A Reinforcement Learning Approach to Optimize the MLC Prefetcher Aggressiveness at Run-Time

MAJOR QUALIFYING PROJECT

Submitted to the faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Bachelor of Science in

ELECTRICAL & COMPUTER ENGINEERING

COMPUTER SCIENCE

by

Matthew Joseph Adiletta

December 18, 2020

Approved:

Professor Berk Sunar, Project Advisor

Professor Craig Shue, Project Advisor

Sponsored by Intel Corporation

This report represents work of WPI undergraduate students submitted to the faculty as evidence of a degree requirement. WPI routinely publishes these reports on its web site without editorial or peer review. For more information about the projects program at WPI, see http://www.wpi.edu/Academics/Projects

1 Abstract

The memory subsystem is a critical component of a CPU, composed of architectures such as caches and predictors. A cache is used to reduce the latency from when a request to memory is generated and when it is available to the core. Data prefetchers further reduce latencies by speculating about future accesses and increasing data coverage. However, aggressive prefetching may increase cache pollution, create a memory bandwidth performance bottleneck, and add latency to critical path demand queues. Cache pollution may occur because the prefetcher may bring in unnecessary data into the cache that will never be used. A memory bottleneck may occur because the prefetcher generates excessive requests to memory such that critical memory demand misses incur extra latency because there is not enough allocated memory bandwidth. Finally, the core queues may fill up because of the prefetching and demand queues are shared, thus excessive prefetching will add additional latencies for these queues.

Managing the aggressiveness of the prefetchers is necessary to mitigate these problems. State of the art hardware prefetcher solutions manage aggressiveness by analyzing telemetry data such as prefetcher accuracy and memory bandwidth consumption. This is an insufficient solution because telemetry data alone does not necessarily correlate to the overall system performance. Furthermore, the current solution optimizes the prefetchers individually, rather than allowing the prefetchers to work together to improve the overall system performance. Appropriate management of the prefetcher aggressiveness may lead to performance improvements. Preliminary investigations motivating our work analyze enabling and disabling the prefetchers. We demonstrate that 60% of regions of interest form the SPEC CPU2017 suite show performance improvement by disabling one of the prefetchers. We take this work a step further by managing the aggressiveness at a finer granularity than purely on or off.

We propose the Aggressiveness Degree Manager (ADM), employing Q-learning to find the optimal prefetcher aggressiveness policy for multiple prefetchers at run-time. The aggressiveness degree represents the number of cache lines a prefetcher may demand on a single request. A prefetcher with aggressiveness degree zero means the prefetcher is disabled. Highly aggressive prefetchers demand up to ten cache lines in a single request. The ADM agent manages the aggressiveness degree by varying the degree from zero to a maximum threshold. The current model is designed for a single-core, single-process implementation, optimizing the MLC prefetchers. We evaluated the ADM agent using 10 prefetch sensitive workloads from the SPEC CPU2017 suite. The ADM agent demonstrated a 4.2% higher speedup than the best static hardware configuration, with a 2.6% average variance from the optimal prefetcher aggressiveness degree.

Table of Contents

Table of Figures

List of Tables

2 Introduction

Caches are hardware structures used to reduce the time from when a request to main memory is generated to when the data is available to the core. The difference in time is the latency of the request. Caches help reduce the latency by bringing smaller amounts of memory closer to the core. A caching hierarchy is useful because it acts like a canal lock between a small river, and a large body of water. The small river in this analogy is the core, and the large body of water is the main memory. Each lock within the canal is a cache, and the closer the lock to the river, the faster it can be reached. Furthermore, all the water is shared between the locks, analogous to an inclusive caching system, where all the data in lower caches must exist in higher level caches.

A technology intended to further reduce latency is a cache prefetcher. Prefetchers bring blocks of data into the cache before they are referenced. Using the previous analogy, a prefetcher lowers the water in a lock prior to a boat needing to go between locks. Speculative analysis using spatial locality and memory access patterns help prefetchers decide which cache blocks might be valuable. Successful prefetching reduces or eliminates demand misses, diluting the memory bandwidth bottleneck caused by a memory bound workload. Unfortunately, prefetching unsuccessfully may have significant negative consequences on performance: prefetching may increase cache pollution by evicting useful cache blocks with useless cache blocks; high volume speculative requests for both necessary and unnecessary cache blocks may create a memory bandwidth performance bottleneck and significantly increase latency on demand requests; prefetching increases pressure on demand queues and pipeline stages, adding latency to resources on the critical path of a workload.

An optimal prefetcher is perfectly accuracy, timely, and complete. Accurate meaning that the prefetcher only requests cache blocks that will be needed by the demand stream. Timely meaning that the prefetcher is ahead of the demand stream such that all prefetched blocks are available in the cache before they are needed. Complete meaning that the prefetchers requests all future memory accesses needed in the demand stream. Different types of prefetchers are required to improve completeness by capturing various types of locality and memory access patterns.

The work described in this report addresses aspects of accuracy and timeliness by managing the aggressiveness of a prefetcher. This is a well-studied area because managing the aggressiveness has significant performance impact. High level solutions to this problem dynamically enable or disable prefetchers based on run-time telemetry data, such as prefetcher accuracy, memory bandwidth availability and demand queue pressure [2, 9, 10]. Finer granularity solutions throttle the aggressiveness at run-time by adding a spectrum of states in-between enabled and disabled to optimize performance more precisely [7, 13, 15, 16, 23, 26].

These solutions all demonstrate significant performance improvement over static prefetcher configurations for prefetch sensitive workloads. However, these solutions have several unaddressed problems.

1. Deep/Machine learning enabled solutions require pretraining on working set of workloads to appropriately configure prefetchers.
2. Deep/Machine learning enabled solutions require significant hardware overhead to implement.

3. Simple hill-climbing solutions converge slowly on optimal prefetcher configurations in dynamic environments and are prone to becoming “stuck” in a single configuration that is not necessarily globally optimal.
4. Many solutions presented in literature do not use a fair reward scheme to maximize overall system performance.

2.1 Contributions

In order to address these problems, we propose the Aggressiveness Degree Manager (ADM), a reinforcement learning agent employing Q-learning to find the optimal prefetcher aggressiveness policy for multiple prefetchers at run-time. An ADM agent adjusts prefetcher configurations to optimize a Quality of Service (QoS) metric representing the overall system performance. Synergism between prefetchers maximizes the overall system performance rather than an individual prefetcher’s performance.

This work contributes the following:

- We studied and analyzed existing techniques that are currently implemented in hardware and techniques that are detailed in literature.
- We explain the motivation for managing prefetchers by identifying performance variance when enabling or disabling prefetchers for SPEC CPU2017 workloads.
- We present ADM, a reinforcement learning agent employing Q-learning used to manage the prefetcher aggressiveness at run-time. Then we provide intuition about why an ADM agent outperforms current solutions.
- We detail optimization techniques to improve the rate of convergence of an ADM agent on the optimal prefetcher configuration. We demonstrate the impact of these optimizations for 507.cactuBSSN_r, 500.perlbench_r and 523.xalankbmk_r.
- We conduct two types of experiments; a single prefetcher implementation and a dual prefetcher implementation. We evaluate the ADM agent’s performance against static prefetcher configurations for specific regions of interest within prefetch sensitive workloads. We demonstrate the single prefetcher ADM agent converging to the optimal configuration, reaching 98.5% of the optimal static configuration performance and the dual prefetcher ADM agent reaching 97.4% of the optimal static configuration performance.

We consider a single core, single application system with multiple active prefetchers. In the future we hope to present work supporting a multi-core solution because CPU developers have already introduced 64+ core systems where prefetching failures have a larger negative performance impact. The results of our research demonstrate significant performance improvement with minimal hardware overhead. A high-level goal of this work is to demonstrate the importance of applications of artificial intelligence at the hardware level, to pave a path for future works using similar low-cost AI implementations to realize performance improvements.

2.2 Road Map

Section 3 reviews prior art. Section 4 discusses necessary background information. Section 5 introduces motivation for this work. Section 6 presents the ADM agent implementation and optimizations. Section 8 evaluates the performance of an ADM agent using the methodology from Section 7. Finally, Section 9 concludes the report.

3 Related Work

Managing the prefetcher aggressiveness has been studied previously [2, 8, 9, 10, 13, 15, 16, 23, 24, 25, 26]. Current areas of research lie with enabling or disabling prefetchers. Heibel et.al propose a fine-grained hardware prefetcher control using the contextual bandit framework for dynamically enabling or disabling hardware prefetchers at run time [9]. They train the agent by randomly enabling or disabling prefetchers at regular intervals, showing a 4.3% performance speedup on average over a set of memory bandwidth intensive workloads. Liao et.al presented similar also enabling or disabling prefetchers, focused on data center applications [2]. They developed a tuning framework to predict the optimal hardware configuration based on performance counters to achieve a performance improvement of 1.4% to 75.1%. Rahman et.al proposes a solution to optimally enabling prefetchers by pruning program counters through an algorithm to obtain an expressive feature set [10]. This feature set is used in machine learning models to optimally enable prefetchers, showing a 96% convergence on the optimal configuration.

Another area of prefetcher aggressiveness research focused on finding the optimal aggressiveness distance. Srinath et.al proposes Feedback Directed Prefetching (FDP) which estimates prefetcher accuracy, prefetcher timeliness, and prefetcher-caused cache pollution to adjust the aggressiveness of the data prefetchers dynamically. FDP improves average performance by 6.5% on 17 memory-intensive benchmarks from SPEC CPU2000 [7]. The issue with FDP is that it reacts to poor performance rather than proactively anticipating negative performance. Heirman proposes a solution dubbed "Near-Side Throttling (NST)" which detects late prefetches and tunes the prefetcher aggressiveness to balance late prefetches with a small but non-zero fraction of all prefetches [23]. NST touts a 0.2% performance improvement over the state of the art at a far cheaper implementation cost.

The most relevant prefetcher aggressiveness research are fine-grained controllers. Ebrahim et.al proposes HPAC, a solution for controlling multiple prefetcher in multi-core systems by accounting for both inter-core and intra-core prefetcher-caused interference [13]. HPAC manages the aggressiveness by using telemetry data from both local and global perspectives, improving system performance by 14% against the state-of-the-art prefetcher aggressiveness control technique for an eight-core system. The issue with this solution is that it uses low-correlation telemetry data to gauge performance. Performance might correlate to telemetry data; however, it is not a fair method for managing the prefetcher aggressiveness.

Panda took a different approach, focusing on fairness when managing the prefetcher aggressiveness. His first aggressiveness solution titled CAFFEINE is a single core implementation [16]. Panda quickly expanded his prior work for multi-core systems, proposing SPAC: A Synergistic Prefetcher Aggressiveness Controller for Multi-Core Systems [15]. SPAC has two phases of execution: exploration and implementation. During exploration, a meta-controller explores all possible combinations of throttling levels. Then during implementation, the SPAC agent uses a synergistic throttling combination using a QoS metric to provide the maximum Fair Speedup (FS). FS is calculated using harmonic mean of IPC speedups. A similar implementation by Hasenplaugh was used for controlling cache allocations in multithreaded environments [26]. In this work, each thread is offered a new cache allocation and tested for a “large” number of cycles. If the overall hit rate is higher during the testing stage than the previous stage, the agent concludes that the new state is better than the old state and accepts the change. Experiment and evaluate models greatly improve the performance of an agent. Panda compared SPACs performance against the new state-of-the-art HPAC, demonstrating a 12.3% improvement over HPAC for a 4-core system, a 14.9% improvement for an 8-core system, and an 18.3% improvement for a 12-core system.

4 Background

This research manages the prefetcher aggressiveness on MLC prefetchers using a Q-learning implementation. To provide the reader a better understanding of the problem we are solving, we first introduce the fundamental principles of caching and the two MLC prefetchers we are optimizing. Then we discuss the importance of a prefetcher's aggressiveness. Finally, we explain the fundamental ideas of Q-learning.

4.1 Caching Overview

Program data is stored in memory. When the program needs to use data, the core generates a request to retrieve it from memory. Memory request have long latencies – hundreds to thousands of clock cycles. Latency is incurred because the memory system is gigabytes in size causing long look-up times and the main memory is far away from the core and built with a slow transistor design. The general process of a memory request is the following: first, the core must generate a request for a memory address, then the memory system must locate the data and send it to the core, finally the core receives the data and continues execution. Long latencies slow down the execution of a program, negatively impacting performance.

The solution to reducing the long latencies is caching. Caches are hardware structures that store smaller amounts of memory closer to the core to improve performance. This allows for efficient reuse or retrieval of previously computed data. Intel Xeon processors have four main caches; L1-Instruction cache, L1-Data cache, L2 cache (MLC), and a shared L3 cache (LLC). The L1 cache is unique to each core, while the L2 and L3 caches are shared between cores. Each cache is composed of 64-byte cache lines. The L1 caches are the smallest caches, typically only 32KB in size created using high-performance transistors. L2 caches are the second smallest, typically 256KB, sometimes shared between two cores. Finally, the L3 cache is the largest cache, typically 32MB or larger, shared by many cores.

When a memory address request is generated, caching speeds up the process of retrieving the data if the memory address is stored in a cache, hence bringing the memory closer to the core located in smaller data structures with faster look-up times. If a memory request is generated and the data is stored in the L1 cache, there is almost no latency from when the memory request is generated, to when the information is available for the core. L2 and L3 caches are farther away from the core and have longer lookup times, thus causing longer latencies. A memory request that does not exist in the L3 cache has the longest latency. It can take hundreds to thousands of cycles to retrieve a memory address from DRAM before it is available to the core. Therefore, one goal of the memory subsystem is to minimize the number of DRAM memory accesses.

A structure within a cache is a cache block. A cache block is composed of a certain number of cache lines; the number of cache lines per block is the associativity. In Intel architectures, the L1 and L2 caches are typically 8-way caches, meaning that eight cache lines can fit into a single cache block. Having more ways in a cache is useful because it reduces the number of collisions resulting in a valuable cache line being evicted. A collision is when a new cache line must be added to the cache, but the cache does not have an available cache line. The problems with increasing the associativity is threefold: difficult hardware design, power constraints, and lookup latency. Higher associativity is difficult to design in hardware because of the additional routing paths, hashing algorithms, and eviction policies. This incurs higher power to manage these additional features. Finally, the lookup latency increases because more tag bits are required, thus more bits are compared to determine if a line exists in a cache.

A cache line is derived from a memory address. Today's machines use either 32 or 64-bit memory addressing. A memory address is broken into three main sections, the tag, the index, and the offset bits. The offset bits specify the specific word within a cache line. For example, in a 64-byte cache line with a 4-byte word, there are 16 unique data values, so we need 4 offset bits to be fully representative. The index bits decide which set the cache line hashes into. For example, in a 256KB cache with an 8-way associativity and 64-byte cache line, there are 512 cache sets, so it uses 9 index bits to represent each of the 512 sets. Finally, the tag bits are used to identify a cache line with a larger group a data. Tags are used to quickly decide if a cache line exists in a cache. If the previous example is a 32-bit machine, 4-bits are used for the offset, 9-bits are used for the index, and 19-bits are used for the tag.

Using a 32-bit addressing mode, a core can generate a request for a 64-byte cache line from a virtual memory space of 2^{32} addresses. The index bits are used by the memory subsystem to calculate which cache set the cache line goes into. If there is a free cache line in the cache set, then the cache line is added directly into the cache. If all the ways within a cache set are used, then the memory subsystem must choose one of the ways to evict, making space for the new line. Cache eviction or replacement policies are how the memory subsystem decides which cache line to evict. Intel processors use a Pseudo Least Recently Used (PLRU) algorithm to decide which cache line to evict. The PLRU method combines recency with frequency to evicts the cache line that is least likely to be used in the future.

4.2 MLC-Prefetchers

The Mid-Level-Cache for an Intel Xeon core has two active prefetchers; the Data Prefetch Logic and the Adjacent Cache Line Prefetcher [1, 2, 3, 4]. The Data Prefetch Logic (DPL) detects streaming requests and fetches streams of instructions and data from memory to the L2 Cache. The DPL initialize a new stream for individual page accesses. Each stream of the DPL detects simple stride memory access patterns. Consider the following memory access pattern {a, c, e}. This pattern has a stride length of 2. One of the DPL streams will capture this plus 2-stride pattern and generate a prefetch for address {g}. Hyperparameters surround this prefetchers functionality, such as minimum and maximum detectible stride length, number of prefetches to generate on a detected stride pattern, confidence thresholds based on DPL accuracy metrics, and maximum number of streams.

The Adjacent Prefetch Cache Line (ACL) is a simple prefetcher which fetches the cache line adjacent to the current memory request [4]. This prefetcher is a highly active prefetcher that exploits spatial locality. A cache-miss on an Intel processor with ACL enabled brings in 128 bytes, leading to higher bus utilization. When ACL is disabled, the system only fetches 64 bytes. Enabling the ACL prefetcher is useful for workloads with high spatial locality [4]. That said, disabling the ACL prefetcher reduces memory bandwidth traffic by flooring unused prefetches. Both the DPL and ACL prefetchers follow similar rules about generating prefetches. A few prefetcher design choices for when to generate a prefetch are; on a cache hit or miss, on a prefetch hit, on a store address, on a software or hardware prefetch, or on a page walk [5]. These design decisions are built into the core and are not configurable post silicon.

4.3 Prefetcher Aggressiveness

The purpose of prefetching is to hide I/O latencies. Instead of generating a memory request and waiting for the memory address to become available, a prefetcher will try to predict future memory addresses and pull them into the cache for immediate use, saving hundreds to thousands of cycles per prefetch hit. The issue with speculative future accesses and increasing data coverage is the potential to prefetch unnecessary data [6].

The following are three important benefits of prefetching. 1) Prefetching reduces or eliminates demand misses by speculating necessary information into the cache prior need. 2) Prefetching avoids long delays by demand requests when device response times are irregular. 3) Data-driven workloads exhibit high memory demand with little reuse making prefetching an optimal decision to limit number of single-use demand misses.

The following are four negative effects of prefetching. 1) Prefetching can cause cache pollution where a cache becomes polluted with unnecessary data, ejecting useful data. 2) Prefetching increases physical memory pressure where page replacement daemon is stressed, and new pages may be evicted before being used. 3) Prefetching adds inefficiency to I/O bandwidth, especially when large amounts of unnecessary data are requested, creating a memory bandwidth performance bottleneck. 4) Prefetching increases device congestion where demand requests are padded with asynchronous prefetch requests causing latency in demand queues [6].

