAB agent can detect the appropriate action. As we will see in Section 3, high temporal homogeneity is present in some microarchitectural problems. This property enables the effective use of the simple MAB RL flavor.

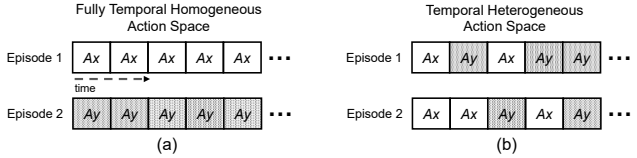

Figure 3: Examples of temporal homogeneous and heterogeneous action spaces.

3 MOTIVATION

This section examines the viability of using MAB algorithms to attack two different microarchitectural action selection problems: (i) data prefetching and (ii) instruction fetching in simultaneous multithreaded (SMT) processors.

3.1 Temporal Homogeneity in Prefetching

In data prefetching, a hardware agent reacts to a request for a line at address X by also prefetching a set of cache lines that are likely to be accessed in the near future. Spatial prefetching algorithms typically specify the lines to be prefetched through degree (d) and offset (o) parameters. The addresses of the prefetched lines are $X + o$, $X + 2 \times o$, ... $X + d \times o$. In this case, prefetching can be formulated as an RL decision-making problem with the action space corresponding to the selection of the prefetch degree and offset. It is well known that different degrees and offsets work best for different scenarios [23, 28, 30, 38, 42]. In this section, we examine the temporal homogeneity of the prefetching action space.

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 non-trivial task. Consequently, we use as a proxy the actions taken by Pythia [11], which is a spatial MDP-RL prefetcher that shows high performance. Pythia supports 16 different offsets and four different degrees, for a total of 64 actions. We profile the frequency of the two most popular Pythia actions in SPEC traces for a duration of 1 billion instructions. Section 6 presents more details about our methodology. Figure 2 displays our findings. On average, 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

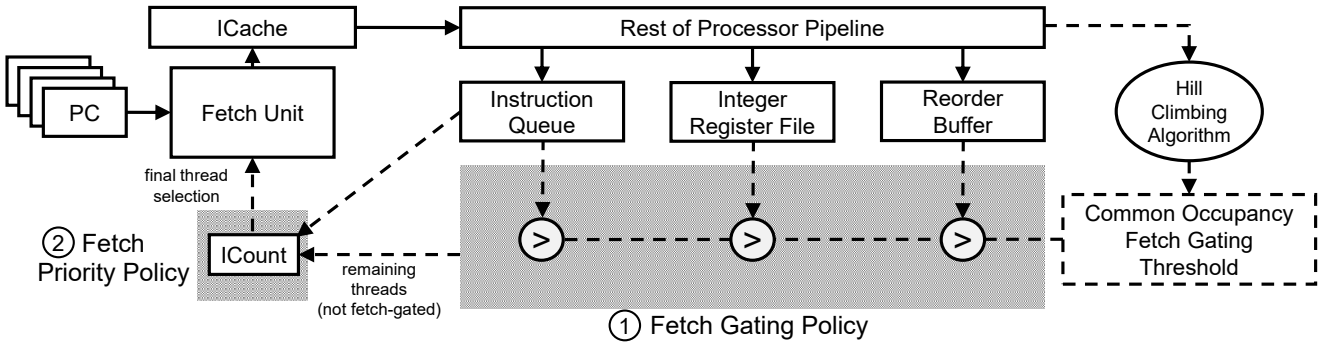

Figure 4: Pipeline structures and fetch priority & gating policy in [17]. Threads that exceed the occupancy threshold in either the Instruction Queue, Register File, or ROB are fetch-gated. The remaining threads are prioritized using ICount.

of Pythia. 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 suggests 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 a 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 the 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—in a manner similar to [38]. Section 5 provides more details about how we employ our agent towards this goal, and Section 7 presents its effectiveness.

3.2 Policies for Fetch Priority and Gating of SMT Threads

An important factor that affects performance in SMT processors [75] is the policy that determines which thread to fetch instructions from. In [74], different *fetch priority* policies were investigated. They give priority to threads based on different microarchitectural metrics: Branch Count (BrC) prefers to fetch from threads that have fewer branch instructions in the Reorder Buffer (ROB); ICount (IC) favors threads with fewer occupied entries in the Instruction Queue (IQ); and LSQCount (LSQC) prioritizes threads with fewer entries in the Load-Store Queue (LSQ). The Round Robin (RR) policy alternates between threads in a round-robin manner without considering any microarchitectural metric.

Choi and Yeung [17] introduce 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 (IRF), 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 [17]. This is consistent with what we observed for prefetching in Section 3.1. Finally, Choi and Yeung determine the priority of the non fetch-gated threads using ICount.

In our work, we call the combination of the thread fetch gating policy and the thread priority policy the fetch *Priority & Gating (PG)* policy. Figure 4 illustrates the pipeline structures and the fetch PG policy used by Choi and Yeung. In the rest of the paper, we refer to the fetch PG policy used by Choi and Yeung as the *Choi* policy.

Because the fetch gating policy uses thresholds, it does not prevent threads with high ILP from utilizing a large number of entries in the shared hardware structures. A thread with high ILP can be allowed to use most of the hardware structure entries, while one with low ILP may be restricted to fewer entries—if this entry distribution maximizes performance.

The fetch gating and the fetch priority policies do not accomplish the same goal; instead, they complement one another. Consider the times when none of the threads is fetch-gated. One still needs to select which thread to fetch from. In addition, in contrast to fetch gating, not all the fetch priority policies are based on measuring the stress that different threads place on hardware structures. Specifically, the Round Robin (RR) and Branch Count (BrC) policies take different approaches.

3.3 Extending the Choi Algorithm

Although the Choi policy is very effective, it has limited adaptability. First, it always uses ICount for the non fetch-gated threads. Second, its fetch gating policy is fixed: it fetch-gates a thread if its occupancy of the IQ, IRF, or ROB exceeds a given threshold. As a result, the fetch gating policy is unaware of the occupancy of other pipeline structures such as the LSQ. In addition, in some workloads, threads may use resources asymmetrically, and it may be best to disregard the occupancy of certain hardware structures. For example, assume a 2-threaded scenario where we limit the occupancy of all three structures to 50%. However, one thread wants to use more entries from the ROB and does not need to use as many entries from the IQ, while the other thread has the opposite behavior. Under the Choi policy, both threads are conservatively fetch-gated, while the pipeline could support more instructions. In this case, one could exclude monitoring the occupancy of the ROB or the IQ (or both) from the fetch gating policy and attain higher performance.

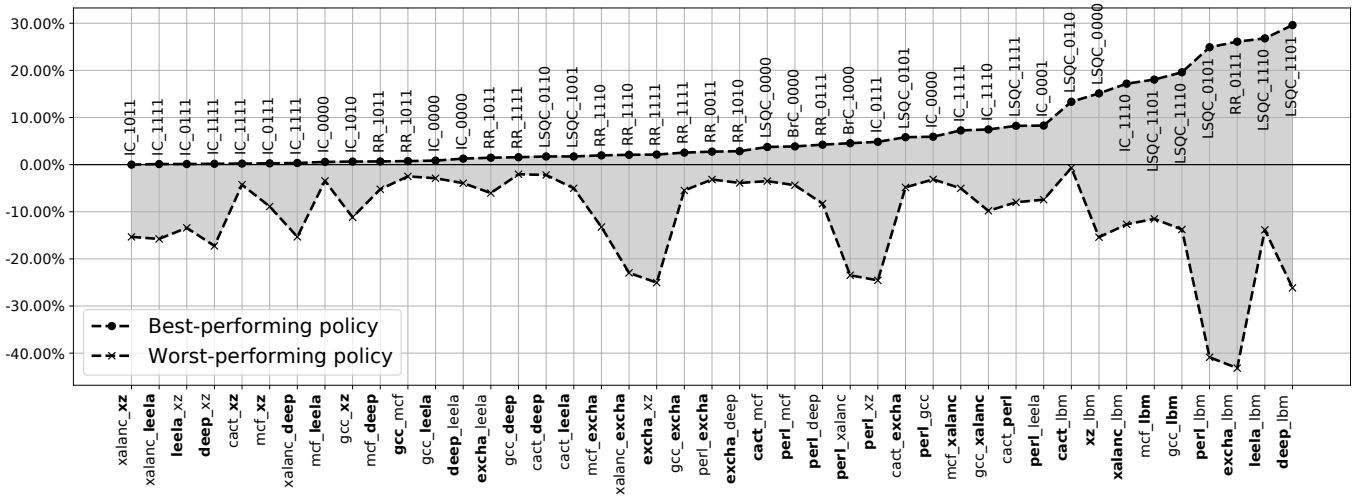

Figure 5: IPC change relative to the Choi policy of the best- and the worst-performing fetch PG policies among the ones we consider for different multithreaded SPEC17 mixes. The policy that performs the best for each mix is added as a label. For the best policy, the thread that committed the most instructions in the mix appears in bold.

As we show next, determining which structures to monitor is an interesting use case for RL-based decision-making mechanisms.

To mitigate the aforementioned limitations, we consider many fetch PG policies beyond the Choi one. Specifically, we consider the four fetch priority policies of Section 3.2 (BrC, IC, LSQC, and RR), and fetch gating policies based on the occupancies of zero or more of the following hardware structures: IQ, LSQ, ROB, and IRF. We represent these fetch PG policies as $X\_b_3b_2b_1b_0$, where X is the fetch priority policy and can be {BrC, IC, LSQC, RR}, and $b_3b_2b_1b_0$ are bits that denote whether the fetch gating policy monitors the occupancy of the IQ, LSQ, ROB, or IRF, respectively.

Table 1 shows some examples of these fetch PG policies. For example, IC_0000 uses ICount as the fetch priority policy and does not consider the occupancy of any structure when fetch gating. Thus it does not perform fetch gating; it is the original ICount policy of Tullsen et al. [74]. IC_1110 uses ICount and fetch-gates a thread whose utilization of the IQ, LSQ, or ROB is above the threshold. IC_1011 corresponds to the Choi policy.

Table 1: Examples of different fetch PG policies.

Policy Mnemonic	Fetch Priority	Fetch-gate if occupancy of any of these structures exceeds threshold: IQ	LSQ	ROB	IRF
IC_0000	ICount	no	no	no	no
BrC_1000	Branch Count	yes	no	no	no
IC_1110	ICount	yes	yes	yes	no
IC_1111	ICount	yes	yes	yes	yes
LSQC_1111	LSQ Count	yes	yes	yes	yes
RR_1111	Round Robin	yes	yes	yes	yes

In this way, we have a total of 64 ($4 * 2^4$) different fetch PG policies. We tested all of them for a set of 43 2-threaded workloads from SPEC17 [67] (refer to Section 6 for our methodology). For each of the workloads, Figure 5 shows the IPC change relative to the Choi policy of the best- and the worst-performing fetch PG policies among the ones we consider. The policy that performs the best for each mix is added as a label. For the best policy, the thread that committed the most instructions in the mix appears in bold.

From the figure, we see that different fetch PG policies work best in different application mixes. In addition, there is significant IPC variation between the best- and the worst-performing policies in an application mix. Selecting a bad policy can decrease performance by more than 40% compared to IC_1011.

Application mixes consisting of lbm attain large speed-ups. In this case, policies that use LSQC as the fetch priority policy or consider LSQ occupancy when fetch gating (i.e., x1xx) offer substantial speedups (13–30%) over IC_1011. Note that these policies still allow the thread that benefits the most from memory-level parallelism to get more LSQ entries, if this distribution of entries maximizes the average performance. As discussed in [71], in a fully dynamically shared pipeline (with no LSQ-aware fetch gating mechanism) lbm frequently tends to aggressively consume all of the SQ entries, leaving the other thread with close to none SQ entries. In such cases, an LSQ-aware gating or priority mechanism prevents the exhaustion of the SQ entries by a single thread. Such policies result in a higher average IPC than the LSQ-unaware IC_1011.

Overall, the temporal stability of the threshold of the Hill Climbing algorithm in combination with the heterogeneity in the optimal fetch PG policy across different applications is fertile ground for MAB-based decision-making mechanisms.

4 MULTI-ARMED BANDIT ALGORITHMS FOR MICROARCHITECTURE

In this section, we focus on different Multi-Armed Bandit (MAB) algorithms and discuss their implications for decision-making problems in microarchitecture.

Table 2: MAB algorithm variables.

Variable	Definition
arm	Action available to the MAB agent
$bandit\ step$	Time duration that the agent is idle waiting to observe the reward from its previous arm selection
r_i	Average reward that previous selections of arm i have yielded
n_i	Number of times that arm i has been selected in the past
n_{total}	Total number of times that all arms have been selected in the past
r_{step}	Reward received at the end of a bandit step

Table 3: Multi-armed bandit algorithm variants.

	ϵ-Greedy	*Upper Confidence Bound*	*Discounted Upper Confidence Bound*
nextArm	$arm \leftarrow \begin{cases} \arg\max\{r_i\} & with\ prob.\ 1-\epsilon \\ random\ arm & with\ prob.\ \epsilon \end{cases}$	$arm \leftarrow \arg\max\{r_i + c\sqrt{\frac{\ln(n_{total})}{n_i}}\}$	$arm \leftarrow \arg\max\{r_i + c\sqrt{\frac{\ln(n_{total})}{n_i}}\}$
updSels	$n_{arm} \leftarrow n_{arm} + 1$ $n_{total} \leftarrow n_{total} + 1$	$n_{arm} \leftarrow n_{arm} + 1$ $n_{total} \leftarrow n_{total} + 1$	$n_i \leftarrow \gamma \times n_i, \forall i \in [1, M]$ $n_{arm} \leftarrow n_{arm} + 1$ $n_{total} \leftarrow \gamma \times n_{total} + 1$
updRew	$r_{arm} \leftarrow \frac{(r_{arm} * (n_{arm} - 1) + r_{step})}{n_{arm}}$	$r_{arm} \leftarrow \frac{(r_{arm} * (n_{arm} - 1) + r_{step})}{n_{arm}}$	$r_{arm} \leftarrow \frac{(r_{arm} * (n_{arm} - 1) + r_{step})}{n_{arm}}$

4.1 General Template for MAB Algorithms

The variables used in a MAB algorithm are shown in Table 2. An *arm* refers to a specific action available to the MAB agent, while the *bandit step* is defined as the time duration for which the agent is idle waiting to observe the outcome of its previous arm selection. For every arm i, two variables are needed: 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. Finally, n_{total} refers to the total number of selections of all arms and r_{step} is the reward received at the end of a bandit step.

Algorithm 1 General template for MAB algorithms

```
1:  Inputs: M arms
2:  Variables: r_i : average reward of arm i,
    n_i : number of times that arm i has been selected

    Initial Round Robin Phase
3:  n_total ← 0
4:  for t = 1 to M do
5:      arm ← t
6:      n_arm ← 1
7:      n_total ← n_total + 1
8:      // receive reward at the end of the bandit step
9:      r_arm ← r_step
10: end for

    Main Loop
11: for t = M + 1 to ∞ do
12:     arm ← nextArm()
13:     updSels(arm)
14:     // receive reward at the end of the bandit step
15:     r_arm ← updRew(r_step)
16: end for
```

Algorithm 1 provides a general template for MAB algorithms. For compactness, we use r_{arm} and n_{arm} instead of $r_{i=arm}$ and $n_{i=arm}$, respectively, in our notation. The algorithm takes the number of arms available M. It begins with an initial round robin phase, during which all arms are tried once. r_{arm} is set to the r_{step} received during that arm's step, and n_{arm} is set to 1. Then, the main loop of the algorithm begins, which lasts for as long as the agent keeps interacting with the environment. The main loop consists of three basic functions, 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_{arm} for the currently selected arm and potentially other arms; and *updRew(r_{step})*, which updates the reward r_{arm} for the currently selected arm after the bandit step is over and the r_{step} has been collected.

4.2 Three MAB Algorithms

The implementations of these three functions define how a MAB algorithm handles the exploration-exploitation tradeoff. We now discuss the implementations of these functions for three MAB algorithms: *ϵ-Greedy*, *Upper Confidence Bound (UCB)*, and *Discounted Upper Confidence Bound (DUCB)*. Their concise implementation is shown in Table 3.

(a) ϵ-Greedy: ϵ-Greedy [4] is the simplest of the algorithms discussed. As shown in Table 3, the *nextArm* function selects with probability $1 - \epsilon$ the arm that has the highest average reward so far ($\arg\max\{r_i\}$), and with probability ϵ a random arm. ϵ is an algorithm hyperparameter that is directly related to the degree of exploration: large values of ϵ mean that the agent is eager to explore more and exploit less. The *updSels* and *updRew* functions are also simple. In *updSels*, n_{arm} and n_{total} are incremented. In *updRew*, the r_{step} is added to the running average. ϵ-Greedy is used in prior MDP-RL works [11, 50, 64] as an exploration technique (not as a standalone algorithm).

(b) Upper Confidence Bound (UCB): Two shortcomings of ϵ-Greedy is that the exploration picks random arms and is non-decaying. The problem with randomized exploration is that very bad arms and nearly-optimal ones have similar probabilities to be explored. Consider an example from the microarchitecture domain with two different actions. In the past, one action has resulted in low IPCs and the other in high IPCs. ϵ-Greedy will keep on exploring the two with equal probability. In addition, non-decaying exploration means that the agent will keep on exploring at the same rate as time passes, even if it has acquired enough certainty about the quality of each arm.

The Upper Confidence Bound (UCB) algorithm [3] tries to solve both problems. While it uses the same *updSels* and *updRew* functions as ϵ-Greedy, it uses a more sophisticated exploration method in the *nextArm* function, as shown in Table 3. The next arm to be selected is the one with the highest *arm potential*. The potential is

the sum of the arm's average reward so far r_i plus a term added as a "bonus" given by $c\sqrt{\frac{\ln(n_{total})}{n_i}}$. This term is the exploration factor. In this expression, c is an algorithm hyperparameter, which we call *exploration constant*. It controls the degree of exploration. If an arm has not been selected much in the past, n_i is small compared to $\ln(n_{total})$, and that arm's exploration factor is large. That arm will be favored for selection. The exception is if the arm's r_i is unacceptably low, in which case preference will be given to other arms. As time passes and different arms are tried, exploration decays since $\frac{\ln(n)}{n}$ tends to 0 as n increases.

(c) Discounted Upper Confidence Bound (DUCB): The Discounted Upper Confidence Bound (DUCB) [24] is an extension to the UCB algorithm that is suitable for non-stationary (i.e., highly varying) environments. These environments are common in microarchitectural problems since the behavior of benchmarks changes with time. In these cases, we want to forget the past behavior that we observed.