Prefetchers balance positive and negative impacts by changing aggressiveness. Prefetching aggressiveness is defined by is defined prefetcher distance and prefetcher degree. Prefetcher distance represents the distance from the current address to a new prefetch request. Distance is a metric of timeliness in that it represents how far the prefetcher stays ahead of the demand access stream. For example, if a demand access to address $\{a\}$ is generated immediately following a prefetch request for address $\{a\}$, then the prefetcher has poor timeliness because the request did not significantly reduce memory latency. Conversely, if a prefetch is generated for address $\{a\}$ but $\{a\}$ is evicted before it is used, the prefetcher is too far ahead of the demand access stream, thus also exhibiting poor timeliness.

Prefetcher degree represents how many lines are prefetched in a single request. An aggressive prefetcher fetches many lines farther away from the current address whereas a conservative prefetcher fetches few lines close to the current address [7].

4.4 Q-Learning

Q-learning is a reinforcement learning algorithm that seeks to learn a policy that maximizes total rewards. This algorithm is a form a dynamic programing requiring minimal computational and memory demands [27]. A Q-learning agent is composed of states, actions, and rewards. An agent represents the world it exists in using states noted as S. At every state, an agent can take some set of actions A. The agent receives a reward for an action a, taken in state s. If the action is beneficial, then the agent receives a large reward r. If the action is not beneficial, the agent receives a small reward.

A simple situation where Q-learning can be applied is a car at a traffic light. A traffic light has three states - red for stop, yellow for yield, and green for go - encoded $S = \{red, yellow, green\}$. The agent can take two actions in this situation - drive or stop - encoded $A = \{drive, stop\}$. The total number of states in this case is three, and the number of actions is two. Therefore, the agent must learn six rewards.

In this environment, if an agent drives through the traffic light legally (either a green or yellow state) the agent is rewarded a +1 reward. If the agent acts illegally by driving through a red light, then the agent receives a -1 reward. All other cases, the agent receives no reward. We can encode the rewards learned by this agent after many updates in Table 1. All entries in this table is notated as $Q(S, A)$ and a single entry is notated $Q(s, a)$.

State	drive	stop
red	-1	0
yellow	1	0
green	1	0

Table 1: Example Q-learning Agent

A Q-learning agent must learn the reward values at each state by taking an action an receiving a reward. To do this, the agent first survey's its environment to decide its state. Then the agent chooses an action a, that maximizes the reward r, in state s, notated $\max(Q(s, A))$. This agent may decide with some probability ϵ to take a random action allowing it to explore. After the agent decides which action to take, the agent receives a reward.

The value $Q(s, a)$ is updated using the Bellman equation shown in Equation 1.

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma\ * \max(Q(s', A)) - Q(s,a)\big) \tag{1}$$

The Bellman equation is composed of two parts; the previously assumed reward $Q(s, a)$, and the update using the actual reward r [28]. The immediate reward r is the reward provided to the agent after taking action a. The learning rate α weights the current reward against the assumed reward. Notice that this equation uses the best reward from the new state s' in the reward for the previous action. This is useful because it weights future rewards in the agents' current reward. The discount factor γ scales the effect of the future reward on the previous state-action pair. In the example above, the discount factor was 0.

The fundamental idea of the update function in Q-learning is that the agent updates its reward using the difference between the received reward r and the previously stored reward $Q(s, a)$ plus some scaling of future rewards by taking the action a. In the example above, this means that the reward at $Q(green, drive)$ will approach 1 but never reach 1.

5 Motivation

This section presents an investigation of the interactions of the MLC prefetchers. As previously noted, prefetching can affect a systems performance positively or negatively. We conducted a preliminary investigation which realized the performance impact for enabling or disabling the MLC prefetchers. Current systems enable the DPL and ACL prefetchers by default. Ideally, a dual prefetcher configuration should perform better than a single prefetcher configuration. If the dual prefetcher configuration performs worse than a single prefetcher configuration, then the two prefetchers are negatively interacting. This experiment was intended to reveal negative prefetch interactions, motivating future work for prefetcher synergism.

We compared three system configurations; a system with only the ACL prefetcher enabled, a system with only the DPL prefetcher enabled, and a system with both the DPL and ACL prefetchers enabled. We evaluated each system configuration using traces of regions of interest within workloads from the SPEC 2017 suite. The performance metric used for comparisons was instructions per cycle (IPC). We used a simulation tool to model the three system configurations. The results from the simulation are shown in Figure 1.

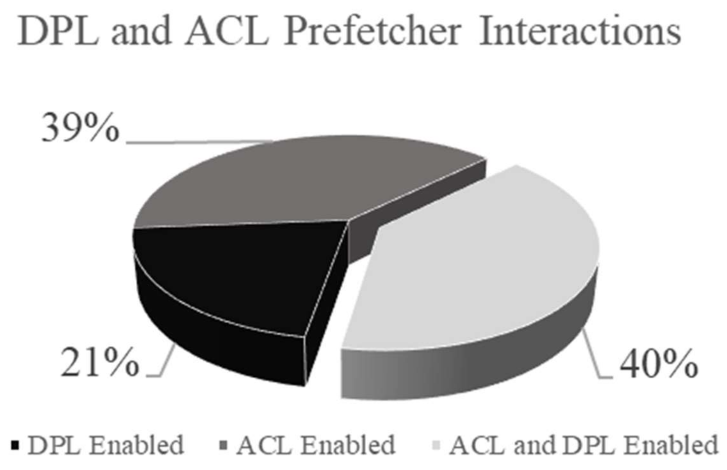


Figure 1: ACL and DPL prefetcher interactions show benefit for 40% of the traces with both prefetchers enabled, indicating that the interactions between the ACL and DPL two prefetchers must be managed.

We found that only 40% of traces benefited from using both the ACL and DPL prefetchers. The system is underperforming on 60% of the traces due to negative interactions between the ACL and DPL prefetchers. Causes for negative prefetcher interactions are discussed in Section 4.3. Clearly, a prefetcher management scheme is required to maximize the systems performance.

Our preliminary results are supported by prior work of Liao and Hung. Liao and Hung who showed that enabling and disabling prefetchers within an entire system has significant impact on performance [2]. They pretrained a machine learning model to optimally enable or disable the MLC prefetchers, revealing a 20% speedup execution time for cloud computing applications. Heibel improved upon Liao and Hung's work, designing an online training model using a contextual bandit reinforcement learning framework to learn optimal MLC prefetcher enablement configurations [9]. Rahman took these studies a step further, analyzing all the prefetchers within a system. Rahman researched the effect of hardware prefetching on multithreaded code and presented a machine-learning technique for predicting a system-wide optimal combination of prefetchers for a given application, noting that turning on all available prefetchers within a system rarely yields the best performance [10]. These three research teams showed that toggling prefetchers on or off based on system telemetry data has performance impact.

Motivated by our initial findings and supported by prior art, we focused our investigation on optimally configurating prefetchers at a finer granularity than the prior art, two-state models, which only enable or disable prefetchers. A prefetcher with aggressiveness degree zero has the equivalent functionality of disabling a prefetcher, and non-zero aggressiveness degrees have the equivalent functionality of enabling a prefetcher and weighting its performance by an undiscovered scaling factor. Our primary investigation was focused on developing a controller for tuning the prefetcher aggressiveness degree at run time.

6 ADM Implementation: Prefetcher Aggressiveness Controller

In this work, we propose the Aggressiveness Degree Manager (ADM). ADM utilizes the general Q-learning algorithm outlined in Section 4.4, proposing a state, action, and reward scheme to best manage the prefetcher aggressiveness degree. In this section, we introduce the reinforcement learning algorithm we used for controlling the aggressiveness of a prefetcher. First, we describe the Q-learning initiation and provide intuition about why this configuration performs well. Then we discuss optimization techniques that we developed to decrease the time for an agent to converge on an optimal prefetcher configuration policy.

6.1 ADM Initialization and Configuration

The proposed ADM is designed to maximize the processor performance subject to a quality of service (QoS) metric, such as instructions per cycle (IPC), power, or performance per watt. A Q-learning scheme is used for managing the prefetcher aggressiveness, with rewards based on the QoS metric. The Q-learning methodology described in Section 4.4 is implemented in a data structure called a Q-table which has states, actions and rewards. The Q-table in this implementation uses states representing the number of cache lines to prefetch (aggressiveness degree), actions that change the prefetcher aggressiveness, and rewards representing the expected performance of taking an action in each state. For example, an ADM agent initialized with three states, <0, 1, 2> means the agent may learn a policy to optimize its performance using a disabled mode (state 0), a single line prefetch mode (state 1), or a dual line prefetch mode (state 2).

The ADM agent has three actions; increase aggressiveness degree, decrease aggressiveness degree, and maintain current aggressiveness degree. These three actions are encoded by the vector <-1, 0, +1>. If an agent requests an action, then the returned value will be +1 to increase the aggressiveness degree, -1 to decrease the aggressiveness degree, and 0 to not change the aggressiveness degree. Thus, the size of the Q-table is N*3 where N represents the maximum number of possible degrees, $S = \{0, 1, \dots, N\}\}$and $A = \{-1, 0, 1\}$.

The reward for any state s and action a is represented as $Q(s, a)$. The reward value for each state-action pair is learned over time. The value of any state-action pair represents the expected reward according to the user defined QoS metric. For this research, we use IPC as our QoS metric [12, 13]. The agent updates the reward values according to the Bellman equation as shown in Equation 1 [11].

Degree	Decrease	Same	Increase
0	0.50	0.50	1.00
1	0.50	1.00	2.00
2	1.00	2.00	3.00
3	2.00	3.00	2.00
4	3.00	2.00	1.00
5	2.00	1.00	0.50
6	1.00	0.50	0.50

Table 2: Sample situation after reaching steady state.

A sample Q-table is shown in Table 2. Notice that the aggressiveness degree states range from 0 to 6 and the actions are increase, decrease or stay the same. The rewards are shown at the intersection of each state-action pair. For example, if the agent is in state 1, the expected reward from taking the increase action is 2.00. Looking at this example from a higher level, the optimal configuration is aggressiveness degree 3, showing the highest reward.

Notice that each of the surround states are guiding the agent towards this middle state. When the agent needs to choose an action, it takes the action presenting the maximum reward in a given state. For example, if the agent was forced into state 0, it's next action would be to increase aggressiveness, thus changing from state 0 to state 1. Then the agent would increase aggressiveness to state 2, then to state 3. Once the agent is in state 3, the agent chooses the "Same" action because it presents the highest reward. Occasionally, the agent will take a random action and leave the optimal configuration in case the environment changes and state rewards need to change, however, assuming steady state with no random actions, the agent will converge optimally.

6.2 Updating Q-table and Calculating Reward

A critical design decision for a Q-learning agent is when the agent updates its rewards. An ADM agent should be rewarded from the impact of the current state. The difficulty with rewarding an agent acting on a prefetcher is that a new prefetcher configuration does not necessarily show immediate performance improvement. This is because a cache line might be prefetched but go unused for a long period of time. Only after thousands of instructions may this cache-line become useful. For an agent to evaluate the effectiveness of its current state, the agent must allow the system's performance to "settle" by allowing many instructions to pass before calculating the reward and taking a new action. The "settling" period varies between workloads representing both timeliness and lifetime of a prefetched cache line.

Timeliness is a quantitative metric for comparing when a cache line is prefetched to when it is needed on the critical path of a program's execution. Work by W. Heirman addresses timeliness using an adaptive prefetching technique dubbed "Near-side Throttling" [23]. If a prefetcher generates a request for a cache-line, but then that cache-line is immediately needed on the critical path, the speculative prefetch distance is not aggressive enough. Similarly, if a prefetcher generates a request for a cache line but is inserted into the cache but evicted before it was needed, the speculative prefetch distance is too aggressive. This is also an indicator of the lifetime of a cache-line. If the lifetime of a prefetched cache-line is short, it may potentially have poor timeliness by becoming an unused prefetch eviction. If the lifetime of a prefetched cache-line is long, then aggressive prefetching may pollute the cache. In our work, we needed to address timeliness and lifetime issues to optimally tune the prefetcher aggressiveness degree.

To address QoS changes due to MLC prefetching issues such as cache pollution, a bottleneck at the memory pipeline, prefetch timelines, and cache line lifetime, we updated the Q-table after executing a specific number of instructions. In our ADM performance evaluation described in Section 8, we used an update frequency of 128,000 instructions. Note that ~33% of all instructions are either load or store instructions for the SPEC CPU2017 suite [14]. Thus, every Q-table update represents the performance impact of a prefetcher against ~40,000 load or store instructions. Since a miss at the L2 cache can cost hundreds to thousands of cycles, ~40,000 loads or stores will provide enough data for an agent to realize a performance impact from the current configuration. That said, further work is needed to quantify the importance of the update frequency hyperparameter.

At each update interval, the agent calculated the QoS value. In our studies we calculated instructions per cycle using the instruction and cycle counters from the associated MLC prefetcher core. These performance counters are available both in hardware and simulation, making the IPC a realistic QoS metric to maximize. Further analysis for QoS metric is presented in Section 7.3.

Algorithm 1 is an implementation for the initialization and training functions for an ADM agent. Note there are two global functions getInstructionCount() and getCycleCount() which return the instruction counter and cycle counter values respectively.

Algorithm 1: Q-Learning Initialization and Training Function

```
class QLearning:
    states = []
    actions = []
    qtable = {}

    function addState(state):
        self.states.append(state)

    function addAction(actions):
        self.actions = a

    function initialize():
        for s in self.states:
            self.qtable[s] = {}
            for a in self.actions:
                self.qtable[s][a] = 0

    function getAction(s):
        max_reward = 0
        for a in self.qtable[s]:
            if self.qtable[s][a] > max_reward then:
                best_action = a
                max_reward = self.qtable[s][a]
        self.old_state, self.old_action = s, a
        return best_action

    function updateReward(r):
        old_reward = self.qtable[self.old_state][self.old_action]
        self.qtable[self.old_state][self.old_action] +=
                        LR * (r - old_reward + G*future_reward)

update_frequency = UPDATE_FREQUENCY
prev_i_count, prev_c_count = 0, 0
action = 0

function InitializeAgent(MAX_DEGREE):
    q_table = new QLearning()
    for (i = 0; i < MAX_DEGREE; i++):
        q_table.addState(i)
    q_table.addActions(<-1, 0, +1>)
    q_table.initialize()

function TrainAgent(q_table, degree)
    delta_i = getInstructionCount() – prev_i_count
    if (delta_i > update_frequency) then:
        state = <degree, action>
        reward = delta_i / (getCycleCount() - prev_c_count)
        q_table.updateReward(reward)
        prev_i_count = getInstructionCount()
        prev_c_count = getCycleCount()
```

The ADM agent updates its state based on a delta instruction count, seen in the TrainAgent function of Algorithm 1. The intuition why we used instructions rather than cache hits or misses was because instructions is constant between degree states. Assume the update frequency is based on cache misses. If aggressiveness degree 0 causes more cache misses than aggressiveness degree 16, then an agent in aggressiveness degree 0 will be updating its reward more frequently than an agent in aggressiveness degree 16. This is an unfair scheme and biases the number of updates one state receives compared to another. This causes some states to converge on a realistic estimated reward faster than others, which is unfair. The instruction count does not vary with different degree states, how fast those instructions are executed will change, and the makeup of these instructions may differ, but the raw number of instructions executed can be held constant. Thus, the instruction count is a useful metric for when to update the ADM agent.

The ADM agent uses a reward derived from a high level QoS metric. An alternate reward scheme uses a combination of cache performance statistics to gauge the success of a system [7, 8, 10, 13]. The issue with optimizing a cache performance statistic is that the statistic might not have a strong correlation to overall system performance.

For example, the cache prefetcher accuracy may not be a useful metric. A cache prefetcher might have poor accuracy because it is prefetching more lines than needed. Although this pollutes the cache and uses extra memory bandwidth, if the workload has poor data-reuse and memory bandwidth is not a bottleneck, then the prefetcher may be improving performance despite poor accuracy. Similarly, a high cache hit-rate does not necessarily mean higher performance. A high hit-rate may indicate thrashing in the L1 cache. These two reward schemes show that designing a reward metric using cache performance statistics does not necessarily have high correlation to the overall system performance. In fact, prior work [7] found a correlation factor of ~.8 to IPC using a combination of cache statistics. Other researchers argue that using a reward derived from a high-level QoS metric shows better system performance [12, 15, 16] because this reward has 1.0 correlation to IPC. If an ADM agent's goal is to maximize IPC, it should use a metric derived from the IPC as a reward. This is explained further in Section 7.3.

6.3 Intuition about Success of this Algorithm

Q-Learning, a powerful machine learning technique, is implemented to address the prefetcher aggressiveness degree problem. Q-learning is an algorithm that expands upon simple hill climbing. Hill climbing works well in static environments because it can quickly ascend steep slopes to a decent maximum reward. A known drawback to hill climbing is that it may not find the optimal configuration. However, for this problem, most environments will have a convex optimization space, meaning a local minimum is also the global minima. Thus, hill climbing in general would be a great solution to this problem, except that the environment is not static.

Hill climbing is a poor solution in a dynamic environment because it relies on the assumption that all actions increase its performance. Hill climbing does not learn about the environment, it only searches for an optimal solution. Searching and learning are different because learning requires the agent to remember previous states, previous actions, and previous rewards. Searching does not have this requirement and general hill climbing does not have an "unlearning" mechanism. A major issue for hill climbing occurs in a dynamic environment if a configuration is optimal, but then becomes sub-optimal.

The hill climbing algorithm will become stuck in this configuration because it can not “unlearn” that the configuration is no longer the optimal configuration. Furthermore, if two states are similar in performance, creating a plateau or shoulder in the optimization space, the hill-climbing algorithm becomes stuck. These are two significant challenges with simple hill climbing that Q-learning solves.

Q-learning solves the challenge of being able to “unlearn” previously optimal because of the Bellman equation. The Bellman equation causes each state to converge to the expected reward in a changing environment by using the difference between the received reward and the expected reward. This is a characteristic of most reinforcement learning algorithms, making reinforcement learning a powerful solution.

A Q-learning agent solves the second challenge of escaping plateaus and shoulders through random actions. Using a random exploration technique, a Q-learning agent will leave assumed optimal plateau regions and learn about the surrounding area. This is another common technique among reinforcement learning algorithms, making reinforcement learning a useful solution.