As shown in Table 3, DUCB shares the *nextArm* and *updRew* functions with UCB, but employs a different *updSels* function. In *updSels*, all selections n_i are discounted by a γ constant, which is a hyperparameter that is less than 1. γ acts as a forgetting factor. For the currently selected arm, n_{arm} is first discounted by γ and then incremented by 1. As time progresses, since the n_i for rarely selected arms is discounted, the exploration factor for such arms grows, and the arms are possibly retried. This enables the capture of varying behaviors of an arm. Since DUCB is more dynamic than UCB, it is less likely to get trapped in suboptimal arms than UCB.

4.3 Modifications for Microarchitecture Environments

During our evaluation, we found two algorithm modifications that result in higher performance in microarchitecture environments. The first one is related to the fact that, in all of our use cases, we use IPC as the MAB reward, which can vary drastically across different benchmarks. At the same time, we use a common exploration constant c for all benchmarks. Recall that the potential of an arm in UCB and DUCB is given by $r_i + c\sqrt{\frac{\ln(n_{total})}{n_i}}$. This leads to the following unwanted effect: for low-IPC benchmarks, r_i is small and the effect of the exploration factor $c\sqrt{\frac{\ln(n_{total})}{n_i}}$ is relatively more prominent than in high-IPC benchmarks. As a result, MAB algorithms end up exploring a lot more in low-IPC benchmarks.

To mitigate this unwanted effect, we modify the MAB algorithm as follows. First, after the initial round robin phase of Algorithm 1 is completed, we calculate the average initial reward across all arms (r_{avg}). Then, we use r_{avg} to normalize: (1) the r_{arm} calculated during the initial round robin phase for each arm and (2) every new r_{step} collected during the main loop of the algorithm. This ensures that there is much less variation in exploration across benchmarks.

The second modification is related to our observation that when many cores are running MAB algorithms concurrently, inter-core interference phenomena can affect the bandit exploration. For example, assume that a core selects a very aggressive arm that causes DRAM bandwidth starvation for other cores. Then, those cores may mistakenly attribute the drops they are observing in their IPCs to the arm that they are currently exploring. This can lead to cores getting trapped in suboptimal actions.

To address this problem, a core p independently restarts the initial round robin phase of the algorithm with a small probability *rr_restart_prob* during the main loop execution, but without resetting the already-collected r_i and n_i values. With this design, p has a chance to re-evaluate all the arms again in a more stable environment, as it can be assumed that all the other cores have reached a steady state and little exploration is in progress. While this is not a perfect solution, it addresses the problem to some extent.

5 MICRO-ARMED BANDIT DESIGN

We design a lightweight microarchitectural agent that implements MAB algorithms in hardware. We call the agent *Micro-Armed Bandit* or *Bandit* for short. In this section, we describe its functionality and microarchitecture. We focus on the DUCB algorithm since our evaluation (Section 7) reveals that it is the highest-performing one. We discuss how Bandit can be used for prefetching and SMT fetch PG policy selection. While our work focuses on these two use cases, we anticipate that Bandit will find applications in other microarchitectural decision-making problems with sufficient temporal homogeneity in their action space.

5.1 Microarchitecture and Functionality

Bandit has a simple microarchitecture. 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. The *nTable* contains, for each arm i, the number of times that i has been selected so far (n_i); the *rTable* contains, for each arm i, the current value of its reward (r_i). The arithmetic unit executes the arithmetic operations in the *nextArm*, *updSels*, and *updRew* functions of Table 3 in hardware. The control logic triggers the selection of the next arm to apply, communicates the selected arm to the controlled microarchitecture units (e.g., data prefetcher or instruction fetch unit), and waits until the bandit step is over to read the values of the appropriate hardware performance counters into the Bandit microarchitecture. The latter will update its state so that it is ready for the next arm selection. This process repeats continuously.

Figure 6 shows the steps in more detail. Figure 6(a) shows the implementation of the *nextArm* function. The hardware sequentially reads the nTable and rTable for all the arms, calculates the corresponding potentials, and selects the arm with the highest potential as the new arm. Then, in Figure 6(b), the Bandit control logic communicates the arm selection to the controlled entity—in the figure, the L2 data prefetcher and the instruction fetch unit of an SMT core. In the background while the arm selection is communicated, the Bandit updates the nTable (Figure 6(c)) according to the logic of function *updSels* shown in Table 3.

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 6(d)). The figure assumes that the reward is the core's average IPC. Hence, the arithmetic unit receives the total number of committed instructions by the CPU core, subtracts from it the number of committed instructions at the end of the

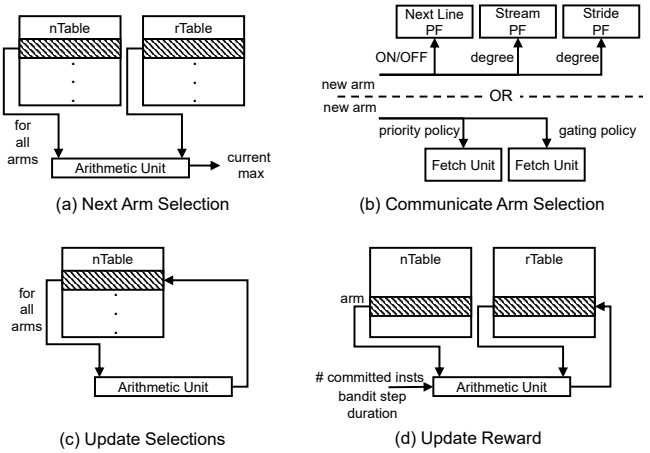

Figure 6: Micro-Armed Bandit microarchitecture.

previous bandit step, and divides the result by the step duration in cycles (calculated similarly through hardware counters). The result is the step's IPC or r_{step}. We will see in Section 5.4 that some of the operations in Figure 6(a) and Figure 6(d) can be performed before the bandit step is over.

5.2 Prefetching Use Case

We use Bandit to control the degree and type of a set of lightweight and widely adopted L2 prefetchers. This set includes a next-line (NL) prefetcher, a stream prefetcher, and a PC-based stride prefetcher. A bandit arm encodes whether the NL prefetcher is on or off, the degree of the stream prefetcher, and the degree of the stride prefetcher (Figure 6(b)). A degree of zero means that the corresponding prefetcher is off. We assume that, like in the POWER7 [40], the prefetcher degrees can be controlled through programmable registers. Hence, Bandit communicates its selected arm to the prefetchers by writing to the programmable registers. We define the duration of a bandit step as a certain number of L2 demand accesses.

5.3 SMT Instruction Fetch Use Case

We use Bandit to control the fetch PG policy of an SMT processor as introduced in Section 3. The processor additionally runs the Hill Climbing algorithm [17] to determine the best occupancy threshold to trigger thread fetch gating. Bandit runs on top of the Hill Climbing algorithm and does not control the occupancy threshold. A bandit arm encodes which fetch priority policy will be used and which pipeline structures will be considered for fetch gating (Figure 6(b)). The bandit step duration is defined as a certain number of Hill Climbing epochs, where a different threshold is tested in each epoch [17]. After this specific number of epochs, Bandit is invoked to select the next arm. The Hill Climbing algorithm requires the execution of some epochs before it can pick a good enough threshold for a given arm. For this reason, we use a larger bandit step during the initial round robin phase of the MAB algorithm (Algorithm 1). By holding the selected arm stable for a longer period of time, the Hill Climbing algorithm is given more time to converge, and the r_{step} received in this phase of the algorithm is more representative of the true capabilities of the arm. We refer to this initial bandit step as the *bandit step-RR*. After the initial round robin phase is over, we decrease the bandit step to a smaller number of epochs during the main loop of the algorithm. Section 6.3 describes how we select appropriate values for the bandit steps. Finally, every time the arm changes, the Hill Climbing threshold of the old arm is saved, and the one for the new arm is restored.

5.4 Storage Overhead and Latency

Bandit has a tiny 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 8B. For the maximum number of arms in our evaluation, which is 11 (Section 6), the total storage overhead is less than 100B. This low number is a direct result of not decomposing the agent's environment into individual states but instead treating it as a single MAB state as explained in Section 2.2 and Figure 1. In comparison, Pythia, which is an MDP-RL agent, requires 24KB just to store the state-action values.

The main operation that contributes to Bandit's latency is the selection of the next arm (Figure 6(a)). In a naive design, when the current step's reward (r_{step}) is received, the agent calculates the average reward of the current arm (r_{arm}). Then, it sequentially calculates the potential for all arms and picks the arm with the highest potential. Assume that n_{total} is available and that $ln(n_{total})$ is calculated once and reused for all arms, since n_{total} is common. In this case, computing the potential of an arm involves two reads (one from the nTable and one from the rTable), a division, a square root, a multiplication, and an addition (Table 3). By conservatively assuming a single non-pipelined arithmetic unit and a latency of 20 cycles for each of a square root and a division [22, 32], we estimate the total latency for picking the highest potential arm among 11 arms to be less than 500 cycles.

In practice, an advanced design takes much less time. Specifically, while the bandit step is in progress, the agent can compute in the background the potential of all arms except for the one being tested, and identify the best among them. It can also compute part of the r_{arm} from *updRew* in Table 3. When r_{step} is finally available, the critical path involves finishing the computation of the tested arm reward (r_{arm}), calculating the potential of the tested arm, comparing it to the potential of the best arm identified so far, and picking the best of the two. This operation takes about 50 cycles.

To be conservative, in our evaluation, we assume a critical path of 500 cycles as if the potentials of all the 11 arms are computed in the critical path. Still, this number is negligible compared to the total duration of a bandit step in our evaluation, which corresponds to 1,000 L2 demand accesses for prefetching and 128k cycles for SMT fetch priority and gating, as described in Section 6.