The reason why Q-learning is specifically useful for this problem over other solutions is because of the minimal hardware requirements. Q-learning only requires an array of memory and a few calculations over a long period of time. The simplicity of the algorithm allows it to be implemented in hardware. The simplest method of implementing this algorithm is to store the Q-table at the OS level, and when the hardware needs to know how to change the configuration, the hardware can request information for the OS. Furthermore, the amount of memory locations needed is the number of states time the number of actions. For the prefetcher aggressiveness degree problem, this will require less than 50 states total, which is an insignificant memory footprint.

Another reason why Q-learning is a successful algorithm for this problem is because of its scalability. The problem we are solving is for a single core, single application; however, Q-learning is scalable to multi-core, multi-application environments with very few modifications. Future work in this space will reveal the benefits of Q-learning in a multi-application, multi-core environment.

To support multi-application, the OS must allocate the memory for the Q-table for each application. An obvious location to put this memory is at the process control block. Thus, when the application is running, the Q-table is loaded for the OS to use, and when it is not running, the OS can use a different applications Q-table to configure the hardware optimally.

To support multi-core, the Q-tables can work together to optimize the performance of the global system, rather than the local system. This may require additional Q-learning agents which manage lower level Q-learning agents. In this case, the lower level Q-learning agents operate on each prefetcher per core, and the higher level Q-learning agents look at the performance of all the cores and scale the rewards for each lower level prefetcher based on the large scale system performance.

Finally, this Q-learning can be applied to other hardware components, not just the prefetcher. Hardware components are configured statically by hardware designers. These designers test the different configurations across a set of workloads and take the globally optimal configuration. However, performance can be improved by dynamically adjusting these hardware parameters. Q-learning offers a low-cost method for dynamically configuring any hardware parameter.

6.4 Optimization Techniques for ADM Agent

In Sections 6.1 and 6.2 we introduced the fundamental Q-learning implementation for a prefetcher aggressiveness degree manager. Three drawbacks to a basic ADM agent are poor cooperation with other ADM agents, minimal exploration in new environments, and slow time to converge on an optimal configuration. In this section we discuss optimization techniques used to combat these issues.

6.4.1 Alternating Rewards

We examine the impact of our ADM agent using three experiments described in Section 8.1. The first experiment is applying an ADM agent to the Adjacent Cache Line (ACL) prefetcher, the second experiment applies an ADM agent to the Data Prefetch Logic (DPL) prefetcher, and the third experiment applies an ADM agent to both the ACL and DPL prefetchers. The issue with the current Q-learning scheme is that it does not provide an interface for agents to interact. Without the ability for agents to interact, it is difficult for the agents to work together to find the optimal prefetcher configuration.

A central challenge in reinforcement learning with multiple agents is the "temporal credit" assignment problem: to which actions should each agent attribute positive or negative reward? In our dual prefetcher optimization environment, two agents are trying to optimize for the overall system performance. If both agents act simultaneously, it becomes difficult for an agent to quantify the impact of its action.

We propose an optimization technique we call Alternating Rewards (AR). Consider a dual agent environment with agents a and b. Agent a will take an action, then receive a reward for its action. Then agent b will take an action and receive a reward for its action. This alternation continues for the duration of the agent's execution. The result of Alternating Rewards is when an agent needs to update a state-action value, it knows that the presented reward accurately represents the impact of its most recent action. Alternating Rewards can be taken a step further using a windowing technique, where dual agent's action-reward sequences overlap by some fraction of instructions. The fraction of overlap becomes the percent impact of the combined actions of both agents.

To help improve synergism between agents, we build off Alternating Rewards using a weighted moving average of the past rewards for the current reward. The formula for weighted moving average is show in Equation 2.

$$R_{WMA} = \frac{n-1}{n} R_{WMA} + \frac{1}{n} R_{new} \qquad (2)$$

R_{new} represents the newly calculated reward. R_{WMA} encodes the performance of both agents and significantly improves their ability to work together. In our implementation, we use a weighted moving average using $n = 3$. Consider the following sequence of rewards: x, y, z. The reward values x and z both come from actions by agent a and reward value y comes from an action by agent b. This is the type of reward sequence generated by AR. The weighted moving average calculates the current reward using; $\frac{1}{3}$ of the reward from the agent's most recent action, $\frac{2}{9}$ of the reward from the other agents most recent action, $\frac{4}{27}$ of the reward from the agents second most recent action, and $\frac{8}{27}$ of the reward from the WMA of all previous rewards. These agents are working together in the sense that they are encoding each other's performance into the reward value, weighing the impact of their own reward higher than the other agents' reward.

An important note about the environment is that the calculated QoS reward from an agent's action has high variance from the average reward. This is due to the nature of how programs execute, in that cache misses can cause significant performance loss. Thus, if one sequence of instructions has only a few cache misses, but the next sequence of instructions has many cache misses, the reward for each sequence will vary greatly.

The weighted moving average equation accounts for the variance in the reward values, weighting all previous rewards at almost 30% of the current reward. This is important to allow the agent to learn the average reward over time, rather than a highly volatile current reward. Also note that when this reward is used to update the agent's state-action value, the agent applies the Bellman equation which has a learning rate, further reducing the variance and allowing the agent to learn the average reward for the current state.

6.4.2 Random-Action Annealing

An important function of Q-learning is the ability for an agent to learn an environment through random actions. Allowing an agent to take random actions during the initial phases of execution is important for the agent to learn about its environment without worrying about negative consequences. This is a critical learning phase, especially if the reward space is multi-dimensional with many local maxima or minima.

As mentioned in Section 4.4, a Q-Learning agent uses a variable ϵ for its percent chance of a random action. We propose Random-Action Annealing (RAA) where ϵ is initialized to a high value and decreased over time until it reaches a minimum value. Figure 2 shows an agent who's ϵ is initialized with an 80% chance of taking a random action. The agent decreased its chance of random actions from 80% to 8% over 128 iterations. The agent then maintained the 8% chance of random action.

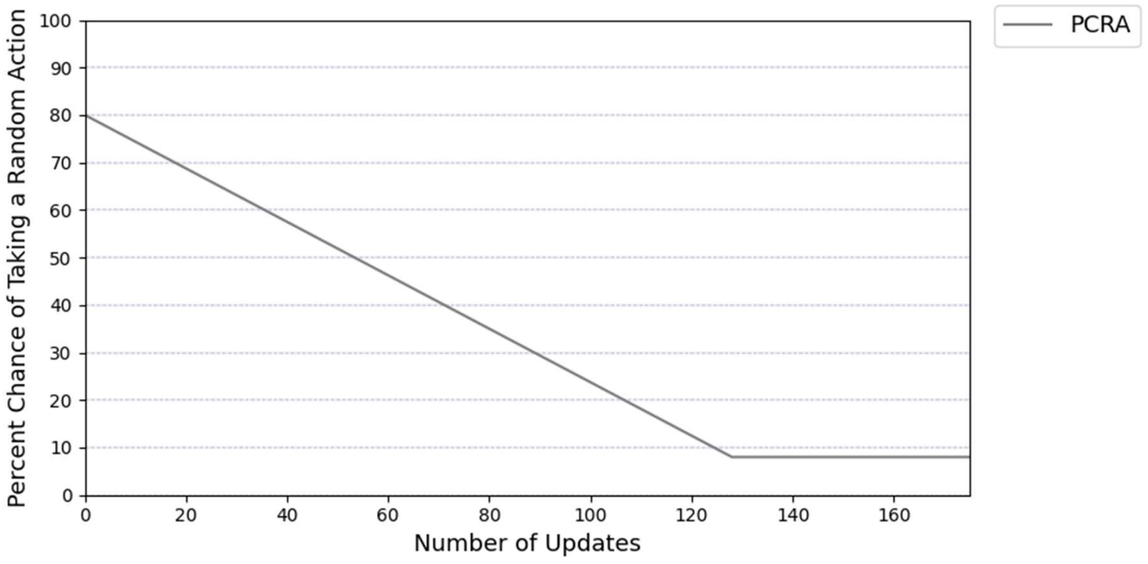


Figure 2: Agent Annealing percent chance of Random Action.

In a static environment with few states this method works well to help the agent build a quick understanding of its environment. However, an agent in a dynamic environment must have the ability to explore new environments quickly. A simple method to allow the agent to explore its new environment is to raise its ϵ value and apply Random-Action Annealing.

When an agent detects a major change in its environment, the agent raises its ϵ value and anneals the value to a defined minimum ϵ value. The number of iterations for the agent to anneal its PCRA value is a hyperparameter that must be further examined. The agent detects major changes in its environment by comparing long term rewards vs. short term rewards. The long-term reward and short-term rewards are calculated using a weighted moving average as described in Equation 3. An implementation for this is shown in Equation 4. In this implementation, the long-term reward is calculated using n=100 and the short term is using n=10.

$$R_{long} = \frac{99}{100} * R_{long} + \frac{1}{100} * R_{immediate} \quad (3)$$

$$R_{shor} \ = \frac{9}{10} * R_{short} + \frac{1}{10} * R_{immediate} \quad (4)$$

If the agent detects a large discrepancy between the long-term and short-term reward, the agent raises its PCRA and applies Random-Action Annealing. Future work is necessary to determine an optimal threshold for the discrepancy between the long-term and short-term IPC and the optimal n values for the long and short term reward calculations.

6.4.3 Top Down Convergence

Traditional Q-Learning implementations initialize all rewards to zero. This means that the first state an agent explores will immediately have a higher reward value than all other states (assuming a non-negative reward). The agent in this case is trying to prove that certain state-action pairs are better than the others. Thus, in a static environment, the agent will always take the same sequence of actions assuming no random actions. Clearly, an agent initialized with all rewards initialized to zero has restricted learning potential and commonly leads to the agent learning a sub-optimal policy.

The alternative to initializing all rewards to zero is to set all the rewards to a value close to the estimated optimal reward. Instead of proving a state-action pair is beneficial, the agent is proving that a state-action pair is not-beneficial, lowering it compared to the other possible states. Using this policy, the agent is forced to explore widely initially, proving that certain states are immediately worse than others and should not be revisited. We dub this policy Top Down Convergence (TDC).

In our Q-learning implementation, both the ACL and DPL prefetchers are restricted to a few states with three actions. Thus, our agent can afford to explore all possible state-action pairs without incurring too much performance penalty. This is called top-down convergence because all rewards are initialized higher than expected and are lowered such that the optimal state-action pair is revealed by process of elimination.

An agent using Top Down Convergence, converges on the optimal solution by first assuming all states are potentially beneficial. Then through exploration, the agent proves that some states offer worse rewards than others, eventually converging on the optimal solution. The name Top Down Convergence makes sense because all rewards are initialized high and are lowered such that the optimal state-action pair is revealed through process of elimination.

Top Down Convergence works well when the agent's Q-table size is small. This is because Top Down Convergence essentially forces the agent to explore all possible state-action pairs and agents, while only a few state-action pairs can afford this exploration. However, Q-learning agents with hundreds of state-action pairs must initialize all rewards to zero, otherwise the agent will spend too much time proving actions are poor, rather than discovering a good policy.

Top Down Convergence can be paired with Random-Action Annealing such that when the agent detects a significant change in its environment, both the ϵ is increased and all Q-table rewards are set higher than new short-term reward. Top Down Convergence and Random-Action Annealing significantly improve an agent's ability to explore, giving it a better chance at discovering the globally optimal policy.

6.4.4 Set on Increase or Decrease

Regularization is a powerful technique used to improve a model's generality. Regularization is used in deep learning models that require training and testing phases to prevent the model from overfitting to the training data [17]. Reinforcement learning models use regularization to simplify an agent's understanding of its environment by taking the agents' large search space and reducing the problem's complexity via a regularization policy, thus deriving a faster and simpler solution [18].

The Set on Increase or Decrease (SID) optimization technique occurs when the agent selects the increase or decrease action. For example, if an agent's current state is aggressiveness degree 1 and it takes an action to increase its aggressiveness to degree 2, the agent will not use the reward it received by transitioning from degree 1 to degree 2, but rather set the "increase aggressiveness" action from degree 1 with the "no change" action from degree 2.

In an environment where rewards for each state are constant, the "increase aggressiveness" action for degree 1 and "no change" action for degree 2 will converge to the same value. Similarly, the "decrease aggressiveness" action from degree 2 and "no change" action for degree 1 will converge to the same value. Thus, the agent is trying to learn duplicate reward values unnecessarily. Furthermore, the "increase or decrease aggressiveness" actions are less frequent than the "no change" action after a period of time, thus learning the correct reward for increase or decrease actions is more difficult.

Our solution eliminates the need to learn the same value twice by setting the "increase aggressiveness" reward to the adjacent higher degree "no change" action reward value, and the "decrease aggressiveness" reward to the adjacent lower degree "no change" action reward value.

This optimization technique is a form of regularization for Q-learning. The Set on Increase or Decrease technique can be derived from the Bellman reward equation shown in Equation 1, by setting the discount factor to a high value and making the learning rate the inverse of the discount factor; alpha is the inverse of gamma. As the learning rate approaches zero, the difference between the reward and the old state-action reward approaches zero, and the most rewarding action in the new state becomes our new reward. Prior to executing this function, the agent must set the Q (S, A) reward to zero to prevent the Bellman equation from becoming a summation driving towards infinity.

Now consider a ratio of alpha to gamma shown in the following expression.

$$\alpha{:}\gamma = \frac{1}{n} : (n - m) | \, n \gg m \qquad (5)$$

Note alpha and gamma are still approaching infinity. In this expression, we are adding a transition buffer requiring that an increase or decrease in aggressiveness demonstrate an (m/n) percent improvement over the current aggressiveness. This value is adjustable based on the volatility of the environment, to increase or decrease the difficulty of a state transition.

Table 3 shows an example of the Q-table for a prefetcher after many updates in a steady environment where n = 100 and m = 1.

Degree	Decrease	Same	Increase
1	0.99	1.00	1.98
2	0.99	2.00	2.18
3	1.98	2.20	0.99
4	2.18	1.00	0.49
5	0.99	0.50	0.19
6	0.49	0.20	0.19

Table 3: Sample situation after reaching steady state using the set on increase or decrease technique.

From this table we can see that degree 3 is optimal in this environment with the largest reward of 2.20. We also see that each increase or decrease action reward is 99% of the corresponding next state "same" action reward. This technique has a strong regularization affect and forces quicker convergence.

6.4.5 Quick Convergence Update

One challenge with the Set on Increase or Decrease technique is when two aggressiveness degrees are very close in performance. As previously noted, the increase or decrease action is only taken if the new action presents an improvement against a threshold (n/m).

If two degrees show performance within that threshold, a saddle point occurs in the Q-table and the agent may be caught at a sub-optimal configuration. Consider the situation in Table 4 where the agent selected random actions and resides in degree 1 highlighted in yellow.

Degree	Decrease	Same	Increase
1	0.99	1.00	0.99
2	0.99	1.00	4.95
3	0.99	5.00	2.97
4	4.95	3.00	1.98
5	2.97	2.00	0.99
6	1.98	1.00	0.99

Table 4: Sample situation after reaching steady state using the previous algorithms.

In this situation, the optimal configuration is degree 3, however, the agent is stuck at degree 1 and will not choose to increase its aggression except by a random action because degree 2 has the same performance. Therefore, we need an algorithm to periodically sweep across the table and propagate valuable rewards through the increase and decrease action reward values.

We introduce the technique of Quick Convergence Update (QCU) where the agent updates all "increase aggressiveness" actions based on the adjacent "no change" and "increase aggressiveness" actions and all "decrease aggressiveness" actions based on the adjacent "no change" and decrease aggressiveness" actions. The algorithm still uses the hysteresis threshold, but instead of only using the "no change" action it also values the corresponding increase or decrease action.

Table 5 shows the results of applying a Quick Convergence Update.

Degree	Decrease	Same	Increase
1	0.99	1.00	**4.90**
2	0.99	1.00	4.95
3	0.99	5.00	2.97
4	4.95	3.00	1.98
5	**4.90**	2.00	0.99
6	**4.85**	1.00	0.99

Table 5: Results after applying a Quick Convergence Update to the situation shown in Table 4.

The rewards shown in bold are the updated values after a Quick Convergence Update. The agent previously stuck at degree 1 sees that there exists a different state with a higher reward if the agent increases its aggressiveness. After applying this technique, the agent will move towards the optimal state quickly.

6.4.6 Stay-Six

An important issue we addressed was the rate of convergence when entering a new state. To do this, we created a solution dubbed Stay-Six (SS). When an agent transitions to a new state, the agent might only get one opportunity to explore the new state before transitioning to another state. The issue is the agent did not have a fair opportunity to converge to the real reward at that location.

The Stay-Six algorithm addresses this issue by requiring that the agent spend a minimum of six updates at the new state before transitioning to a new state. Figure 3 shows the convergence of four reward values using the Bellman equation.

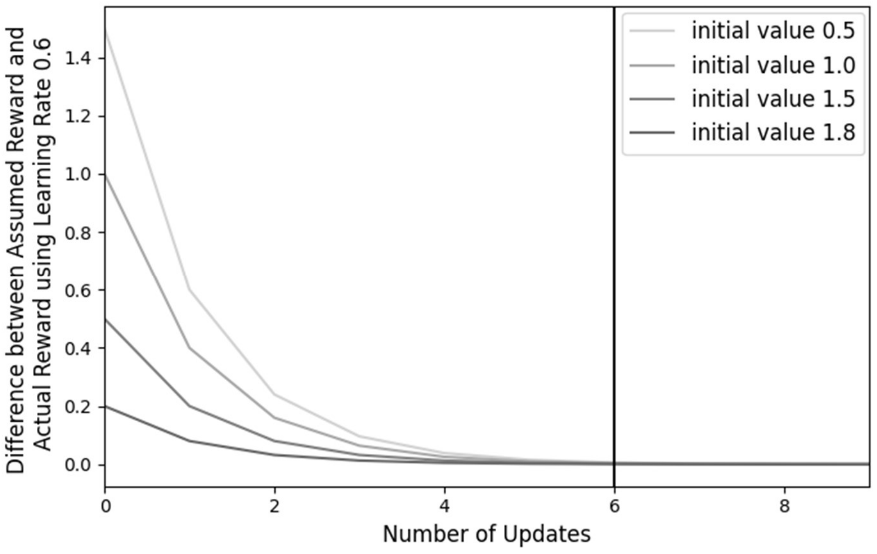


Figure 3: Q-table convergence using the Bellman equation with a learning rate of 0.6 after staying in one state for 10 updates.