6 EVALUATION METHODOLOGY

6.1 Evaluation Environment

We evaluate Bandit and other prior microarchitecture proposals through simulation for both the prefetching and the SMT instruction fetch use cases. For the prefetching use case, we use the trace-driven Champsim [25] simulator, which is the standard choice for such studies [13, 36, 77, 78]. We build on the framework released by Pythia, which has integrated multiple prefetchers with Champsim.

For fairness, we use the exact same parameters as the ones used in the Pythia paper for the simulated cores, memory subsystem, and prior prefetchers. The core resembles an Intel Skylake [81] and the parameters are given in Table 4. The prefetcher is associated with the L2: it is trained on L1 cache misses and fills prefetched lines into L2 and LLC. We additionally simulate a configuration with alternative cache sizes (i.e., L2=1MB and LLC/core=1.5MB). We run both single-core and 4-core experiments.

Table 4: CPU parameters in prefetching experiments.

Fetch Width:	6	**LQ/SQ Size:**	72/56 entries
Decode Width:	6	**IRF/FRF Size:**	256/256 regs
Issue Width:	8	**Frequency:**	4 GHz
Commit Width:	4	**L1:**	32KB 8-way
IQ Size:	128 entries	**L2:**	256KB 8-way
ROB Size:	256 entries	**LLC/core:**	2MB 16-way

For the SMT use case, since Champsim does not offer a very detailed pipeline model and does not support SMT, we use Gem5 v20 [14]. In more detail, we build on top of the framework released by SecSMT [70, 71], which has added full support for SMT in the out-of-order core model of Gem5. Similar to SecSMT, we assume that all the pipeline structures (including ROB, IRF, and LSQ) are dynamically shared between threads. We use the exact same parameters as the SecSMT paper. The parameters are shown in Table 5. The architecture again resembles an Intel Skylake core.

Table 5: CPU parameters in SMT experiments.

Fetch Width:	16B	**LQ/SQ Size:**	72/56 entries
Decode Width:	5 uops	**IRF/FRF Size:**	180/164 regs
Issue Width:	8 uops	**Frequency:**	3.3 GHz
Commit Width:	8 uops	**L1:**	32KB 8-way
IQ Size:	97 entries	**L2:**	4MB 16-way
ROB Size:	224 entries	**LLC:**	No L3

In our simulation, we model a conservative latency of 500 cycles at the end of each Bandit step, until the next arm is selected. During those cycles, the prefetcher and the SMT scheduler do not stall but continue operating with the previously selected arm. This 500-cycle latency is negligible compared to the duration of a bandit step which, as shown in Table 6, is 1,000 L2 demand accesses for the prefetching experiments, and 128k cycles for the SMT instruction fetch experiments.

6.2 Applications

To evaluate Bandit for prefetching, we use a large collection of traces spanning different application suites, including SPEC06 [66], SPEC17 [67], PARSEC [49], Ligra [63], and CloudSuite [21]. For SPEC06 and SPEC17 we use the traces from the 3rd Data Prefetching Championship (DPC-3) [2]. For CloudSuite, we use traces provided by the 2nd Cache Replacement Championship (CRC-2) [1]. For Ligra and PARSEC, we use the traces released by Pythia.

For single-core experiments, we simulate 1 billion (B) instructions to capture as many program dynamics as possible.

When a trace is less than 1B instructions, we create two new traces. One is created by concatenating the original trace multiple times until 1B instructions are reached; another is created by extending the original trace with smaller traces from different phases of the same program until 1B instructions are reached, to simulate highly-dynamic scenarios. We evaluate both traces and report the average result. Note that, if we stop execution when short traces are consumed, some of the prefetchers may not have time to reach steady-state behavior, potentially leading to inaccurate performance results.

For 4-core experiments, to reduce the simulation time, we simulate until each core has completed 250M instructions. Similar to Pythia, we simulate two types of scenarios for 4-core experiments. First, we assign the same application to every core (forming homogeneous mixes). Second, we assign different applications to different cores (forming heterogeneous mixes).

Since Gem5 is execution-driven, for our SMT evaluation, we capture simpoints [26] from 22 SPEC17 applications using the reference input set. From those applications, we create 226 2-threaded combinations and simulate until each thread has completed 150M instructions. We also simulate an environment where each thread has to complete 250M instructions.

6.3 Bandit Tuning and Hyperparameters

To tune the different algorithm hyperparameters (e.g., c, γ, bandit step duration, and Hill Climbing δ in SMT), we use a small subset of our applications, which we will refer to as *the tune set*. For prefetching, we use 46 SPEC traces as the tune set, while for SMT, we use 43 2-threaded mixes stemming from 10 applications. We do not include non-SPEC traces in the prefetching tune set since we want to test the Bandit's adaptability to completely unseen application suites. This is conservative since Pythia uses more traces spanning different application suites when tuning its hyperparameters. By trying different parameter values in the tune set, we identify the best combinations and use them for our evaluation. Table 6 displays those parameter values. Note that, similar to the original Hill Climbing paper, we define δ (Section 3.2) in terms of IQ entries. In the prefetching use case, we tune the hyperparameters using the cache sizes of Table 4. We do not retune Bandit (or any other prefetcher) for the experiment with the alternative cache sizes.

Table 6: Parameters for the SMT thread fetch PG policies and data prefetching policies.

SMT Thread Fetch Priority and Gating		Data Prefetching	
Bandit Algorithm:	DUCB	**Bandit Algorithm:**	DUCB
γ:	0.975	γ:	0.999
c:	0.01	**c:**	0.04
# Arms:	6	**# Arms:**	11
Bandit Step-RR:	32 HillClimb Epochs	**Bandit Step:**	1000 L2 acc
Bandit Step:	2 HillClimb Epochs	**# Trackers in Streamer :**	64
Hill Climbing Epoch:	64k cycles	**# Trackers in Stride:**	64
Hill Climbing δ:	2	**rr_restart_prob (4 cores):**	0.001

We use 11 bandit arms for prefetching. Table 7 displays the semantics of these arms. For SMT, we prune the number of arms from 64 to the 6 that are displayed in Table 1. The combination of those 6 arms achieved performance very close to the best static performance of all 64 possible fetch PG policies in the tune set.

6.4 Comparison to Prior Proposals

For prefetching, we use the simple IP-Stride prefetcher [23] as our baseline prefetcher. Furthermore, we compare our design against the state-of-the-art Pythia, MLOP [60], and Bingo [7] L2 prefetchers. We additionally combine Bandit at L2 and a simple IP-stride at

Table 7: Arms used for prefetching.

Arm id	0	1	2	3	4	5	6	7	8	9	10
NL On/Off	Off	Off	On	Off	Off	Off	Off	Off	On	Off	Off
Stride Degree	0	0	0	0	2	4	0	8	0	0	15
Streamer Deg.	4	0	0	2	2	4	6	6	8	15	15

L1 and compare it against the state-of-the-art multi-level IPCP prefetcher [48]. For SMT, we compare our design against ICount (IC_0000) and the Choi policy (IC_1011) (Section 3). For the tune set, we additionally evaluate Bandit against the best static arm selection, by exhaustively keeping individual arms stable for the full experiment duration and selecting the best-performing arm on a per-application basis. We do this experiment to evaluate how close Bandit gets to an oracle that statically selects the best arm.

Although our evaluation reveals that DUCB is the best algorithm choice for Bandit, for the tune set, we additionally evaluate the other 2 MAB algorithms and also compare against simple heuristics that do not use MAB algorithms for the exploration. For simplicity, in the 4-core prefetching experiments and the SMT experiments, we evaluate performance based on the sum of the achieved IPCs by all threads. However, Bandit can easily optimize other metrics, such as the average weighted IPC [65] or harmonic mean of weighted IPC [44] by simply changing the Bandit reward and training Hill Climbing using the appropriate metric [17].

6.5 Area and Power

To estimate Bandit's area and power, we use CACTI [8] for the nTable and rTable, and the numbers from [56] for the arithmetic floating-point unit. We use the area and power scaling factors from [68] to scale down to 10nm. At 10nm, the total area and power of a single Bandit agent are 0.00044 mm^2 and 0.11 mW, respectively. To estimate the relative overheads in a real CPU design, we focus on a server-class 40-core Intel Icelake [31] with a TDP of 270W and a total die area of 628 mm^2 [57]. Assuming that each core of the processor is equipped with one Bandit agent, the relative area and power overheads of all Bandits are less than 0.003%.

7 EVALUATION

7.1 Performance Using the Tune Set

First, we present the results of our evaluation using the tune set. The experiments evaluate all bandit algorithms of Section 4, the best static arm selection algorithm, and two additional heuristic exploration methods. We call these two methods the *Single* and the *Periodic* exploration heuristics. The former stops exploring after the initial round robin phase of Algorithm 1, and keeps the arm that performed the best during that initial phase; the latter alternates between periodic phases of round robin exploration of all arms and exploitation of the best arm. We augment *Periodic* with a moving average buffer similar to the one in the POWER7 adaptive prefetcher [38]. This section aims to provide insights regarding the performance of bandit algorithms. We tune the parameters of all the algorithms to achieve good performance.

We start with the prefetching use case. Table 8 shows the min, max, and geometric mean IPC across all tune traces for Pythia, Single, Periodic, ϵ-Greedy, UCB, and DUCB *as a percentage* of the IPC of the best static arm. We observe that DUCB is the best algorithm in terms of geometric mean and min performance, while Pythia is the best for max performance. By holistically modeling the environment through a single state, bandit algorithms are able to explore effectively. Despite their simplicity, UCB and DUCB show better geometric mean performance than Pythia in the tune set.