The final reward value is 2.0, and each reward value shown starts at a different initial value. We see that the Bellman equation converges to the average reward after 6 updates even if the difference between the initial reward and the average reward is significantly different. The lightest line is initialized at 0.5, initialized at 25% of the real reward. Requiring more time for convergence is likely unnecessary because the difference between the initial reward and the new reward will be minimal after the agent explores the environment.

7 Evaluation Methodology

We evaluated our model using a simulator because current hardware does not allow a user to configure the aggressiveness degree dynamically. In this section, first we present the Sniper simulator, used to simulate ADM. Then we explain our choice of workloads used to evaluate the model. Finally, we discuss the Quality of Service metric we used.

7.1 System Configuration

We used the Sniper simulator invented by Ibrahim Hur and Wim Heirman [9]. This simulator is a next generation, parallel, high-speed x86 simulator based on an interval core model and Graphite simulation infrastructure, allowing for fast and accurate simulation when exploring different homogeneous and heterogeneous multi-core architectures [9]. Sniper has an average performance prediction error within 25% at simulation speeds of several MIPS. Source code for Sniper is available to Intel employees through permission of the inventors who work at the Intel ExaScience Lab.

The Sniper simulator was useful for this research because it allowed us to simulate the design for ADM. Sniper has high precision, but average accuracy, which allowed us to make architectural comparisons using simulation results generated by Sniper. A performance increase in Sniper due to an architectural change will scale proportionally to real hardware. We conducted architecture comparisons by simulating different aggressiveness degree configurations. The simulation dumped results into a file containing performance counters for each of the major hardware components such as; number of executed instructions, number of cycles per core, TLB accesses and misses, L1-I, L1-D, L2 and L3 cache hits, misses, pending hits, unused evictions, etc. This file provides information across the entire system and information localized to each core. We quantified and visualized the results from the results files.

The Sniper simulator needs a type of architecture configuration to use for simulation. We used a Skylake Server configuration featuring a 32KiB 8-way private L1 Data cache, a 1MiB 16-way private L2-cache and a 1.375 MiB per core 11-way shared L3-cache. Caching information is the most relevant feature of the Sniper simulation because we are modifying the aggressiveness of L2 prefetchers which mainly relates to the memory subsystem. Intel introduced the Skylake system to industry in 2017 and is the most relevant architecture model we could use for comparisons to current architectures [20].

Simulating a programs execution in software generally takes a long time. One feature of Sniper is that it can “Fast Forward” through uninteresting regions of code. Then Sniper can enter “Detailed” mode where it captures all performance statistics during that region. For our studies, we applied this sampling technique to generate results from multiple regions of interest in each workload, capturing specific behaviors. Exact sampling regions are detailed in Appendix B. We sampled ~1% of each benchmark total instruction count in “Detailed” mode. This type of sampling is useful because short regions of interest capture moments when one aggressiveness degree is more beneficial than the others, and in most cases, creating a convex optimization space with only one local minima. This observation is derived from two characteristics. First, one static configuration is optimal throughout the entirety of the region. Second, the surrounding static configurations follow a bell-shaped performance curve. Thus, the local minima is also the global minima in a convex optimization space.

7.2 Workload Selection

We used the Standard Performance Evaluation Corporation (SPEC) CPU2017 benchmark suite to evaluate the performance of the ADM. The SPEC benchmark suite contains next-generation, industry-standardized, CPU intensive workloads for measuring and comparing compute intensive performance, stressing a system's processor, memory subsystem and compiler [21]. By design, these benchmarks are useful for evaluating the success of simulation-based designs and optimization research for next-generation processors, memory subsystems and compilers [14]. Therefore, using this set of benchmarks made sense for evaluating the performance of the ADM. Descriptions for all benchmarks from the SPEC 2017 suite can be found in Appendix C.

Since the ADM agent seeks performance gains by optimizing the MLC prefetchers, the workloads useful for evaluation must be MLC prefetch sensitive. Published performance characterizations by Navarro-Torres provided a preliminary list of prefetch sensitive workloads from the SPEC suite [22]. These results were found by enabling and disabling all prefetchers at all cache levels. The workloads with the greatest prefetcher sensitivity were 500.perlbench_r, 502.gcc_r, 505.mcf_r, 507.cactuBSSN_r, 510.parest_r, 519.lbm_r, 520.omnetpp_r, 521.wrf_r, 523.xalancbmk_r, 527.cam4_r, 549.fotonik3d_r, and 544.roms_r.

Using the Sniper simulator, we performed a similar evaluation of the SPEC CPU2017 workloads, isolating the sensitivity of the MLC prefetchers on each workload. We conducted this experiment by enabling or disabling all MLC prefetchers. We found similar results to Navarro-Torres but we removed 527.cam4_r and 521.wrf_r from the evaluation set because they did not show MLC prefetcher sensitivity.

7.3 Quality of Service Models

A Quality of Service (QoS) metric is a function of statistics, used to evaluate and compare systems. QoS metrics, such as performance, direct the evolution of technology. In this section we examine performance QoS metrics and how they are used in practice.

Performance is best defined as a comparison of execution time. Given two computer systems, the system that can execute a workload faster has the higher performance. Execution time is the only metric of performance that makes sense for comparing architectures that do not share the same fundamental architecture design – instruction set architecture, clock frequency, pipeline stages, etc. Systems with similar architectures can be compared using a performance metric which combines instructions, cycles, and frequency. Metrics such as Instructions Per Cycle (IPC) and Millions of Instructions Per Second (MIPS) combine these metrics. IPC and MIPS use instruction count per unit time to evaluate performance. IPC can be converted to MIPS by dividing IPC by the clock frequency and scaling by 10^6. If two systems use the same clock frequency, these metrics encode the same information. The benefit of using IPC over MIPS is that IPC is a bounded metric. The theoretical maximum IPC is the number of pipelines in a system, whereas the maximum MIPS depends on the clock frequency and pipeline stages. Since IPC is non-negative and a higher IPC represents higher performance, graphs of IPC are useful visual comparisons to quickly understand relative system performance. We use IPC as our performance QoS metric to compare the ADM model against other hardware configurations.

8 ADM Performance Evaluation

In this section, we demonstrate the impact of an ADM agent on prefetch sensitive workloads from the SPEC CPU2017 suite. We consider two experiments; a single prefetcher implementation and a dual prefetcher implementation. Both experiments compare the ADM agent against static prefetcher aggressiveness configurations. Static prefetcher configurations are useful for comparisons because we sample programs over short regions of interest where a single static configuration is optimal for the entirety of the region.

We also present an investigation about the impact of the optimization techniques applied to the 500.perlbench_r, 507.cactuBSSN_r and 523.xalankbmk_r benchmarks. We demonstrate the effects of the ADM agent by progressively adding optimization techniques. The results demonstrate rapid convergence for both workloads analyzed.

8.1 Experimental Setup

To implement an ADM Q-learning agent for single MLC prefetcher, we enabled either the ACL prefetcher or the DPL prefetcher with an ADM agent. For dual MLC prefetchers, we enabled both the ACL and DPL prefetchers, each with its own ADM agent. The ADM agents used the current aggressiveness degree as the state variables. The actions were to increase the aggressiveness, to decrease the aggressiveness, or to not change the aggressiveness. The DPL aggressiveness degree varied from zero to eight lines, and the ACL aggressiveness degree varied from zero to six lines. This means that if the DPL prefetcher detects a stride, it may prefetch up to eight stride-separated cache-lines. Similarly, the ACL prefetcher may prefetch up to six adjacent cache-lines in a single request. Thus, the Q-table for the DPL prefetcher had nine states, and the Q-table for ACL prefetcher had seven states.

We updated the aggressiveness of the prefetcher on intervals of 128,000 instructions; this period provides ample time for the prefetcher configuration to impact the performance as discussed in Section 6.2. At each update, the Q-learning agent takes a random action eight percent of the time, and an action predicting the maximum reward ninety-two percent of the time. The reward function for each action taken uses a weighted moving average of the IPC.

Using a grid search, we derived the hyperparameters which are the DPL and ACL degree ranges, update interval, percent chance of a random action, weights for the weighted moving average. Further research is needed to optimally choose these hyperparameters for a larger set of workloads. The necessary investigations include those that explore the impact of the state degree values, interval size, percent change of a random action, and reward function.

We evaluated our aggressiveness-degree tuner on a subset of ten workloads from the SPEC 2017 suite exhibiting prefetch sensitivity. We compared eight static prefetcher aggressiveness degrees for both the ACL and DPL prefetchers against the ADM agent: static aggressiveness degrees zero, one, two, four, eight, sixteen and thirty-two.

Short regions of interest have a convex optimization space over changes in aggressiveness degree. We derived this from two observable characteristics. First, one static configuration was optimal throughout the entirety of the region. Second, the surrounding static configurations followed a bell-shaped performance curve. Therefore, our results were not expected to outperform any single aggressiveness-degree, but rather expected to converge on the optimal aggressiveness degree within each region. Instances where the ADM agent outperformed a static configuration for an entire workload were due to different optimal prefetcher aggressiveness degrees for different regions of interest.

8.2 Results for the ADM agent applied to the ACL Prefetcher

The Q-learning agent converged to the globally optimal aggressiveness degree within each region for the APL prefetcher and, in some cases, found a higher performing configuration than the static aggressiveness degrees.

We isolated the effects of the Q-learning aggressiveness tuner on the ACL prefetcher by disabling the DPL prefetcher. We then compared the seven static aggressiveness degrees with the dynamic Q-learning approach. From the results visualized in Figure 4, we classified each workload based on its optimal aggressiveness degree for the APL prefetcher: 500.perlbench_r, 507.cactuBSSN_r, and 523.xalancbmk_r are degree 1 or 2 workloads; 502.gcc_r, 505.mcf_r, 510.parest_r and 554.roms_r are prefetch degree 4 workloads; 519.lbm_r and 549.fotonik_r are degree 8 workloads; and 520.omnetpp _r is a degree 0 workload.

The ADM agent converged to each class of aggressiveness degrees. The Q-learning approach was within 1% of the optimal configuration for 500.perlbench_r, 505.mcf_r, 507.cactuBSSN_r, and 510.parest_r, and exceeded performance of the static configurations for 502.gcc_r, 523.xalancbmk_r, and 549.fotonik_r.

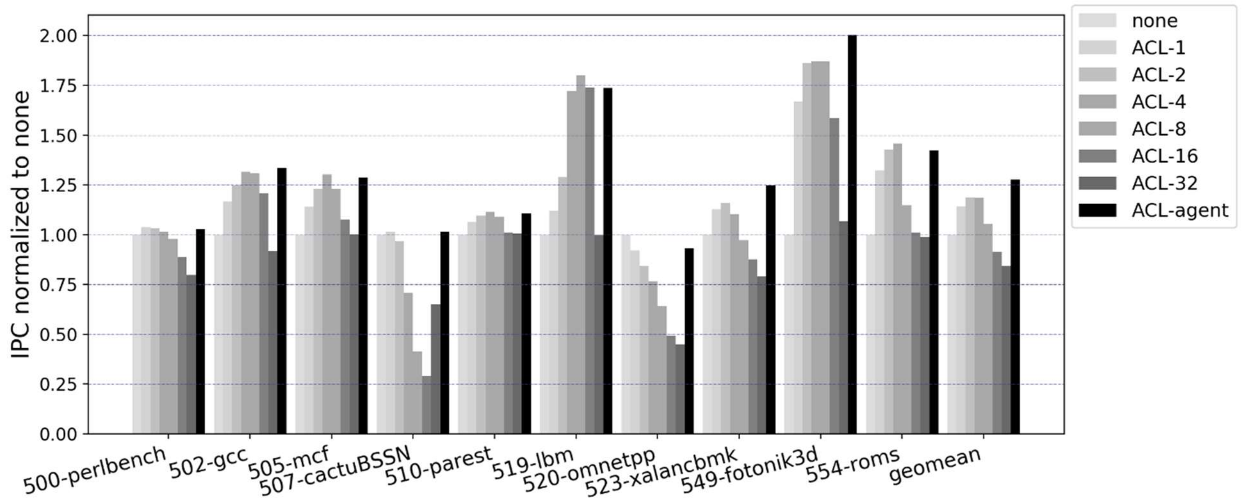


Figure 4: IPC performance comparison normalized to no prefetching for prefetcher aggressiveness degrees applied to APL prefetcher. Higher IPC values are better.

A Q-learning aggressiveness tuner applied to the APL prefetcher on average performs 20% better than the average of the static aggressiveness configurations, and 6.5% better than the optimal static aggressiveness configuration – degree 4. Furthermore, the Q-learning approach has the lowest variance from the optimal configuration, on average 1.5% of the optimal. This is a useful metric because it indicates that there is only 1.5% potential improvement using the ADM agent. The best static prefetcher shows 7.9% performance loss due to not being the optimal configuration, which is why the ADM agent is a good innovation for this problem.

8.3 Results for the ADM agent applied to the DPL Prefetcher

The Q-learning agent converged to the globally optimal aggressiveness degree within each region for the DPL. We performed a similar test to the APL prefetcher where we isolated the effects of the Q-learning aggressiveness tuner on the DPL prefetcher by disabling the APL prefetcher. We then compared the seven static aggressiveness degrees with the dynamic Q-learning approach. The results for the ADM agent applied to the DPL Prefetcher are shown in Figure 5. The ADM agent was within 2% of the optimal configuration for 500.perlbench_r, 502.gcc_r, 507.cactuBSSN_r, 510.parest_r, 519.lbm_r, 520.omnetpp_r, 523.xalancbmk, and 554.roms_r.

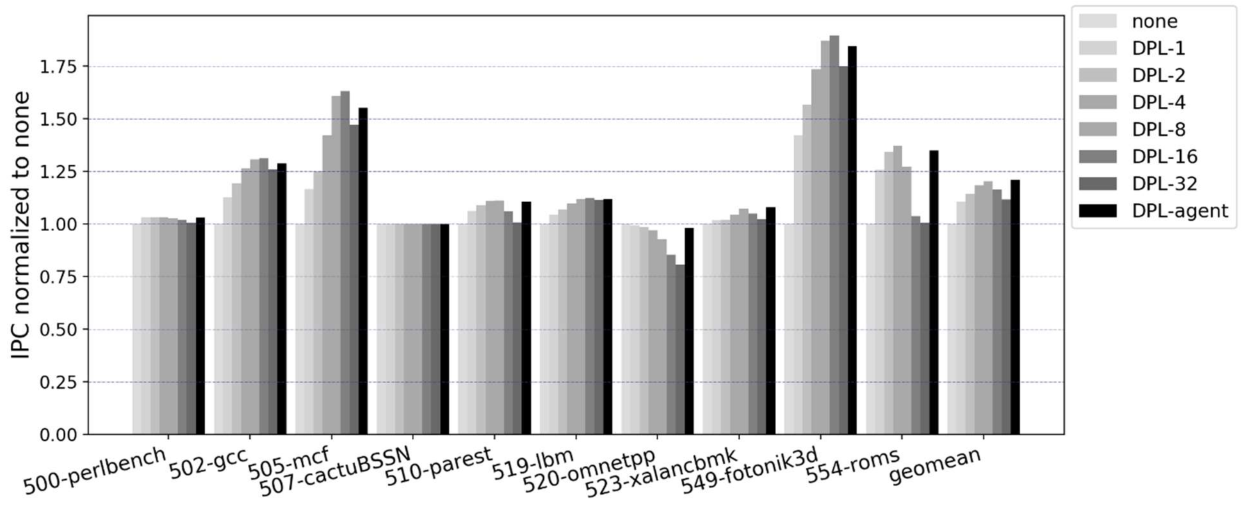


Figure 5: IPC performance comparison normalized to no prefetching for prefetcher aggressiveness degrees applied to DPL prefetcher. Higher IPC values are better.

A Q-learning aggressiveness tuner applied to the DPL prefetcher on average performs 6% better than the average of the static aggressiveness configurations, and .6% better than the optimal static aggressiveness configuration – degree 8. Furthermore, the Q-learning approach has the lowest variance from the optimal configuration, on average 1.4% of the optimal, while the best average static configuration varies 1.9% of the optimal configuration on average. The main reason that the ADM agent does not outperform the static DPL configurations to the same degree as the ADM agent applied to the DPL prefetcher is because the performance variance due to DPL aggressiveness is lower than the APL prefetcher.

8.4 Analysis of a Single Prefetcher ADM Agent

The performance improvement of the ADM agent manifests because of high variance in performance due to changes in prefetcher degree within and between workloads. If there is no variance in performance within a workload, such as 507.cactubssn_r for the DPL prefetcher, then the ADM agent can not demonstrate performance improvement. If there is no variance in performance between workloads, such as 502.gcc_r and 505.mcf_r for the ACL prefetcher with optimal degree four, then the ADM agent cannot show performance improvement. The performance improvement of the ADM agent is a direct result of the variance in optimal prefetcher configuration both between workloads and within each workload.

The following is an example using 505.mcf_r and 507.cactubssn_r for the ACL prefetcher. 507.cactubssn_r has an optimal static prefetcher configuration of degree one and 505.mcf_r has an optimal static prefetcher configuration of degree four. The ADM agent converges on both these configurations. The average of the optimal static configuration performances over no prefetching for these two workloads is 16.0%. The average ADM agent performance is 15.0% improvement over no prefetching. This shows the ADM agent can converge on different optimal configurations. Furthermore, the average performance improvement of static degree one on 505.mcf_r and 507.cactubssn_r is 7.8% and the average performance improvement of static degree four on 505.mcf_r and 507.cactubssn_r is 0.7%. The ADM agent is two times better then degree one and twenty-one times better than degree four using these two workloads. This example demonstrates variance within workloads because the average variance between degree one and four within 505.mcf_r is 16.5% and 507.cactubssn_r is 47.1%. It also demonstrates variance between workloads because the difference between the optimal configurations for each workload is 23.6%.

Tables 6 and 7 show the performance variance between these workloads for the ACL and DPL prefetchers. The rows show performance variance within a workload while the columns show performance variance between workloads.

	Degree								
	0	1	2	4	8	16	32	ADM	average
500.perlbench_r	96.4%	100.0%	99.6%	97.9%	94.2%	85.5%	76.8%	99.0%	93.6%
502.gcc_r	75.0%	87.4%	93.6%	98.6%	98.1%	90.7%	68.8%	100.0%	89.2%
505.mcf_r	76.8%	87.5%	94.4%	100.0%	94.3%	82.5%	76.8%	98.8%	88.8%
507.cactubssn_r	98.5%	100.0%	95.3%	69.8%	40.7%	28.5%	64.3%	99.9%	74.6%
510.parest_r	89.8%	95.5%	98.3%	100.0%	98.0%	90.6%	90.4%	99.4%	95.3%
519.lbm_r	55.6%	62.2%	71.6%	95.6%	100.0%	96.6%	55.6%	96.5%	79.2%
520.omnetpp_r	100.0%	91.9%	84.3%	76.6%	64.0%	49.2%	45.0%	93.2%	75.5%
523.xalancbmk_r	80.2%	90.3%	93.0%	88.5%	78.0%	70.3%	63.4%	100.0%	83.0%
549.fotonik3d_r	49.9%	83.3%	93.0%	93.3%	93.3%	79.1%	53.3%	100.0%	80.7%
554.roms_r	68.7%	90.8%	98.0%	100.0%	78.8%	69.3%	67.8%	97.8%	83.9%
average	79.1%	88.9%	92.1%	92.0%	83.9%	74.2%	66.2%	98.5%	**84.4%**

Table 6: ACL Performance Comparison Against the Optimal Aggressiveness Degree within each Workload.

	Degree								
	0	1	2	4	8	16	32	ADM	average
500.perlbench_r	96.8%	100.0%	100.0%	98.8%	99.4%	98.6%	97.4%	99.7%	98.8%
502.gcc_r	76.2%	85.9%	90.9%	96.4%	99.5%	100.0%	96.1%	98.0%	92.9%
505.mcf_r	61.3%	71.5%	76.6%	87.1%	98.7%	100.0%	90.1%	95.2%	85.1%
507.cactubssn_r	100.0%	100.0%	99.9%	99.9%	99.9%	99.9%	99.9%	99.9%	99.9%
510.parest_r	89.9%	95.5%	98.0%	99.7%	100.0%	95.4%	90.7%	99.5%	96.1%
519.lbm_r	89.0%	92.9%	95.1%	97.6%	99.7%	100.0%	99.1%	99.6%	96.6%
520.omnetpp_r	100.0%	99.3%	98.5%	97.1%	92.7%	85.4%	80.6%	98.0%	94.0%
523.xalancbmk_r	92.7%	94.3%	94.5%	96.7%	99.4%	97.2%	94.8%	100.0%	96.2%
549.fotonik3d_r	52.7%	75.0%	82.7%	91.5%	98.7%	100.0%	92.3%	97.2%	86.3%
554.roms_r	72.9%	91.7%	97.8%	100.0%	92.8%	75.6%	73.3%	98.4%	87.8%
average	83.2%	90.6%	93.4%	96.5%	98.1%	95.2%	91.4%	98.6%	**93.4%**

Table 7: DPL Performance Comparison Against the Optimal Aggressiveness Degree within each Workload.

We note a few interesting observations about the results shown in Tables 6 and 7.

Observation 1: The average performance of the both the static configurations and the ADM agent against the optimal configuration within a workload on the ACL prefetcher is 84.4% while the DPL prefetcher is 93.4%.

Reasoning: This shows that changing the aggressiveness of the ACL prefetcher has greater impact on performance than the DPL prefetcher. The intuition why this is true is because the ACL prefetcher is more active than the DPL prefetcher, thus, the aggressiveness of this prefetcher will cause a significant increase or decrease in number of prefetches, which is the fundamental reason for performance change.

Observation 2: The maximum between-workload average for the ACL prefetcher is degree two at 92.1%.

Reasoning: This is a valuable observation because it shows that there is a maximum of 7.9% performance improvement that can be captured if the agent correctly adjusts the prefetcher degree. The reason this observation occurs is because workloads like 500.perlbench_r and 510.parest_r perform well at aggressiveness degree 2, while workloads like 520.omnetpp_r and 519.lbm_r perform poorly at aggressiveness degree 2. If all workloads performed well with aggressiveness degree 2, then there would be no need for a dynamic agent, however, since there is a 7.9% performance drop on average, a dynamic agent can perform well. The ADM agent captured much of the realizable performance gains, reducing the performance variance against the optimal to 98.5%.

Observation 3: The maximum between-workload average for the DPL prefetcher is degree 8 at 98.1%.

Reasoning: In contrast to Observation 2, Observation 3 highlights the reason why the ADM agent does not show as significant performance increase for the DPL prefetcher as it does for the APL prefetcher. Since aggressiveness degree 8 is already close to optimal with only a 1.9% performance improvement potential, the ADM agent can only achieve a maximum performance speedup of under 2% by correctly adjusting the prefetcher degree. The ADM agent captures a small amount of performance improvement, outperforming the static aggressiveness degree 8 by 0.6%.

8.5 Results for ADM agent applied to both the APL and the DPL Prefetchers

The dual ADM agents converged to the globally optimal aggressiveness degree configuration for both the APL and DPL prefetchers, and in some cases found a higher performing configuration than the static aggressiveness degrees.

We classified each workload based on its optimal static aggressiveness degree for the APL/DPL prefetchers using the results visualized in Figure 6: 500.perlbench_r, 507.cactuBSSN_r, and 523.xalancbmk_r are low-aggressiveness; 502.gcc_r, 505.mcf_r, 510.parest_r, 549.fotonik3d_r, and 554.roms_r are mid-aggressiveness; 519.lbm_r is high-aggressiveness; and 520.omnetpp_r is a no prefetching workload. The ADM agent converged to each class of aggressiveness degrees, within 3% of the optimal configuration for 500.perlbench_r, 505.mcf_r, 507.cactuBSSN_r, 510.parest_r and 519.lbm_r. The agents exceeded performance of the static configurations for 502.gcc_r and 523.xalancbmk_r.

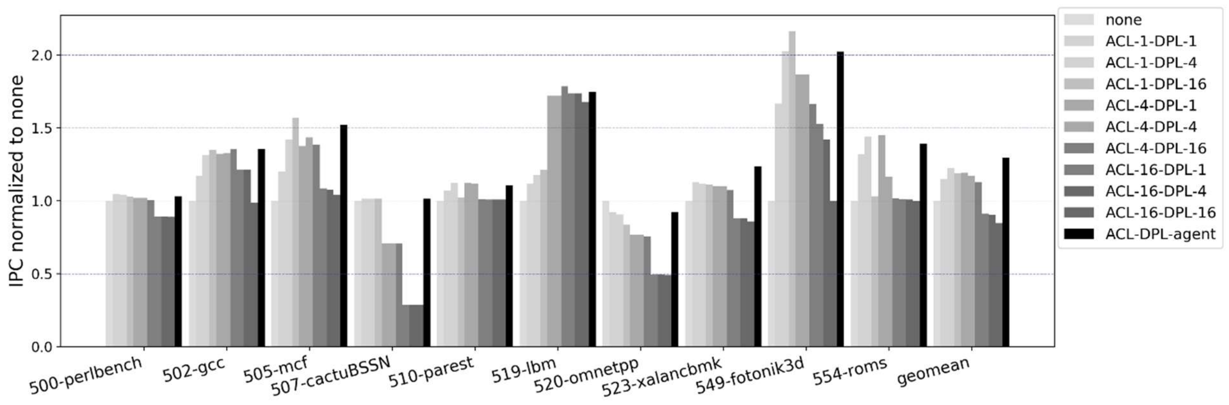


Figure 6: IPC performance comparison normalized to no prefetching for prefetcher aggressiveness degrees applied both APL and DPL prefetchers.

The ADM agent performed the worst on 520.omnetpp_r, with a 9.2% decrease in performance against the optimal static configuration for this benchmark, ACL aggressiveness degree zero, DPL aggressiveness degree zero. This decrease in performance is explained in part by taking random actions, a feature of Random-Action Annealing. This hyperparameter was initialized to 8% random actions. When the ADM agents took a random action and left the degree 0 state, the performance was immediately negatively impacted. This brief period of degree 1 for either ACL or DPL prefetchers significantly decreased the overall performance of the agent.

Another interesting workload was 554.roms_r because the DPL prefetcher had a large impact on its performance relative to other workloads. Three static configurations outperformed the others: ACL1-DPL1, ACL1-DPL4, ACL4-DPL1. Low to mid aggressiveness prefetcher configurations performed well. The higher performing configurations were when the ACL prefetcher is low aggressiveness and the DPL prefetcher is mid-aggressiveness or vice versa. We do not see perfect convergence for 554.roms_r because finding this unique configuration is difficult in a dynamic environment. Any time the ACL prefetcher and the DPL prefetcher explored higher aggressiveness states simultaneously, the performance was negatively impacted. This is another reason why the ADM agent did not have a higher performance that was closer to the optimal static configuration.

A dual Q-learning aggressiveness tuner applied to the APL prefetcher and DPL prefetcher on average performs 17% better than the average of the static aggressiveness configurations, and 4.2% better than the optimal static aggressiveness configuration (APL degree 1, DPL degree 4). Furthermore, the Q-learning approach has the lowest variance from the optimal configuration, on average 2.6% of the optimal, while the best average static configuration performs 7.3% of the optimal configuration on average.

8.6 Impact of Optimization Techniques on Improving Convergence

In this section, we examine the impact of the optimization techniques described in Section 6.4, specifically analyzing 507.cactuBSSN_r, 500.perlbench_r, and 523.xalancbmk_r. 507.cactuBSSN_r is an interesting workload because it has one aggressiveness configuration that is significantly better than all the others and the IPC performance has very little variance during execution. 500.perlbench_r is an interesting workload because it has discontinuities in IPC every 1 million instructions. An agent operating on 500.perlbench_r needs to demonstrate fast convergence and learn new policies quickly when faced with a changing environment.

8.6.1 507.cactuBSSN_r: Random Action Annealing and Top Down Convergence

The first two optimization techniques we implemented were Random Action Annealing (RAA) and Top Down Convergence (TDC). Figure 7 illustrates the impact of these optimization techniques. Notice that the region of interest shown in this example uses 2250 Q-table updates. For a full workload analysis, we used 40,000-100,000 Q-table updates. We used a shorter region of interest to conduct faster testing which helped us rapidly develop the optimization techniques in Sniper. A shorter testing period also clarifies how the agent is performing at a finer granularity.

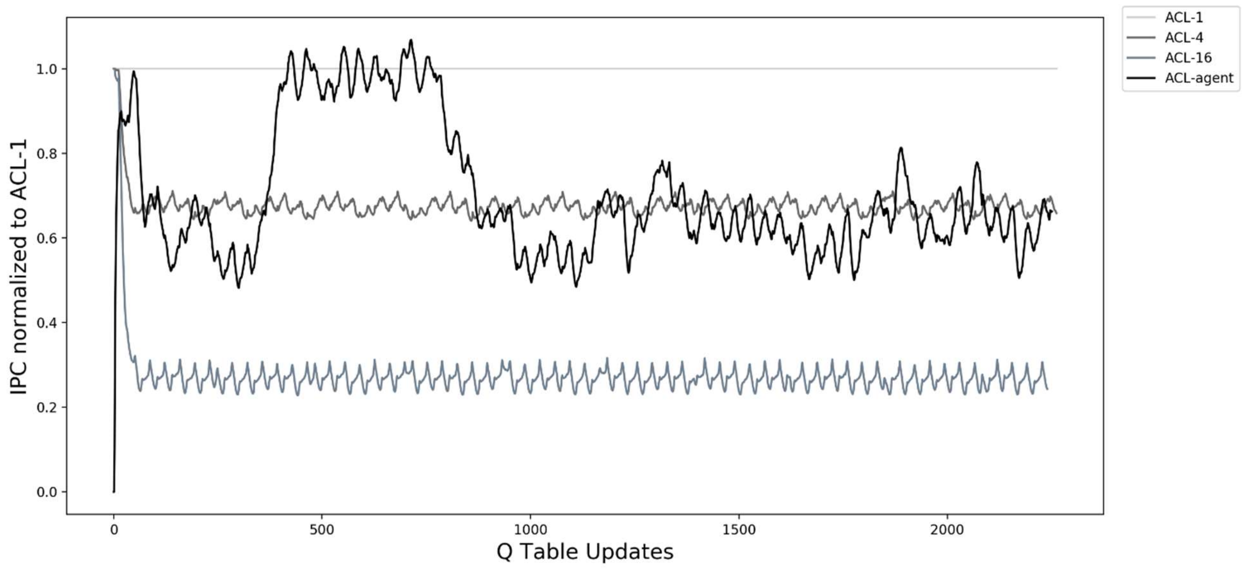


Figure 7: ADM agent with RAA and TDC. The graph compares the ADM agent's performance against two static configurations, ACL degree 1, ACL degree 4 and ACL degree 16. The performance of each run is normalized against ACL degree 1 shown as the top light gray line with performance of 1.0. ACL degree 4 is the second best static degree at 75% of the performance of ACL degree 1, and ACL degree 16 is the worst performance at 30% of ACL degree 1.

The preliminary exploratory effects of Random-Action Annealing are not clear for this workload because it is only applied for the first 128 updates. During these updates, the agent randomly explored its environment, first at the lower aggressiveness degrees, then the higher aggressiveness degrees. The agent could take a random action 8% of the time after the first 128 updates. Random-Action Annealing had minimal effect on performance because there were no discontinuities during the execution of this workload. In a later example using 500.perlbench_r, Random-Action Annealing had a much clearer effect on performance because of major IPC discontinuities every 1000 updates.

The Top Down Convergence method had a strong impact on performance for this workload. Using Top Down Convergence, the agent initialized all the values in its Q-table higher than expected. To accomplish this, the ADM agent first discovered an average initial IPC of 1.0 for 507.cactubssn_r using the default aggressiveness degree 1. The agent increased this value by 20% and initialized all the rewards in the Q-table to a value of 1.2. This value is appropriate for a Top Down Convergence initialization because 1.2 is greater than all static configuration rewards so all states are being “pulled down”. If the agent was initialized to a static configuration degree of four, then then the preliminary calculated IPC would have been around 0.8 so the scaled initialization value would have just been high enough for Top Down Convergence to work well indicating that a scaling factor of 20% works well for this workload.

Here we analyze how the agent acted during discrete update periods. During update periods 0 – 490, the agent had the opportunity to explore through random actions and learn characteristics about its environment. The agent mostly explored the mid-aggressiveness degrees. After 490 updates, the agent decreases its aggressiveness and learned the benefit of low aggressiveness. The agent remained in this state for 300 updates. Then the agent had a sequence of random actions and explored higher aggressive configurations. This is where the Top Down Convergence method fails.

After 800 updates the agent explores the higher aggressiveness states and becomes “stuck” because the state transitions “increase aggressiveness” was not pulled down to the correct reward value. In fact, the transition actions at degree two and three only received five total updates from updates 500 to 1000. Furthermore, the “increase aggressiveness” action updates between 800 – 1000 were receiving rewards still influenced by the lower aggressiveness, biasing the reward values higher than expected, causing the agent to appear “stuck” at degree four.

Top Down Convergence ensures that the agent explores each state at least once. This led to the initial success where the agent found the correct aggressiveness degree, but then failed through a sequence of random actions which caused the agent to become “stuck” at degree four. Over a longer period, the agent will become “unstuck” by “pulling down” the degree 2 and 3 “increase aggressiveness” actions to its actual reward; however, we need the agent to converge on an optimal configuration faster. We improve the agent using the Set on Increase or Decrease optimization technique.

8.6.2 507.cactuBSSN_r: Set on Increase or Decrease

The next set of optimization technique implemented was Set on Increase or Decrease (SID). Figure 8 illustrates the impact of this optimization technique when combined with RAA and Top Down Convergence.

The results shown in Figure 8 show oscillations between the high aggressiveness degrees and low aggressiveness degrees. Oscillations occur because both low and high aggressiveness rewards are being “pulled down” simultaneously. Once the low-aggressiveness state becomes less profitable, the agent changes to the high-aggressiveness degree, until the high-aggressiveness state becomes less profitable. Then the agent changes back to the low-aggressiveness degree. This process repeats until the high-aggressiveness is proven to be worse than low-aggressiveness, where the agent converges to the optimal solution.

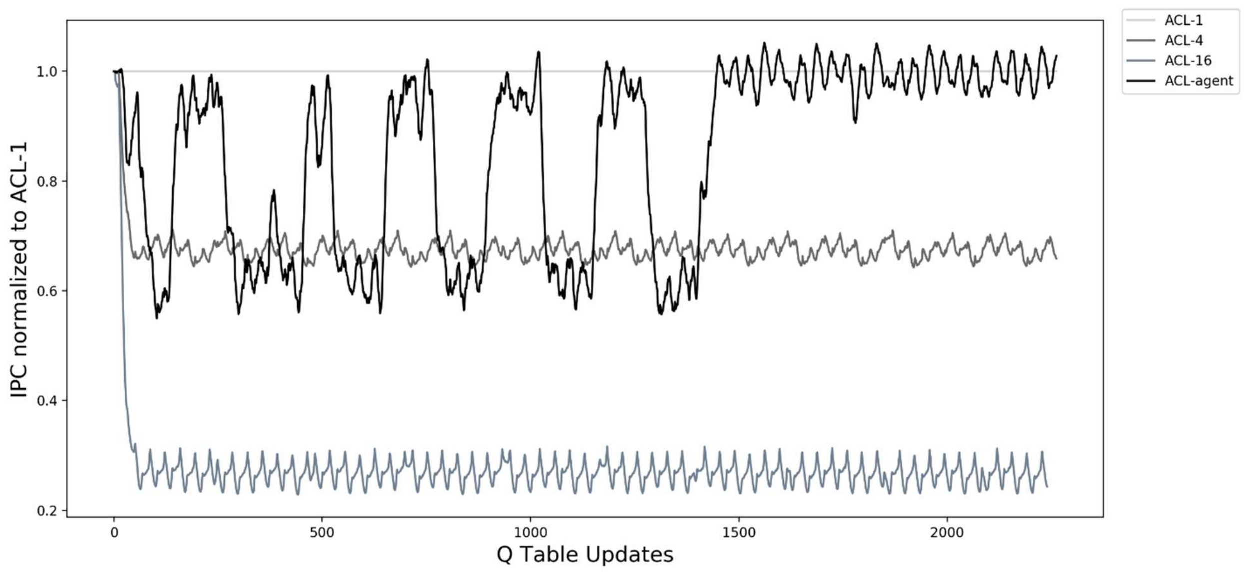


Figure 8: ADM with RAA, TDC, and SID.