Table 8: Min, max, and geometric mean IPC of several heuristic and bandit algorithms as a percentage of the IPC of the best static arm for the prefetching use case.

	Pythia	Single	Periodic	ϵ-Greedy	UCB	DUCB
min	88.7	72.8	80.3	89.8	88.6	**95.0**
max	**102.5**	100.0	99.8	99.9	100.0	101.6
gmean	98.4	96.5	94.1	97.3	98.8	**99.1**

When it comes to the highest max performance, we observe that Pythia outperforms the best static arm by 2.5%. This is because not all applications exhibit sufficient temporal homogeneity in their action space. However, as explained, temporal homogeneity is the common case. Interestingly, the maximum performance of DUCB is also above the best static arm. As we will see later, this is due to coarse-grained dynamic phases inside an application, during which different arms are optimal. DUCB, which is more appropriate than UCB for such scenarios, is able to detect and adapt to phase changes. Naturally, Single displays the lowest min performance since the one-time exploration can lead to very bad choices. The performance of Periodic and ϵ-Greedy is low due to their randomized and non-decaying exploration.

Similar results are presented in Table 9 for the SMT instruction fetch use case. In this table, we also add the Choi policy for reference. The geometric mean performance of DUCB normalized to the best static arm is slightly lower than the one in prefetching (Table 8). We also see that, once again, the max performance of DUCB exceeds the best static arm. We found out that the reason for this is not because the applications have significant phase changes. Instead, it is likely because of the operation of the Hill Climbing algorithm. In the best static arm experiments, the Hill Climbing algorithm can get trapped in local maxima. With DUCB, alternating between different fetch PG policies during exploration injects noise in the IPC that helps Hill Climbing escape those maxima.

Table 9: Min, max, and geometric mean IPC of several heuristic and bandit algorithms as a percentage of the IPC of the best static arm for the SMT thread fetch use case.

	Choi	Single	Periodic	ϵ-Greedy	UCB	DUCB
min	77.2	77.8	88.4	92.0	90.9	**92.2**
max	101.0	101.1	100.4	100.5	101.1	**101.4**
gmean	94.5	96.8	97.2	97.8	98.4	**98.6**

Figure 7 visualizes the exploration that different algorithms perform for different applications. Each plot shows the index of the arm explored as a function of time. The plots in the same row correspond to a specific algorithm (Best Static, Single, UCB, and

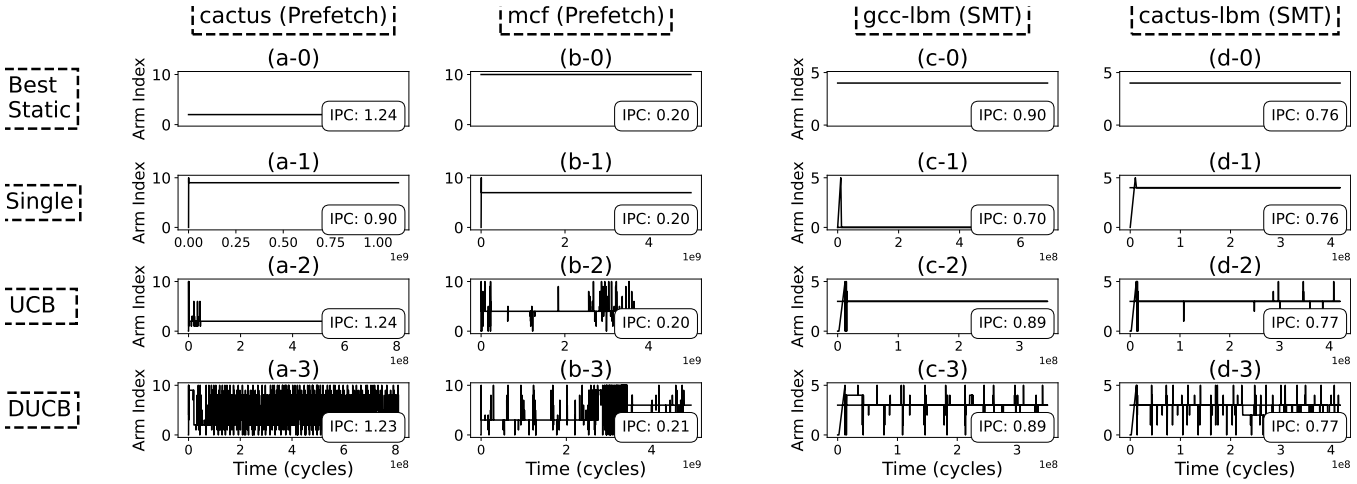

Figure 7: Exploration performed by different algorithms (rows of plots) for different applications (columns of plots).

DUCB); the plots in the same column correspond to a specific application (cactus for prefetching, mcf for prefetching, gcc-lbm for SMT, and cactus-lbm for SMT). Each plot also shows the IPC of the algorithm/application combination. We do not include ϵ-Greedy or Periodic in this figure because their plots are very noisy due to their non-decaying exploration. The leftmost two applications run under the prefetching use case, while the rightmost two run under the SMT thread fetching use case.

The plots show that Best Static does not perform arm exploration and that Single explores only during the initial round robin phase. Both UCB and DUCB explore, but DUCB explores more. Different algorithms choose different arm indices for the same application. For example, Single may choose a different arm than Best Static. This results in a dramatic IPC drop in cactus and gcc-lbm. On the other hand, although UCB and DUCB may frequently not select the best arm, they select an alternative arm that does equivalently well. For example, in gcc-lbm, both UCB and DUCB select arm three, which has a very similar IPC as the optimal arm four.

The mcf plot is an example of how DUCB is able to adapt to phase changes. At around three billion cycles, a phase change occurs. DUCB realizes this and starts increasing its exploration rate. Finally, it settles with a different arm than the one used in the first three billion cycles. The resulting 0.21 IPC is higher than the 0.20 IPC of the Best Static algorithm. On the other hand, UCB is unable to adapt to the phase change and keeps the same arm used in the first three billion cycles. Also note that, for cactus-lbm, the IPCs of UCB and DUCB are higher than the Best Static algorithm and no phase change occurs. This is due to the injection of exploration noise, which helps the Hill Climbing threshold escape local maxima.

Since DUCB outperforms the other bandit algorithms in the tune set, in the rest of this section, we focus on this bandit algorithm.

7.2 Performance of Prefetching

7.2.1 Single-core Evaluation. We compare the performance of Bandit to that of the baseline Stride prefetcher, the non-RL prefetchers MLOP and Bingo, and the MDP-RL prefetcher Pythia. Figure 8 compares the geometric mean IPC of these designs across all the traces for a single core. All IPC numbers are normalized to a scenario with no L2 prefetcher.

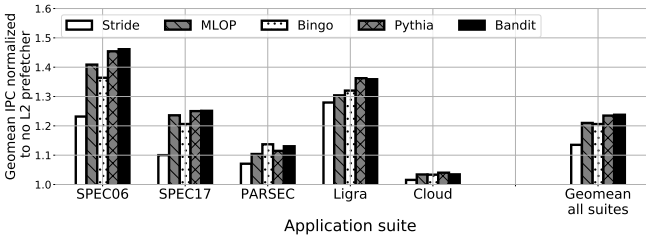

Figure 8: Single-core performance for different state-of-the-art L2 prefetchers.

We observe that Bandit achieves the best or close to the best performance in all the application suites. When the geometric mean of all the suites is considered, Bandit outperforms Stride by 9%, Bingo by 2.6%, MLOP by 2.3%, and Pythia by 0.2%. Remarkably, Bandit matches the performance of Pythia even though it introduces less than 100 bytes of storage overhead on a conventional processor and can be reused across different use cases. In contrast, the storage overheads of Pythia, MLOP, and Bingo are 25.5KB, 8KB, and 46KB, respectively. Even if we include in the Bandit's storage overhead calculation the storage of the NL, stream, and stride prefetchers (which are already fundamental parts of modern processors), Bandit's total storage overhead for prefetching is less than 2KB.

There is a combination of reasons why the simple bandit algorithm matches the performance of the more complex Pythia algorithm. First, prefetching is characterized by substantial temporal homogeneity (Section 3.1). Secondly, Bandit uses the sophisticated DUCB exploration, while Pythia uses an ϵ-Greedy action selection mechanism to explore actions for different states. Finally, Bandit uses the IPC as the reward, while Pythia assigns rewards based on prefetch timeliness and accuracy. Therefore, Bandit is designed to maximize performance directly, while Pythia is designed to maximize performance indirectly through prefetch timeliness and accuracy.

Figure 9 shows single-core prefetches and LLC misses normalized to LLC misses without prefetching (similar to [11]) for NoPrefetch,

Stride, Bingo, MLOP, Pythia, *BanditIdeal*, and Bandit. *BanditIdeal* does not account for the 500 cycles needed to update the bandit structures at the end of each bandit step. The figure shows the prefetches that are timely, the LLC misses that are uncovered (or that there is a late prefetch that is unable to cover the whole miss latency), and the prefetches that are wrong (i.e., the overpredictions).

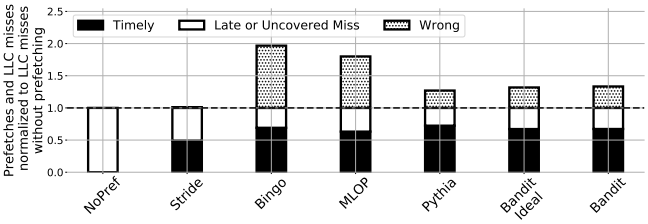

Figure 9: Single-core LLC misses and prefetches classified into timely, late, and wrong.