The Set on Increase or Decrease technique reduces the search space of the agent by two-thirds since it no longer learns the increase or decrease actions, rather sets them based on the next state non-transition reward. This significantly impacts the time to converge on the optimal result. In this example, we used a small learning rate of 0.01. This was a mistake because it required hundreds of updates to "pull down" each reward to the correct value to prove that one state is significantly better than the others. The agent converged after 1500 updates because the agent had to test the high and low aggressiveness degrees many times before proving the high aggressiveness was not beneficial.

In later tests we increased the learning rate to 0.6 which reduced the oscillations to one to two instead of six. The test with the small learning rate is a great example to demonstrate the impact of the Set on Increase or Decrease technique and its interactions with Top Down Convergence.

8.6.3 507.cactuBSSN_r: Quick Convergence Update and Stay 6

The last optimization techniques we implemented dubbed Quick Convergence Update (QCU) and Stay 6 (SS) prevented the agent form exploring states if one state proved to be worse than an known states. This final example combines all the techniques previously mentioned – Random-Action Annealing, Top Down Convergence, and Set on Increase or Decrease. Now, the ADM agent quickly realizes which aggressiveness degree is optimal. Figure 9 shows the performance of the ADM agent where it only explored the mid-aggressiveness once before identifying that low aggressiveness is optimal.

The Quick Convergence Update is the driving force behind stopping the oscillations because it caused the agent to realize when one set of actions was worse than other actions. In this case, the agent proved that the mid-aggressiveness degree was worse than the low-aggressiveness degrees, so the agent updated all the increase or decrease values appropriately, directing the agent by weighting actions towards the maximum attainable reward. ADM converges after 200 updates using these techniques.

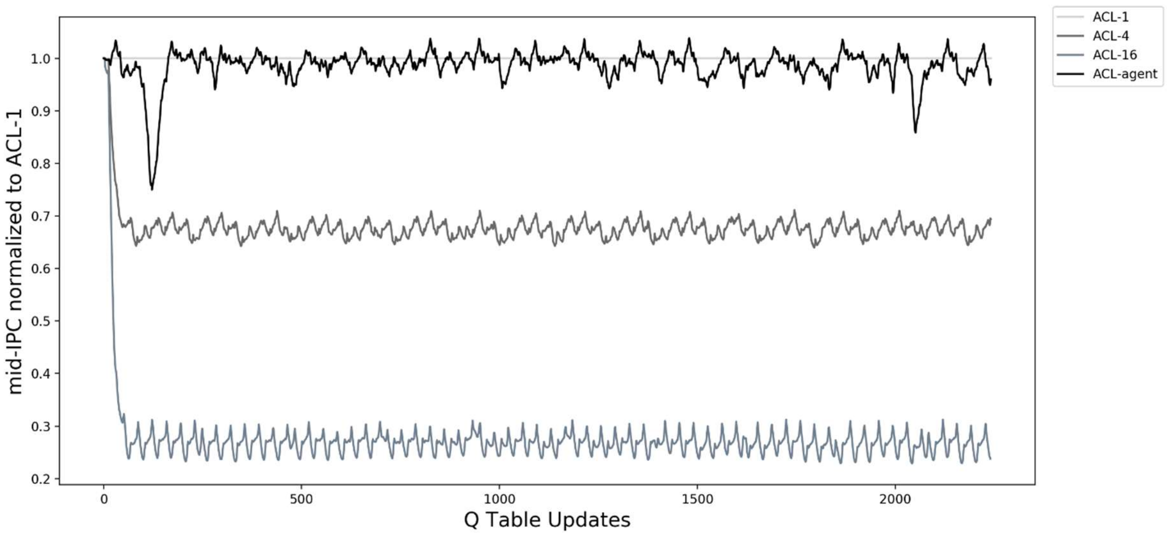


Figure 9: Performance graph of ADM with the stay 6 algorithm.

The combination of Quick Convergence Update and Stay Six is powerful because it forces the agent to learn the "real reward" at all states it encounters before proceeding to other states. In this case, the agent explored degree zero, one, and two, three and four. Then the agent did a Quick Convergence Update where it realized that degrees two, three and four were performing worse than the next lower degree. The Quick Convergence Update indicated to the agent that exploring higher aggressiveness degrees would likely be worse than the states already explored. The Quick Convergence Update updated all the increase and decrease aggressiveness actions appropriately to direct the agent back to the low aggressiveness states and away from the higher aggressiveness states.

Notice around update 2000 that the agent demonstrates lower performance for a short period of time. This is because the agent took consecutive random actions towards higher aggressiveness states. The agent did not perform better in these states and the agent returned to the optimal low aggressiveness degree.

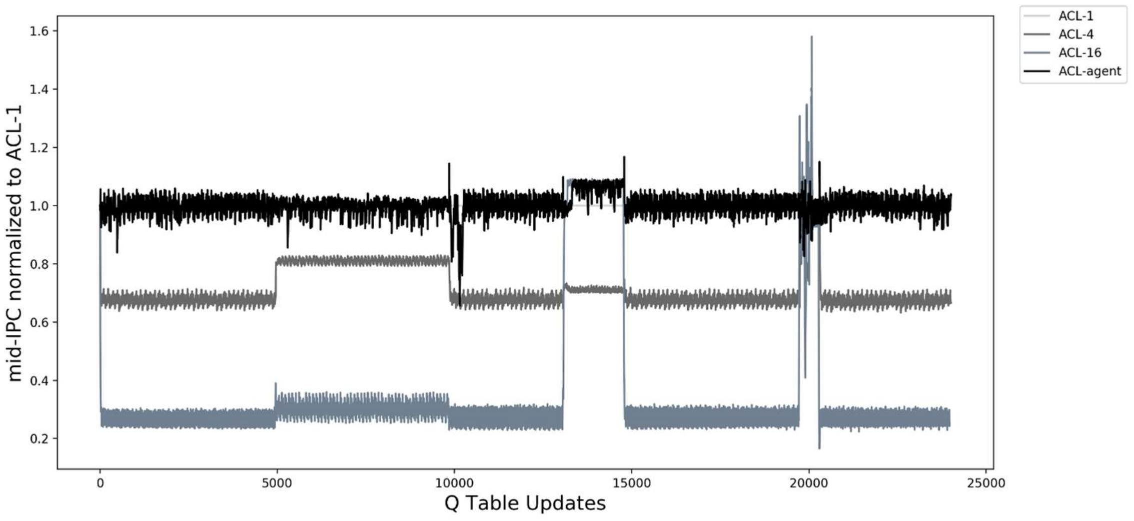


Figure 10: Performance of ADM agent with all optimization techniques over time.

These optimization techniques had a profound impact on the performance of the ADM agent. Figure 10 shows the final performance graph of the ADM agent for the execution of the full region of interest in 507.cactuBSSN_r.

Notice that the ADM agent converges on the optimal aggressiveness degree for the region of interest. At Q-table update 5000, 10000, 13000, 15000, and 20000 the program has discontinuities. This is due to a change in program behavior. At these points, the ADM agent adapted to the new environment, learning new reward values and proving the degree it was already in was optimal. The optimal static prefetcher configuration was degree 1 and the ADM agent reached 99.9% of the optimal static performance.

8.6.4 500.perlbench_r: All optimization Techniques

The 500.perlbench_r workload from the SPEC suite exhibits significant performance drop every 1 million instructions and the performance between different aggressiveness degrees is minimal. Performance changes are issues for the ADM agent because it triggers the Random-Action Annealing and Top Down Convergence to reset the table and enter an exploration phase. We show that our Q-learning optimization techniques convergence towards the highest performing configuration despite these challenges.

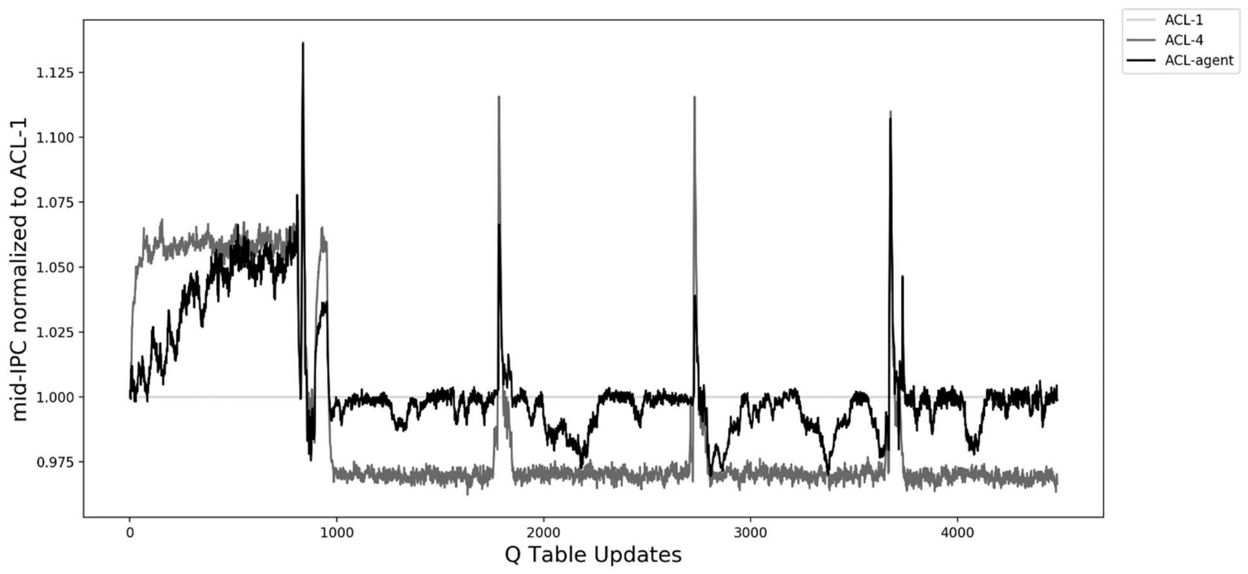


Figure 11: 500.perlbench_r with all optimization techniques.

The results in Figure 11 show the ADM agent converged on the optimal configuration between updates 0 to 1000. During this phase, the optimal configuration was static degree 4. The agent took around 500 updates to converge on this configuration. The reason why it took longer to converge was because the performance variance was similar at degrees one, two, three, and four. This made it difficult for the agent to decide if changing aggressiveness was beneficial.

After the first major performance change, the optimal degree changed from degree four to degree one. The agent also recognized the performance spike around the 1000th update, thus the agent reinitialized the Q-table using the Top Down Convergence method with Random-Action Annealing exploration enabled. We can see that the agent quickly converged on the optimal solution between updates 1000 and 1900.

After the second major performance change, the agent randomly explored higher aggressiveness states. The performance decreased from updates 1900 to 2200 which the agent learned. Then at update 2200 a Quick Convergence Update occurred and the agent rapidly converged on the higher performing lower aggressiveness state. The reason why the agent needed the Quick Convergence Update to converge on the optimal configuration was because the performance in each configuration are too similar. The Set on Increase or Decrease technique has a 1% performance improvement requirement in order to change states (as described in Section 6.4.4). In this case, the only way for the agent to converge optimally was via a Quick Convergence Update.

The third major performance change occurred around update 2850. The agent reset its table values and enabled exploration. The agent learned by update 3000 that aggressiveness degree four is optimal. At update 3300 the agent has a sequence of performance calculations indicating the low aggressiveness states were performing worse than they should have. The agent learned these worse performing values and selected actions directing the agent away from the optimal low aggressiveness configuration. Since 500.perlbench_r has high performance variance relative to the average performance of the other static configurations, it difficult for the agent to learn optimal configurations. Between update 3300 and 3450 the agent selected random actions and learned performance values that directed the agent away from low aggressiveness. Then the performance returned to normal and the agent relearned which aggressiveness state was optimal and returned to the low aggressiveness states. A similar problem occurred around update 4200 where a sequence of low performance causes the agent to think the current state was performing worse than the surrounding states. In either case, the agent converged to the optimal configuration after about 75 updates. 500.perlbench_r is a good example of how the ADM agent can converge on an optimal solution despite challenges of low performance variance and major performance discontinuities.

8.6.5 523.xalancbmk: Regions with Different Optimal Configurations

The ADM agent significantly outperformed all static aggressiveness configurations for 523.xalancbmk. This is because the workload has multiple regions of interest that have different optimal configurations. Figure 12 shows the performance of the workload over time.

The three graphs in Figure 12 show the IPC performance of the various static configurations and the ADM agent. The middle graph shows how the ADM agent adjusted the ACL prefetcher aggressiveness over time and the bottom graph shows how the ADM agent adjusted the DPL prefetcher aggressiveness over time. This workload has two distinct regions separated at update 22500. Between updates 0 to 22500 the workload performed better with a high ACL aggressiveness. The agent learned this configuration and maintained a higher aggressiveness throughout the region. Then after update 22500, the optimal ACL aggressiveness changed to a lower aggressiveness. Again, the agent adapted to this change and maintained a low ACL aggressiveness. The DPL prefetcher did not have a significant impact on performance for this workload. Traces for other workloads are shown in Appendix D.

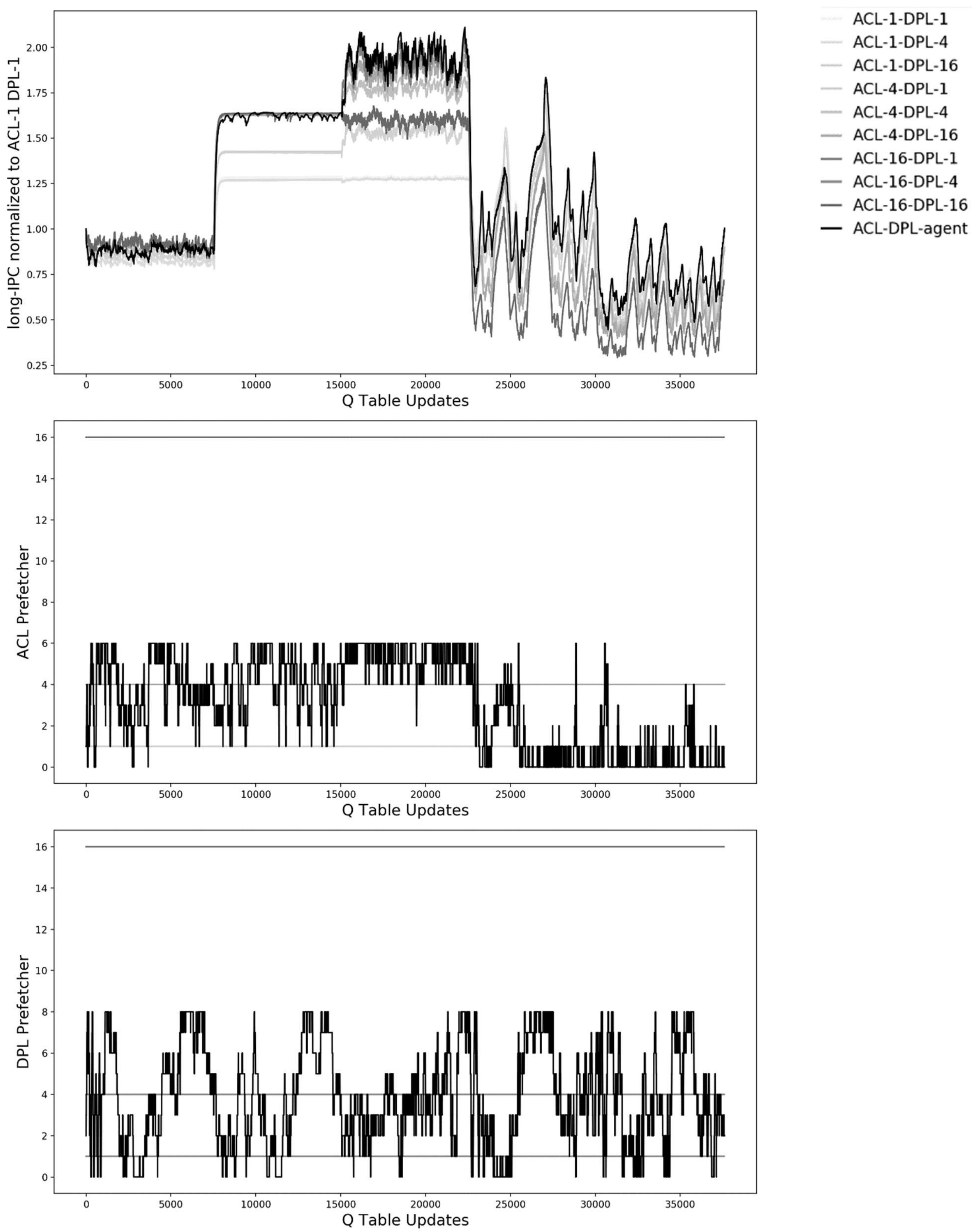


Figure 12: Performance of 523.xalankbmk_r. The top graph shows the performance, the middle graph shows the APL prefetcher configuration, and the bottom graph shows the DPL prefetcher configuration.

8.7 Key Findings

The key findings for this report are separated into two categories, general findings about Q-learning and prefetchers, and specific findings from detailed experiments. The first general finding for this work is that the ADM agent can successfully converge on an optimal hardware configuration to gain performance improvements if they exist. We showed this using the prefetcher aggressiveness degree parameter via experiments in single prefetcher and dual prefetcher implementations. The results of these experiments demonstrated a 6.5% performance increase over the optimal static configuration for the ADM agent only applied to the ACL prefetcher, a 0.5% performance increase over the optimal static configuration for the ADM agent only applied to the DPL prefetcher, and a 4.2% performance increase over the optimal static configuration for the ADM agent applied to both the DPL and APL prefetcher.

A second general finding is that between-workload variance is necessary to see performance improvements. This was realized in the single prefetcher experiments where the ADM agent applied to the ACL prefetcher showed higher performance improvements than the DPL prefetcher. This was a direct result of the between workload variance of the ACL prefetcher being significantly higher than the between workload performance of the DPL prefetcher.

These two general findings provoke future research into other hardware performance parameters. If between-workload performance variance exists for a hardware parameter, then a Q-learning manager will be able to find the optimal hardware configuration.

Each optimization technique accounts for a specific finding. First, Top Down Convergence ensures that the agent explores each state at least once. This is a useful algorithm compared to the normal initialization method because instead of “pulling up” each state, the states are all “pulled down” giving all states the opportunity to demonstrate their performance. Second, Random Action Annealing helps an agent explore its environment and leave configurations leading to shoulder or plateau performances. This is a significant improvement on general hill climbing which cannot explore outside of shoulders or plateaus easily. Next, the Set on Increase or Decrease technique reduces the search space of the agent by two-thirds since the agent no longer learns the increase or decrease actions. This reduction in search space dramatically decreases the time to converge on an optimal configuration, making this a significant optimization technique. A fourth specific finding is the Quick Convergence Update, which is the driving force behind stopping oscillations due to Top Down Convergence and Set on Increase or Decrease because it caused the agent to realize when one set of actions is worse than other actions. This technique allows for exploring, but then after a Quick Convergence Update, the agent rapidly converges on the current optimal configuration. Finally, the Stay Six technique is powerful because it forces the agent to learn the “real reward” at all states it encounters before proceeding to other states.