The figure suggests that Bandit is a conservative prefetcher. On average, it reduces the absolute number of wrong prefetches by 66% and 58% compared to Bingo and MLOP, respectively, while not issuing many more wrong prefetches than Pythia. The fraction of LLC misses covered with timely prefetches is 49% for Stride, 69% for Bingo, 63% for MLOP, 72% for Pythia, and 67% for Bandit. This number is slightly lower in Bandit than in Pythia due to cases of medium temporal homogeneity. However, recall that Bandit is trained to maximize the end result (i.e., IPC) and not the prefetch coverage as in Pythia. For this reason, Bandit is able to detect the configurations that cover the misses that are most critical for performance. As a result, although covering slightly fewer misses, Bandit is able to match Pythia's IPC (Figure 8). Finally, the number of timely prefetches is very similar in Bandit and in *BanditIdeal*. This result suggests that the Bandit latency does not significantly impact the timeliness of prefetches.

As explained by the Pythia authors [11], Pythia uses information provided to it about the memory bandwidth usage in the system. This memory bandwidth usage awareness allows Pythia to substantially outperform MLOP and Bingo in bandwidth-constrained system configurations. In Figure 10, we compare the performance of Bandit and Pythia for different available DRAM bandwidth values in megatransfers per second (MTPS). Our baseline configuration has an available DRAM bandwidth of 2400 MTPS, and the figure performs a sweep using 16 times smaller, 4 times smaller, and 4 times larger available DRAM bandwidth.

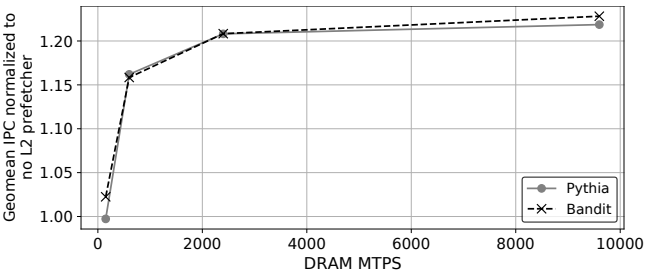

Figure 10: Performance of Pythia and Bandit for different available DRAM bandwidth values in megatransfers per second (MTPS).

From the figure, we see that Bandit performs as well as Pythia in all configurations. Interestingly, Bandit outperforms Pythia by 2.5% in the most bandwidth-constrained configuration (i.e., for 150 MTPS). This is because the Bandit agent judges actions based on the end result (i.e., final IPC). Aggressive prefetching actions in bandwidth-constrained system configurations do not yield good IPC results. Bandit learns this and outperforms Pythia without needing to use any bandwidth-related information in its reward function.

7.2.2 Comparison for Other Configurations. In this section, we consider some architectural variations. Note that we do not retune any of the prefetchers for the new architectures. Figure 11 repeats the experiment in Figure 8 in a system with a different cache hierarchy. We change the L2 size to 1MB and the LLC size to 1.5MB, which are similar to the sizes used in the Intel Skylake. The rest of the system parameters are kept the same. We observe that Bandit still attains better performance than the other prefetchers. When the geomean of all suites is considered, Bandit outperforms Stride by 9%, Bingo by 1.5%, MLOP by 4.9%, and Pythia by 0.2%.

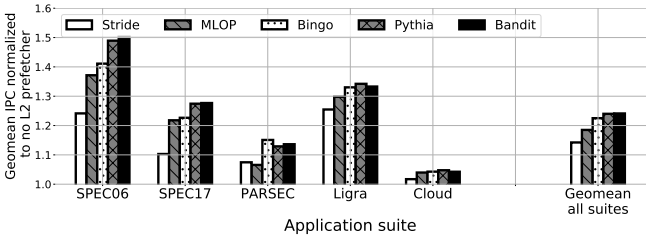

Figure 11: Single-core performance for a different cache hierarchy.

We now consider hierarchies with both L1 and L2 prefetchers. They include the multi-level IPCP prefetcher and different L2 prefetchers (Stride, Pythia, and Bandit) augmented with a simple stride prefetcher at the L1 cache (*Stride_Stride*, *Stride_Pythia*, and *Stride_Bandit*). Figure 12 shows the geometric mean IPC of each of these configurations relative to a scenario with no L1 or L2 prefetcher. The figure shows that the geomean IPC increase over no-prefetching is 16% for Stride_Stride, 24.5% for IPCP, 24.8% for Stride_Pythia, and 24.5% for Stride_Bandit. This reveals that Bandit at L2 in combination with a simple stride at L1 is an excellent option. Some extensions that could further increase the performance of Bandit could be: (1) use a second Bandit to control lightweight prefetchers at L1 or (2) use a single Bandit to control both the L1 and L2 prefetchers in conjunction. We plan to evaluate such extensions in future work.

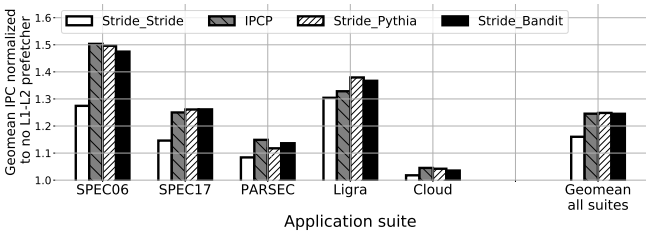

Figure 12: Single-core performance for different multi-level prefetcher combinations.

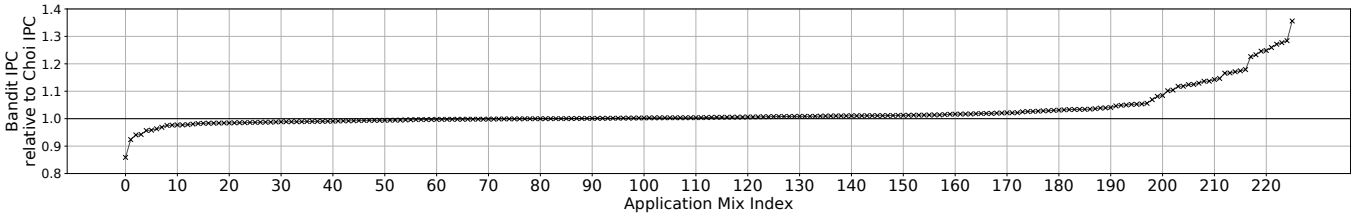

Figure 13: Performance of Bandit relative to Choi for 226 2-thread mixes from 22 SPEC17 applications.

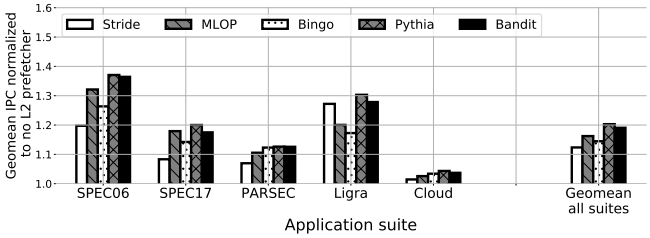

Figure 14: Four-core performance for different state-of-the-art L2 prefetchers.

7.2.3 Multicore Evaluation. In this section, we evaluate a four-core architecture. Figure 14 shows the four-core performance for different state-of-the-art L2 prefetchers. The four cores of the architecture all run the same single-threaded application. Looking at the geometric mean of all suites, Bandit outperforms Stride by 6%, MLOP by 2.4%, and Bingo by 4.0%. However, it performs slightly worse (by 1.0%) than Pythia. The reason is that, in the 4-core scenario, the reward is more noisy than in the single-core scenario. Since four bandits are operating and exploring in parallel, their decisions interfere. If a bandit chooses an aggressive arm that causes DRAM bandwidth starvation for the other cores, the bandits of these other cores may mistakenly attribute the drop in their IPCs to the arm they are currently exploring. The result could be to get trapped in suboptimal arms. It can be shown that, for workloads that combine different applications, the result is similar.

Overall, Bandit's performance for 4-core mixes is good, especially considering its tiny storage footprint. In future work, we plan to investigate alternative rewards that are not very sensitive to inter-core interference and advanced techniques to orchestrate the exploration of multiple bandits [18].

7.3 Performance of SMT Thread Fetching

We now consider the SMT fetch policy use case. We compare the performance when Bandit is used to control the SMT thread fetch PG policy and when the Choi policy is used. In the evaluation, we use the 226 2-thread mixes from 22 SPEC17 applications discussed in Section 6.2. Figure 13 shows the IPC achieved with Bandit relative to the IPC attained with Choi. In the x-axis, the application mixes are sorted from lower to higher ratio of Bandit IPC to Choi IPC.

We observe that Bandit offers clear benefits over Choi. It outperforms Choi by more than 4% in 36 application mixes, and by 36% in one mix. On the other hand, Choi outperforms Bandit by more than 4% in only 6 application mixes. These are cases in which the Bandit exploration is not optimal and settles for policies worse than Choi (i.e., IC_1011). Overall, including all the mixes, Bandit offers a 2.2% geometric mean speedup over Choi and, although not shown in the figure, a 7% geometric mean speedup over plain ICount (i.e., IC_0000). Finally, we repeated these experiments extending the simulation duration to 250M instructions from each thread and observed similar results.

To understand the performance difference between Bandit and Choi, we focus on the activity of the rename stage of the pipeline of the core. The rename stage can be stalled, idle, or running (i.e., doing useful work). Rename is stalled due to full conditions in various shared pipeline structures—mostly the ROB, IQ, LQ, SQ, and RF. Rename is idle when the fetch and decode stages have not pushed any instructions to it, possibly due to fetch gating.