9 Conclusion

In this work, we propose a method for managing a prefetchers aggressiveness titled ADM. An ADM agent quickly converges to the optimal prefetcher configuration using minimal hardware overhead and latency. We employ Q-learning to find the optimal prefetcher aggressiveness policy for multiple prefetchers at run-time. We wrap the foundational Q-learning algorithm to improve convergence using seven optimization techniques: alternating rewards, random action annealing, top down convergence, set on increase or decrease, quick convergence update, and stay six.

We evaluated the ADM agent using 10 prefetch sensitive workloads from the SPEC CPU2017 suite. The ADM agent demonstrated a 4.2% higher speedup than the best static hardware configuration, with a 2.6% average variance from the optimal prefetcher aggressiveness degree.

We extended the functionality of an ADM agent to multiple prefetchers by proposing the idea of alternating rewards using a weighted moving average of the three most recent updates. This method helps solve the "temporal reward" issue with which agent caused the reward. It also supports synergism between the two agents, encoding the positive or negative interactions between the two agents. We learned through experimentation that a dynamic prefetcher optimization agent is better than a static configuration. This agent is a reasonable solution to optimizing the prefetcher aggressiveness degree because it only requires a few states and actions, thus requiring minimal hardware. Future work for this project is to develop a hardware implementation.

10 References

[1] Intel 2016. Intel 64 and IA-32 Architectures Developer's Manual: Volume 3C. Intel

[2] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. 2009. Machine Learning-based Prefetch Optimization for Data Center Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (SC '09). 1-10.

[3] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. 2015. Maximizing Hardware Prefetch Effectiveness with Machine Learning. In *Proceedings of the 17th International Conference on High Performance Computing and Communications*. 383-389

[4] Vish Viswanathan. 2014. Disclosure of H/W Prefetcher Control on some Intel Processors. Technical Report. Intel.

[5] Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *ISCA* 1990.

[6] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. *Hot Topics in Operating Systems*, 2005.

[7] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N Patt. 2007. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. *In Proceedings of the International Symposium on High Performance Computer Architecture* (HPCA). 63–74.\

[8] Seth H. Pugsley, Zeshan Chishti, Chris Wilkerson, Peng-fei Chuang, Robert L. Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. 2014. Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers. *In Proceedings of the International Symposium on High Performance Computer Architecture* (HPCA). 626–637.

[9] J. Hiebel et al. Machine learning for fine-grained hardware prefetcher control. Proc. ICPP, p. 3, 2019.

[10] Saami Rahman et al. "Maximizing hardware prefetch effectiveness with machine learning". In: High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on. IEEE. 2015, pp. 383–389.

[11] Bellman, Richard. The theory of dynamic programming. *Bull. Amer. Math. Soc. 60* (1954), no. 6, 503--515. https://projecteuclid.org/euclid.bams/1183519147

[12] William Hasenplaugh, Pritpal S. Ahuja, Aamer Jaleel, Simon Steely Jr., and Joel Emer. The gradient-based cache partitioning algorithm. *ACM Trans. Archit. Code Optim*. 8, 4, Article 44 (January 2012), 21 pages. DOI:https://doi.org/10.1145/2086696.2086723

[13] A. Aziz, M. Cireno, E. Barros and B. Prado, "Balanced Prefetching Aggressiveness Controller for NoC-based Multiprocessor", SBCCI '14 Proceedings of the 27th Symposium on Integrated Circuits and Systems Design, 2014.

[14] Ankur Limaye and Tosiron Adegbija. A workload characterization of the spec cpu2017 benchmark suite. *In Performance Analysis of Systems and Software* (ISPASS), 2018 IEEE International Symposium on, pages 149–158. IEEE, 2018.

[15] Biswabandan Panda. 2016. SPAC: A Synergistic Prefetcher Aggressiveness Controller for Multi-Core Systems. IEEE Trans. Comput. 65, 12 (Dec 2016), 3740–3753

[16] Biswabandan Panda and Shankar Balachandran. 2015. CAFFEINE: A utility-driven prefetcher aggressiveness engine for multicores. ACM Trans. Archit. Code Optim. 12, 3, Article 30 (August 2015), 25 pages. DOI: http://dx.doi.org/10.1145/2806891

[17] Shiori Sagawa et al. “Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization”. In: International Conference on Learning Representations. 2020.

[18] Amir-massoud Farahmand. Regularization in Reinforcement Learning. PhD thesis, University of Alberta, 2011b.

[19] The Sniper Multi-Core Simulator. (2020, July 21). Sniper. Retrieved July 24, 2020, from https://snipersim.org/w/The_Sniper_Multi-Core_Simulator

[20] Ice Lake (server) - Microarchitectures - Intel. (2020, September 9). WikiChip. Retrieved September 28, 2020, from https://en.wikichip.org/wiki/intel/microarchitectures/ice_lake_(server)

[21] SPEC CPU 2017. (2019, September 9). Standard Performance Evaluation Corporation. Retrieved August 14, 2020, from https://www.spec.org/cpu2017/

[22] A. Navarro-Torres, J. Alastruey-Benede, P. Ibanez-Marin, and V. Vinals-Yufera, “Memory hierarchy characterization of spec cpu2006 and spec cpu2017 on the intel xeon skylake-sp,” PLOS ONE, vol. 14, no. 8, pp. 1–24, August 2019.

[23] W. Heirman, K. D. Bois, Y. Vandriessche, S. Eyerman, and I. Hur, “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, ser. PACT ’18. New York, NY, USA: ACM, 2018, pp. 28:1–28:11. [Online]. Available: http://doi.acm.org/10.1145/3243176.3243181

[24] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. IEEE Transactions on Parallel and Distributed Systems, 1995.

[25] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. 2009. Coordinated Control of Multiple Prefetchers in Multi-Core Systems. In Proceedings of the International Symposium on Microarchitecture (MICRO). 316–326.

[26] W. Hasenplaugh et al., "The gradient-based cache partitioning algorithm", ACM Trans. on Arch. and Code Opt., vol. 8, no. 4, 2012.

[27] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8(3-4):279–292, 1992.

[28] Richard Bellman, Dynamic programming, Princeton University Press, Princeton, N. J., 1957. MR 0090477

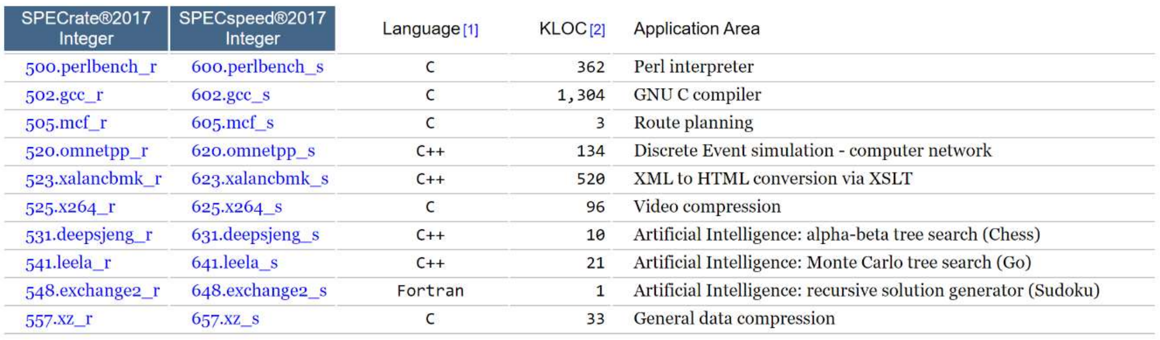
Appendix A: Acronym List

Acronym	Name
ADM	Aggressiveness Degree Manager
DPL	Data Prefetch Logic Prefetcher
ACL	Adjacent Cache Line Prefetcher
AR	Alternating Rewards
RAA	Random Action Annealing
TDC	Top Down Convergence
SID	Set on Increase of Decrease
QCU	Quick Convergence Update
SS	Stay-Six
QoS	Quality of Service
IPC	Instructions Per Cycle
MIPS	Millions of Instructions Per Second
Q-Learning	Quality Learning
HC	Hill Climbing
MLC	Mid-Level Cache
LLC	Last-Level Cache
PLRU	Pseudo Least Recently Used
FDP	Feedback Directed Prefetching
NST	Near-Side Throttling
FS	Fair Speedup
SPEC	Standard Performance Evaluation Corporation

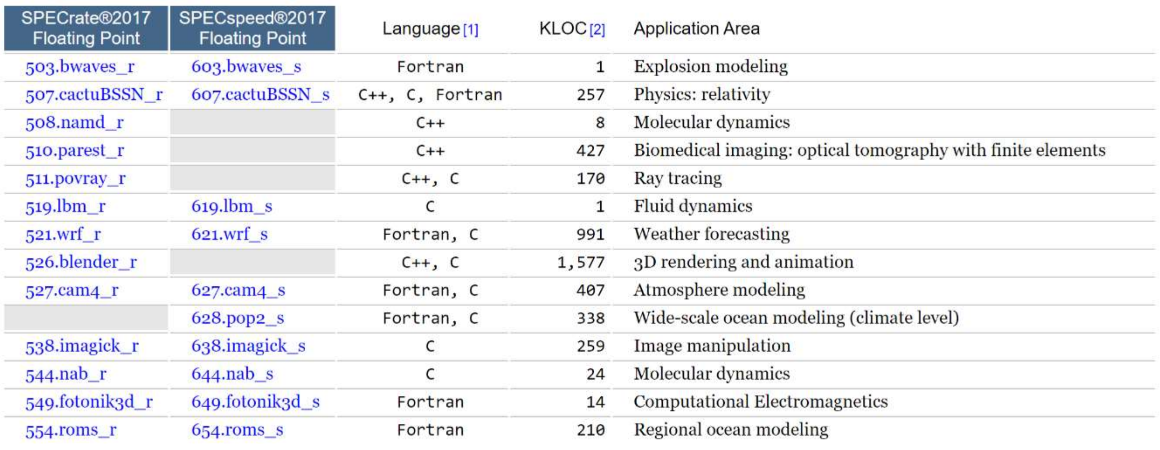
Appendix B: Sampling Regions for SPEC CPU2017

- 500-perlbench
    - 379 billion instructions executed
    - 5 regions of interest
    - 5.6 billion instructions executed in detailed mode
- 502-gcc
    - 235 billion instructions executed
    - 5 regions of interest
    - 3.5 billion instructions executed in detailed mode
- 505-mcf
    - 148 billion instructions executed
    - 5 regions of interest
    - 2.2 billion instructions executed in detailed mode
- 507-cactuBSSN(220b?5?3300m)
    - 220 billion instructions executed
    - 5 regions of interest
    - 3.3 billion instructions executed in detailed mode
- 510-parest
    - 660 billion instructions executed
    - 5 regions of interest
    - 7.9 billion instructions executed in detailed mode
- 519-lbm
    - 179 billion instructions executed
    - 5 regions of interest
    - 2.7 billion instructions executed in detailed mode
- 520-omnetpp
    - 123 billion instructions executed
    - 1 regions of interest
    - 1.8 billion instructions executed in detailed mode
- 523-xalancbmk
    - 500 billion instructions executed
    - 5 regions of interest
    - 5.0 billion instructions executed in detailed mode
- 549-fotonik3d
    - 200 billion instructions executed
    - 2 regions of interest
    - 2.0 billion instructions executed in detailed mode
- 554-roms
    - 289 billion instructions executed
    - 5 regions of interest
    - 2.9 billion instructions executed in detailed mode

Appendix C: Descriptions of SPEC CPU2017 Benchmarks



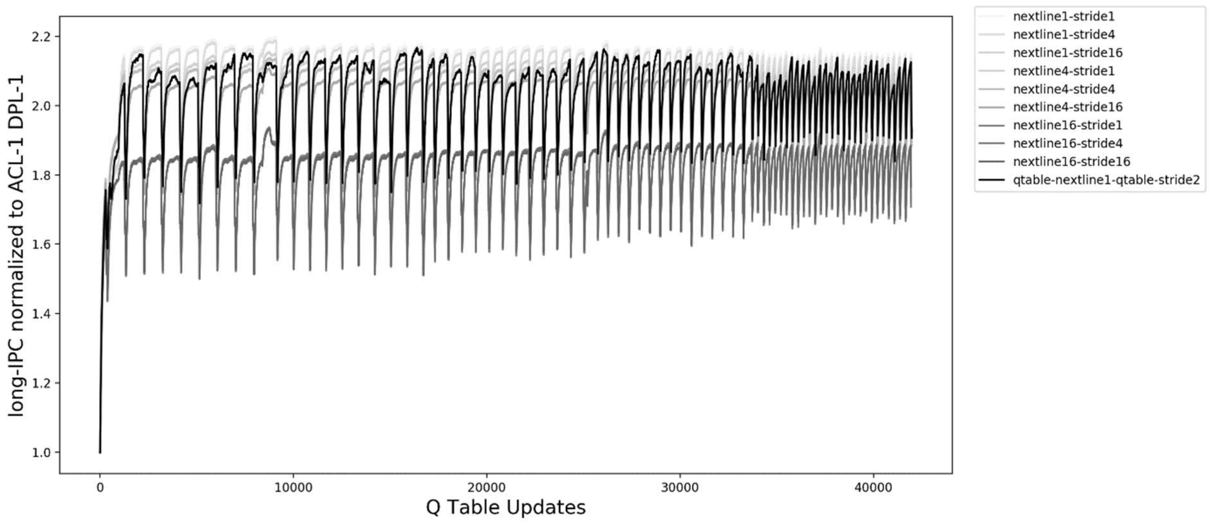
SPECrate®2017 Integer	SPECspeed®2017 Integer	Language[1]	KLOC[2]	Application Area
500.perlbench_r	600.perlbench_s	C	362	Perl interpreter
502.gcc_r	602.gcc_s	C	1,304	GNU C compiler
505.mcf_r	605.mcf_s	C	3	Route planning
520.omnetpp_r	620.omnetpp_s	C++	134	Discrete Event simulation - computer network
523.xalancbmk_r	623.xalancbmk_s	C++	520	XML to HTML conversion via XSLT
525.x264_r	625.x264_s	C	96	Video compression
531.deepsjeng_r	631.deepsjeng_s	C++	10	Artificial Intelligence: alpha-beta tree search (Chess)
541.leela_r	641.leela_s	C++	21	Artificial Intelligence: Monte Carlo tree search (Go)
548.exchange2_r	648.exchange2_s	Fortran	1	Artificial Intelligence: recursive solution generator (Sudoku)
557.xz_r	657.xz_s	C	33	General data compression

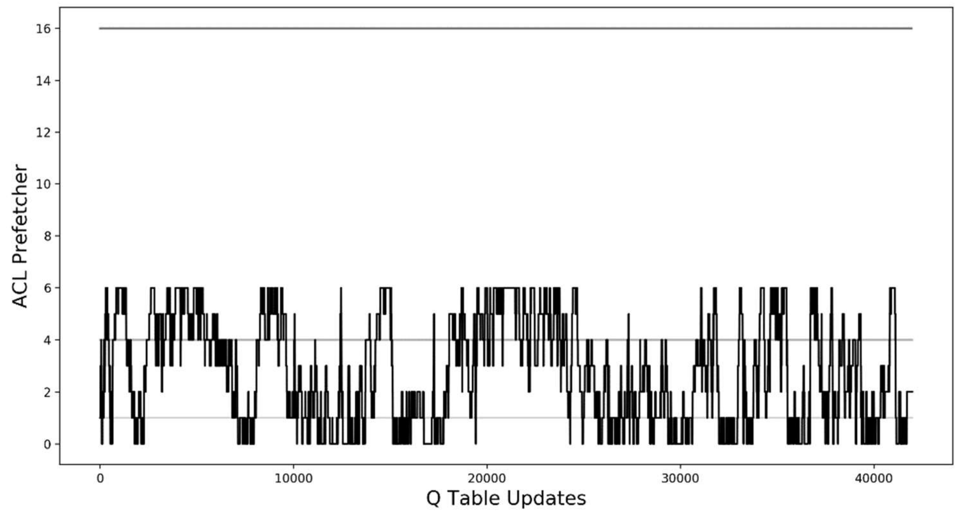


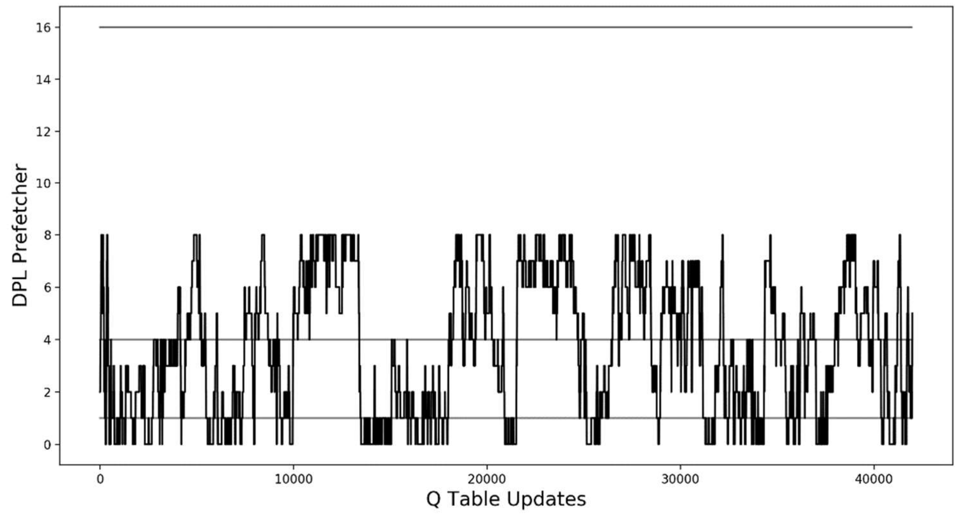
SPECrate®2017 Floating Point	SPECspeed®2017 Floating Point	Language[1]	KLOC[2]	Application Area
503.bwaves_r	603.bwaves_s	Fortran	1	Explosion modeling
507.cactuBSSN_r	607.cactuBSSN_s	C++, C, Fortran	257	Physics: relativity
508.namd_r		C++	8	Molecular dynamics
510.parest_r		C++	427	Biomedical imaging: optical tomography with finite elements
511.povray_r		C++, C	170	Ray tracing
519.lbm_r	619.lbm_s	C	1	Fluid dynamics
521.wrf_r	621.wrf_s	Fortran, C	991	Weather forecasting
526.blender_r		C++, C	1,577	3D rendering and animation
527.cam4_r	627.cam4_s	Fortran, C	407	Atmosphere modeling
	628.pop2_s	Fortran, C	338	Wide-scale ocean modeling (climate level)
538.imagick_r	638.imagick_s	C	259	Image manipulation
544.nab_r	644.nab_s	C	24	Molecular dynamics
549.fotonik3d_r	649.fotonik3d_s	Fortran	14	Computational Electromagnetics
554.roms_r	654.roms_s	Fortran	210	Regional ocean modeling