Figure 15 shows, for Choi and Bandit, the average fraction of cycles that the rename stage is in different states. From left to right, the bars show cycles stalled due to ROB being full, IQ being full, LQ being full, SQ being full, and RF being full. The next three bars are the average fraction of cycles when rename is stalled due to any structure being full, when rename is idle, and when rename is running.

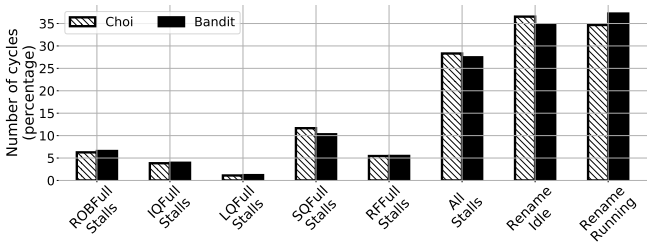

Figure 15: Percentage of cycles when rename is stalled, idle, or running for Bandit and Choi.

The figure shows that Bandit reduces the fraction of cycles when rename is stalled and when rename is idle. As a result, it increases the fraction of running cycles. In particular, Bandit decreases rename stalls because it includes arms that are aware of SQ occupancy. As a result, it decreases the stalls due to SQ full conditions. Importantly, Bandit decreases rename idle cycles by being able to detect the criticality of each structure in cases of asymmetric utilization (Section 3.3). In this way, it eliminates a substantial number of conservative fetch gating occurrences. As a result, on average, the fraction of cycles when rename is running is 2.6% higher with Bandit than with Choi.

8 RELATED WORK

1. Prefetching: Throughout the years, numerous prefetchers have been proposed, including adaptive [28, 30, 38] and composite [42, 76] prefetchers. IPCP [48] groups instruction pointers in classes based on their access patterns and uses a lightweight prefetcher per class. Jimenez et al. [38] propose heuristic software techniques to

control the aggressiveness of the IBM POWER7 stream and stride prefetchers. Software techniques are not as agile as hardware agents and cannot detect short-lived program mini-phases. Our *Periodic* heuristic is inspired by this work.

The Best Offset Prefetcher (BOP) [47] tries to identify a single best offset that is used for all cache lines, by scanning all the possible options and learning the optimal one per epoch. In addition, it always prefetches with a degree of 1. Although such an approach would work in cases of perfect temporal homogeneity, it fails to adapt in scenarios with high (but not perfect) homogeneity, in which a few different degrees and offsets are optimal (Figure 2). On the other hand, Bandit controls an adaptable prefetcher mix and thus can concurrently sustain a few different degrees and offsets in the same program phase. Hence, it can adapt in cases of high but imperfect temporal homogeneity. An additional difference is that, instead of scanning all the possible options as in BOP, Bandit uses the sophisticated DUCB exploration algorithm.

In the RL domain, the most notable examples are Pythia [11] and the Context Prefetcher [50], which were discussed in Section 2. Pythia was shown to outperform the Context and adaptive POWER7 prefetchers [11]. In our evaluation, we extensively compared Bandit's performance to Pythia's and other state-of-the-art prefetchers [7, 48, 60]. Recently, ML and RL-inspired prefetch managers have been proposed [20, 35] to coordinate multi-level prefetchers. We believe that Bandit can also be useful to manage multi-level prefetchers and we plan to test this use case in future work.

2. SMT resource distribution: Apart from the Hill Climbing algorithm [17], there are other methods for distributing shared resources in an SMT pipeline. For example, ARPA [79] assigns more resources to threads based on their usage efficiency. Using Bandit to augment such alternative methods, in a manner similar to how we used it for Hill Climbing is an interesting research direction. Some recent works employ neural networks for SMT processor resource distribution [16, 82]. However, the high complexity and overhead of such ML models are hard to justify for resource-constrained environments such as a processor pipeline.

3. Hardware RL agents for other use cases: Apart from the use cases that we focused on in this work, hardware RL agents have been proposed for a variety of other applications. For example, Ipek et al. [34] propose a SARSA-based memory controller, while Zheng and Louri [83] propose a NoC design with per-router Q-Learning agents. Sibyl [64] uses RL for data placement in hybrid storage systems. Cohmeleon [85] employs Q-Learning to find the best cache coherence mode for different accelerators in heterogeneous SoCs. While effective, all of these approaches suffer from the increased complexity and overhead introduced by decomposing the environment into multiple states, as explained in Section 2. SOSA [18] is a cross-layer system for hierarchical SoC management. It employs a software supervisor that coordinates low-level hardware agents. Ideas from SOSA could be useful to coordinate multiple Bandits operating in parallel.

9 FUTURE WORK

We outline two possible future research directions. One of them is to investigate the applicability of Bandit to other use cases such as the ones discussed in the last part of Section 8. We plan to investigate whether the simplified approach followed by Bandit is sufficient for some of these use cases.

The second direction is to study Bandit extensions with slightly higher storage needs than the design proposed here, which can increase performance further. There are many ways in which potential Bandit extensions could utilize additional storage budget. An obvious way is to increase the action space. For example, for the prefetching use case, Bandit could also control the target cache level and/or prefetch throttling mechanisms. An alternative extension is to use a single Bandit to control multiple ensembles. One example is to control both the L1 and L2 prefetcher configurations or to jointly select the prefetcher configuration and the cache replacement policy. Note that the action space size in this case is given by the product of the action space of each ensemble (e.g., for 10 L1 and 10 L2 prefetcher configurations, there are a total of 100 Bandit actions).

During our tuning experiments, we observed that different values of the DUCB hyperparameters (e.g., γ and c in Table 6) worked best for different applications. This observation motivates an alternative way to utilize any additional storage budget: multiple low-level Bandits with different hyperparameter values could be concurrently active, and a high-level Bandit would select the best one. Finally, although Bandit is unable to discriminate environment states (Figure 1), it could be augmented with a classifier module that classifies memory access patterns [6, 48] online. Then, a separate Bandit could be used to pick the appropriate actions for each pattern type.

10 CONCLUSION

Our analysis of some microarchitecture decision-making problems reveals that, in a given time window, only a small fraction of the action space is useful. We exploit this observation by modeling the decision-making as a Reinforcement Learning (RL) environment using a single state, drastically reducing complexity and storage requirements. We then design a hardware RL agent implementation based on Multi-Armed Bandit algorithms that is lightweight and reusable. We call our RL agent *Micro-Armed Bandit* or Bandit.

We showcase Bandit in two use cases: data prefetching and instruction fetch in SMT processors. For prefetching, our agent outperforms non-RL prefetchers Bingo and MLOP by 2.6% and 2.3% (geometric mean), respectively, and attains similar performance as the state-of-the-art RL prefetcher Pythia—with the dramatically lower storage requirement of only 100 bytes. For SMT instruction fetch, our agent outperforms the Hill Climbing method by 2.2% (geometric mean).

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for their feedback, which significantly improved the quality of the final manuscript. We additionally thank Shreya Seth for her help in making this work possible. This research was funded in part by an Intel Transformative Server Architecture (TSA) gift; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by NSF grants CCF 2107470, CNS 1956007, and CNS 1763658. Their support is gratefully acknowledged.

REFERENCES

[1] 2nd Cache Replacement Championship (CRC-2). 2017. https://crc2.ece.tamu.edu

[2] 3rd Data Prefetching Championship (DPC-3). 2019. https://dpc3.compas.cs.stonybrook.edu

[3] Peter Auer. 2003. Using Confidence Bounds for Exploitation-Exploration Trade-Offs. *J. Mach. Learn. Res.* 3 (2003), 397–422.

[4] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine learning* 47 (2002), 235–256.

[5] Peter Auer, Nicolo Cesa-Bianchi, Yoav Freund, and Robert E Schapire. 2002. The Nonstochastic Multiarmed Bandit Problem. *SIAM journal on computing* 32, 1 (2002), 48–77.

[6] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. 2020. Classifying Memory Access Patterns for Prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 513–526.

[7] 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.

[8] 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 (2017), 25 pages.

[9] Richard Bellman. 1957. A Markovian Decision Process. *Journal of mathematics and mechanics* (1957), 679–684.

[10] Rahul Bera, Konstantinos Kanellopoulos, Shankar Balachandran, David Novo, Ataberk Olgun, Mohammad Sadrosadat, and Onur Mutlu. 2022. Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–18.

[11] 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.

[12] Dimitri Bertsekas. 2019. *Reinforcement Learning and Optimal Control*. Athena Scientific.

[13] Eshan Bhatia, Gino Chacon, Seth Pugsley, Elvira Teran, Paul V. Gratz, and Daniel A. Jiménez. 2019. Perceptron-Based Prefetch Filtering. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.

[15] Ramazan Bitirgen, Engin Ipek, and Jose F. Martinez. 2008. Coordinated Management of Multiple Interacting Resources in Chip Multiprocessors: A Machine Learning Approach. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 318–329.

[16] Shane Carroll and Wei-Ming Lin. 2019. Applied On-Chip Machine Learning for Dynamic Resource Control in Multithreaded Processors. *Parallel Processing Letters* 29, 03 (2019), 1950013.

[17] Seungryul Choi and Donald Yeung. 2006. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, USA, 239–251.

[18] Bryan Donyanavard, Tiago Mück, Amir M. Rahmani, Nikil Dutt, Armin Sadighi, Florian Maurer, and Andreas Herkersdorf. 2019. SOSA: Self-Optimizing Learning with Self-Adaptive Control for Hierarchical System-on-Chip Management. 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, 685–698.

[19] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F. P. O'Boyle. 2010. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43)*. IEEE Computer Society, USA, 485–496.

[20] Furkan Eris, Marcia Louis, Kubra Eris, José Abellán, and Ajay Joshi. 2022. Puppeteer: A Random Forest Based Manager for Hardware Prefetchers Across the Memory Hierarchy. *ACM Trans. Archit. Code Optim.* 20, 1, Article 19 (2022).

[21] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (2012), 37–48.

[22] Agner Fog. 2011. Instruction Tables: Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering* 93 (2011), 110.

[23] John W. C. Fu, Janak H. Patel, and Bob L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture* (Portland, Oregon, USA) *(MICRO 25)*. IEEE Computer Society Press, Washington, DC, USA, 102–110.

[24] Aurélien Garivier and Eric Moulines. 2008. On Upper-Confidence Bound Policies for Non-Stationary Bandit Problems. *arXiv arXiv:0805.3415* (2008).

[25] 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 preprint arXiv:2210.14324* (2022).

[26] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[27] Milad Hashemi, Kevin Swersky, Jamie Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Learning Memory Access Patterns. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*. PMLR, 1919–1928.

[28] Wim Heirman, Kristof Du Bois, Yves Vandriessche, Stijn Eyerman, and Ibrahim Hur. 2018. Near-Side Prefetch Throttling: Adaptive Prefetching for High-Performance Many-Core Processors. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (Limassol, Cyprus) *(PACT '18)*. Association for Computing Machinery, New York, NY, USA, Article 28, 11 pages.

[29] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-Aware Computing. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 199–212.

[30] Ibrahim Hur and Calvin Lin. 2006. Memory Prefetching Using Adaptive Stream Detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 397–408.

[31] Intel. 2021. Intel Xeon Platinum 8380 Processor 60MB Cache 2.30 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/212287/intel-xeon-platinum-8380-processor-60m-cache-2-30-ghz.html

[32] Intel. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

[33] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. 2006. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 195–206.

[34] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, USA, 39–50.

[35] Majid Jalili and Mattan Erez. 2022. Managing Prefetchers with Deep Reinforcement Learning. *IEEE Computer Architecture Letters* 21, 2 (2022), 105–108.

[36] Shizhi Jiang, Qiusong Yang, and Yiwei Ci. 2022. Merging Similar Patterns for Hardware Prefetching. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1012–1026.

[37] 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.

[38] Victor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. 2012. Making Data Prefetch Smarter: Adaptive Prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. Association for Computing Machinery, New York, NY, USA, 137–146.

[39] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. 2006. A Predictive Performance Model for Superscalar Processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 161–170.

[40] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. 2010. POWER7: IBM's Next-Generation Server Processor. *IEEE Micro* 30, 2 (2010), 7–15.

[41] Sam Kaufman, Phitchaya Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. 2021. A Learned Performance Model for Tensor Processing Units. In *Proceedings of Machine Learning and Systems*, Vol. 3. 387–400.

[42] Sushant Kondguli and Michael Huang. 2018. Division of Labor: A More Effective Approach to Prefetching. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 83–95.

[43] Benjamin C. Lee and David M. Brooks. 2006. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 185–194.

[44] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing Thoughput and Fairness in SMT Processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS*. 164–171.

[45] Kai Ma, Xue Li, Ming Chen, and Xiaorui Wang. 2011. Scalable Power Control for Many-Core Architectures Running Multi-Threaded Applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) *(ISCA '11)*. Association for Computing Machinery, New York, NY, USA, 449–460.

[46] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 4505–4515.

[47] Pierre Michaud. 2016. Best-Offset Hardware Prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 469–480.

[48] Samuel Pakalapati and Biswabandan Panda. 2020. Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Spatial Hardware Prefetching. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 118–131.

[49] PARSEC. 2010. http://parsec.cs.princeton.edu

[50] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic Locality and Context-Based Prefetching Using Reinforcement Learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, Oregon) *(ISCA '15)*. Association for Computing Machinery, New York, NY, USA, 285–297.

[51] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. 2016. Using Multiple Input, Multiple Output Formal Control to Maximize Resource Efficiency in Architectures. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 658–670.

[52] Raghavendra Pradyumna Pothukuchi, Joseph L. Greathouse, Karthik Rao, Christopher Erb, Leonardo Piga, Petros G. Voulgaris, and Josep Torrellas. 2019. Tangram: Integrated Control of Heterogeneous Computers. 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, 384–398.

[53] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros Voulgaris, and Josep Torrellas. 2018. Yukta: Multilayer Resource Controllers to Maximize Efficiency. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 505–518.

[54] Herbert Robbins. 1952. Some Aspects of the Sequential Design of Experiments. *Bull. Amer. Math. Soc.* 58, 5 (1952), 527 – 535.

[55] G. Rummery and Mahesan Niranjan. 1994. On-Line Q-Learning Using Connectionist Systems. *Technical Report CUED/F-INFENG/TR 166* (11 1994).

[56] 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.

[57] David Schor. 2021. Intel launches 3rd Gen Ice Lake Xeon Scalable. https://fuse.wikichip.org/news/4734/intel-launches-3rd-gen-ice-lake-xeon-scalable/

[58] Steven L. Scott. 2010. A Modern Bayesian Look at the Multi-Armed Bandit. *Appl. Stoch. Model. Bus. Ind.* 26, 6 (2010), 639–658.

[59] 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.

[60] Mehran Shakerinava, Mohammad Bakhshalipour, Pejman Lotfi-Kamran, and Hamid Sarbazi-Azad. 2019. Multi-Lookahead Offset Prefetching. *The Third Data Prefetching Championship* (2019).

[61] 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.

[62] 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.

[63] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 135–146.

[64] Gagandeep Singh, Rakesh Nadig, Jisung Park, Rahul Bera, Nastaran Hajinazar, David Novo, Juan Gómez-Luna, Sander Stuijk, Henk Corporaal, and Onur Mutlu. 2022. Sibyl: Adaptive and Extensible Data Placement in Hybrid Storage Systems Using Online Reinforcement Learning. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 320–336.

[65] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. 2002. Symbiotic Job Scheduling with Priorities for a Simultaneous Multithreading Processor. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (Marina Del Rey, California) *(SIGMETRICS '02)*. Association for Computing Machinery, New York, NY, USA, 66–76.

[66] SPEC2006. 2006. https://www.spec.org/cpu2006/

[67] SPEC2017. 2017. https://www.spec.org/cpu2017/

[68] Aaron Stillmaker and Bevan Baas. 2017. Scaling Equations for the Accurate Prediction of CMOS Device Performance from 180nm to 7nm. *Integration* (June 2017), 74–81.

[69] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. MIT press.

[70] Mohammadkazem Taram. 2022. SecSMT Artifact. https://github.com/mktrm/SecSMT_Artifact

[71] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT Processors against Contention-Based Covert Channels. In *31st USENIX Security Symposium (USENIX Security 22)*. 3165–3182.

[72] Stephen J Tarsa, Chit-Kwan Lin, Gokce Keskin, Gautham Chinya, and Hong Wang. 2019. Improving Branch Prediction by Modeling Global History with Convolutional Neural Networks. *arXiv preprint arXiv:1906.09889* (2019).

[73] William R Thompson. 1933. On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples. *Biometrika* 25, 3-4 (1933), 285–294.

[74] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture* (Philadelphia, Pennsylvania, USA) *(ISCA '96)*. Association for Computing Machinery, New York, NY, USA, 191–202.

[75] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. 1995. Simultaneous Multithreading: Maximizing on-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture* (S. Margherita Ligure, Italy) *(ISCA '95)*. Association for Computing Machinery, New York, NY, USA, 392–403.

[76] Georgios Vavouliotis, Lluc Alvarez, Boris Grot, Daniel Jiménez, and Marc Casas. 2021. Morrigan: A Composite Instruction TLB Prefetcher. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1138–1153.

[77] Georgios Vavouliotis, Lluc Alvarez, Vasileios Karakostas, Konstantinos Nikas, Nectarios Koziris, Daniel A. Jiménez, and Marc Casas. 2021. Exploiting Page Table Locality for Agile TLB Prefetching. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 85–98.

[78] Georgios Vavouliotis, Gino Chacon, Lluc Alvarez, Paul V Gratz, Daniel A Jiménez, and Marc Casas. 2022. Page Size Aware Cache Prefetching. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 956–974.

[79] Huaping Wang, Israel Koren, and C. Mani Krishna. 2008. An Adaptive Resource Partitioning Algorithm for SMT Processors. 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, 230–239.

[80] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[81] WikiChip. 2016. Skylake (client). https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

[82] Huixin Zhan, Victor S. Sheng, and Wei-Ming Lin. 2022. Reinforcement Learning-Based Register Renaming Policy for Simultaneous Multithreading CPUs. *Expert Syst. Appl.* 186, C (2022), 13 pages.

[83] Hao Zheng and Ahmed Louri. 2019. An Energy-Efficient Network-on-Chip Design Using Reinforcement Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019* (Las Vegas, NV, USA) *(DAC '19)*. Association for Computing Machinery, New York, NY, USA, Article 47, 6 pages.

[84] Anastasios Zouzias, Kleovoulos Kalaitzidis, and Boris Grot. 2021. Branch Prediction as a Reinforcement Learning Problem: Why, How and Case Studies. *arXiv preprint arXiv:2106.13429* (2021).

[85] Joseph Zuckerman, Davide Giri, Jihye Kwon, Paolo Mantovani, and Luca P. Carloni. 2021. Cohmeleon: Learning-Based Orchestration of Accelerator Coherence in Heterogeneous SoCs. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 350–365.