Appendix D: Traces of Workloads

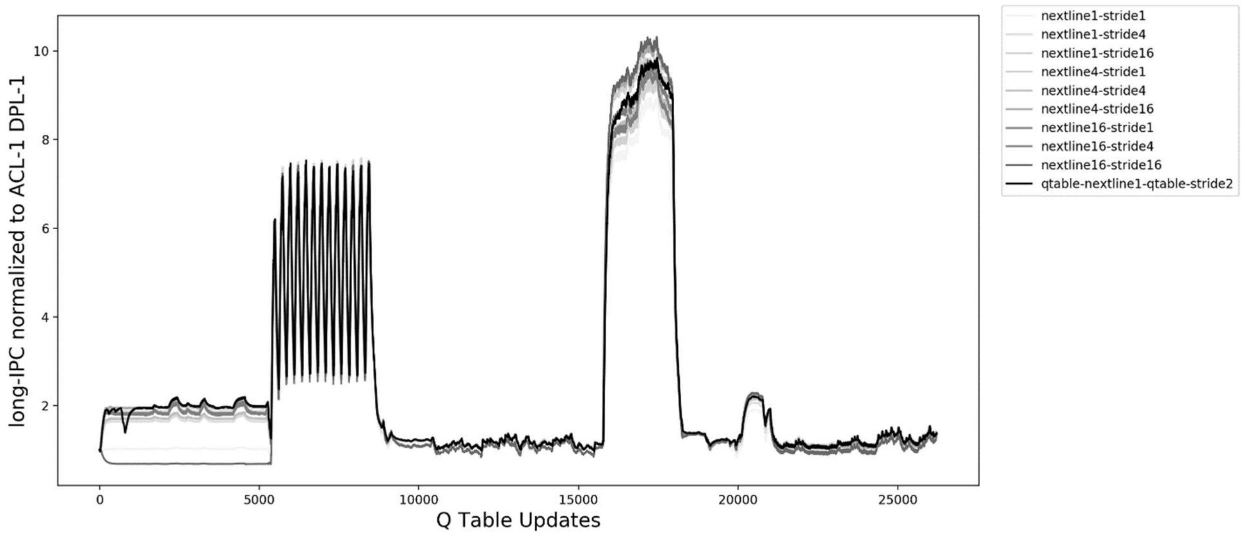
500.perlbench_r

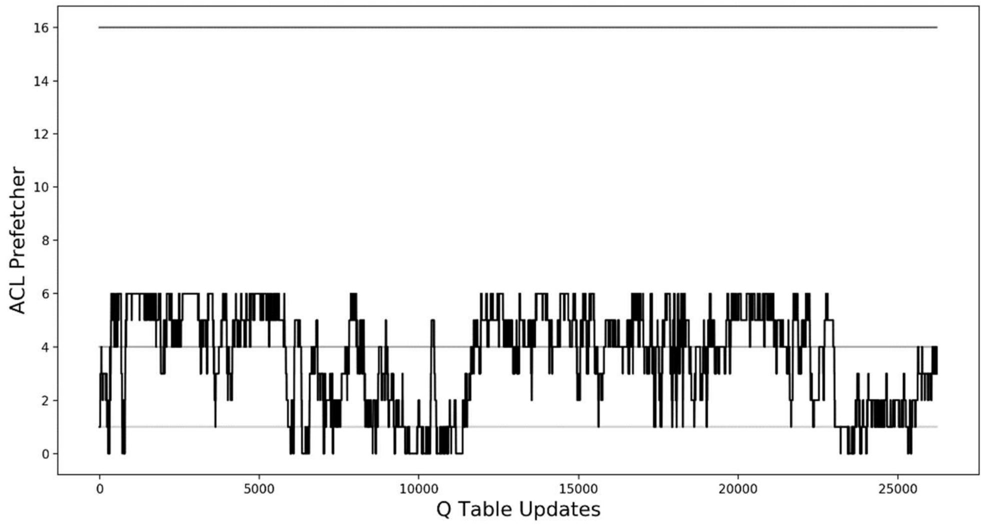


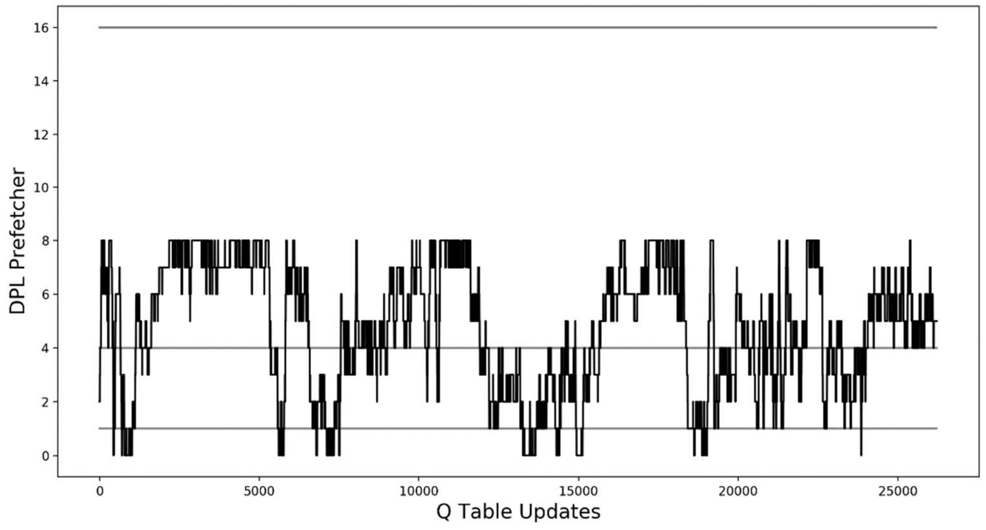




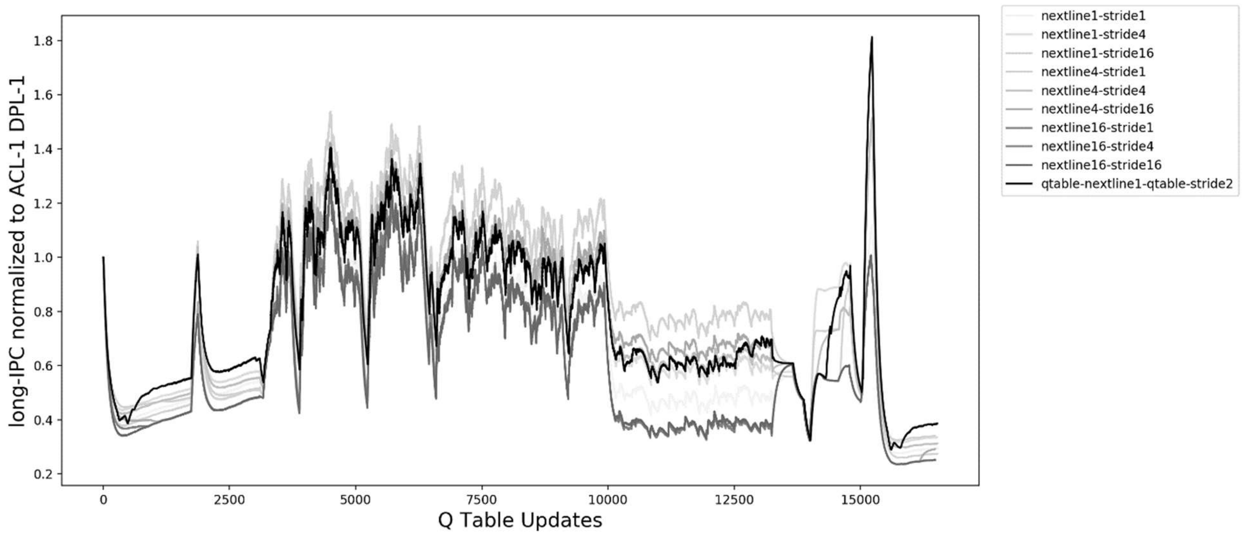
502.gcc_r

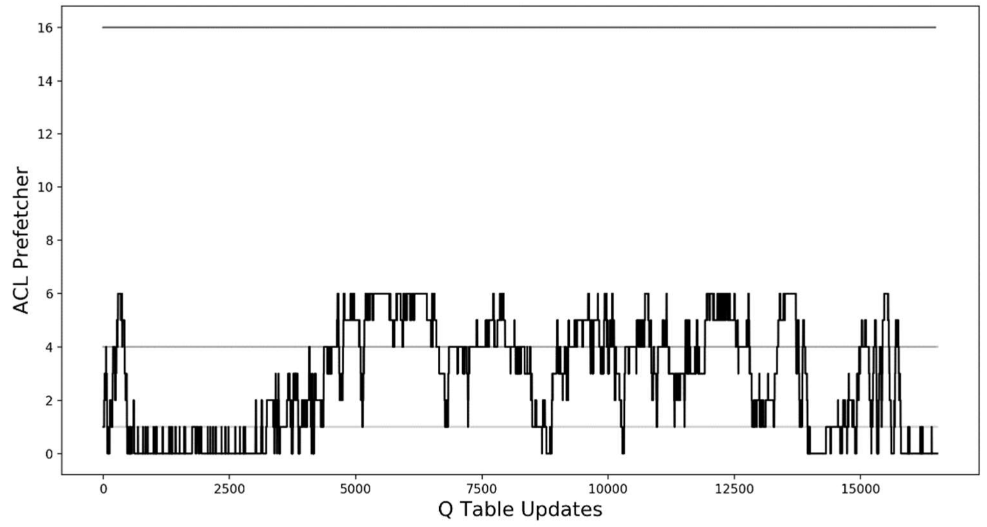

nextline1-stride1
nextline1-stride4
nextline1-stride16
nextline4-stride1
nextline4-stride4
nextline4-stride16
nextline16-stride1
nextline16-stride4
nextline16-stride16
qtable-nextline1-qtable-stride2
long-IPC normalized to ACL-1 DPL-1
Q Table Updates

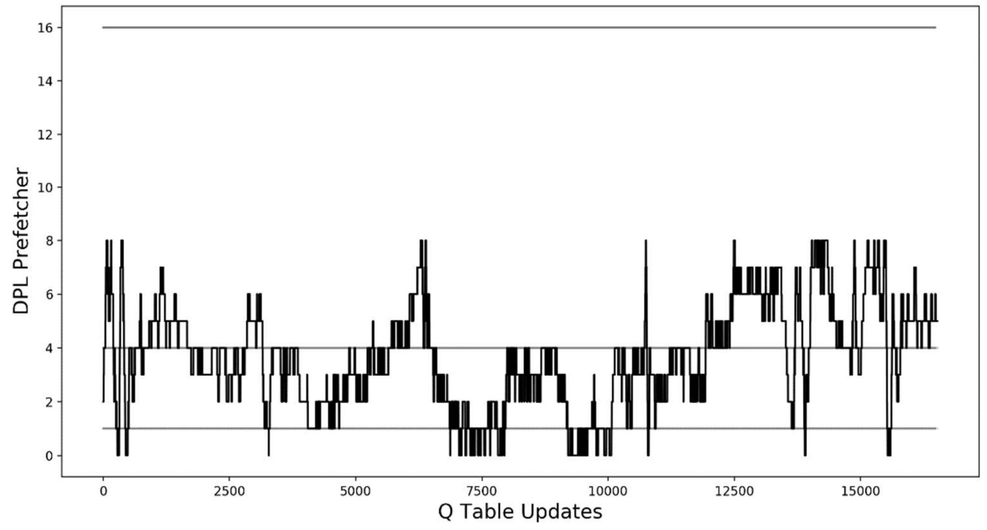

ACL Prefetcher
Q Table Updates


DPL Prefetcher
Q Table Updates

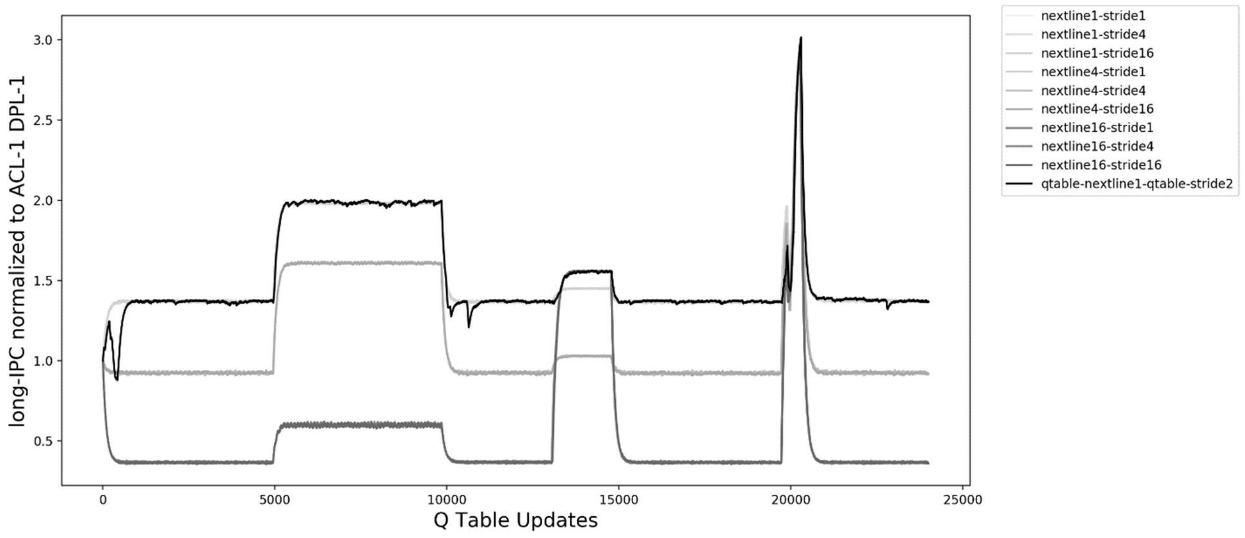
505.mcf_r

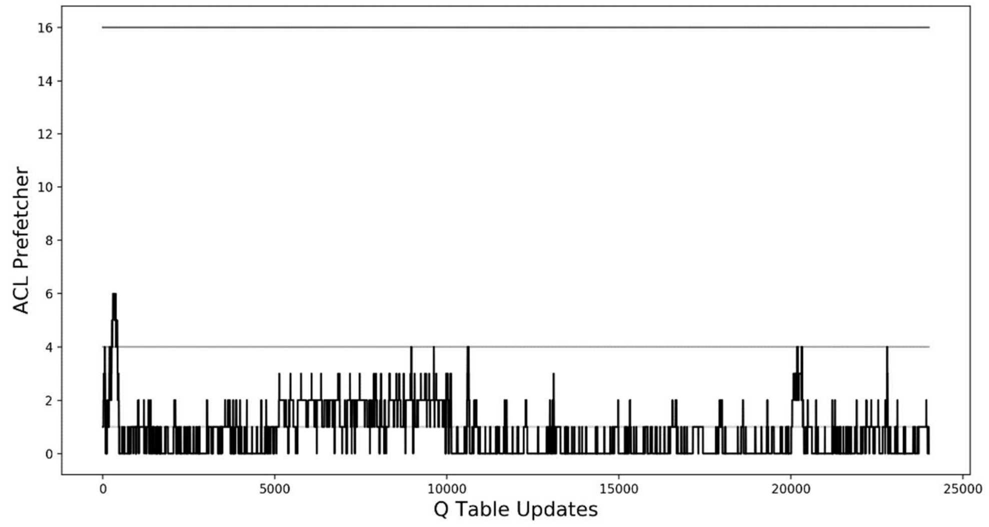

nextline1-stride1
nextline1-stride4
nextline1-stride16
nextline4-stride1
nextline4-stride4
nextline4-stride16
nextline16-stride1
nextline16-stride4
nextline16-stride16
qtable-nextline1-qtable-stride2
long-IPC normalized to ACL-1 DPL-1
Q Table Updates

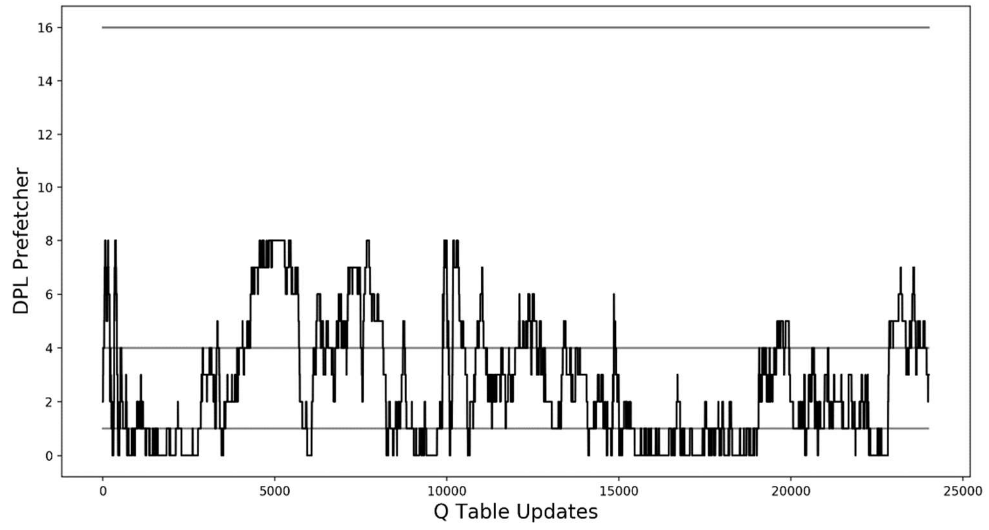

ACL Prefetcher
Q Table Updates


DPL Prefetcher
Q Table Updates

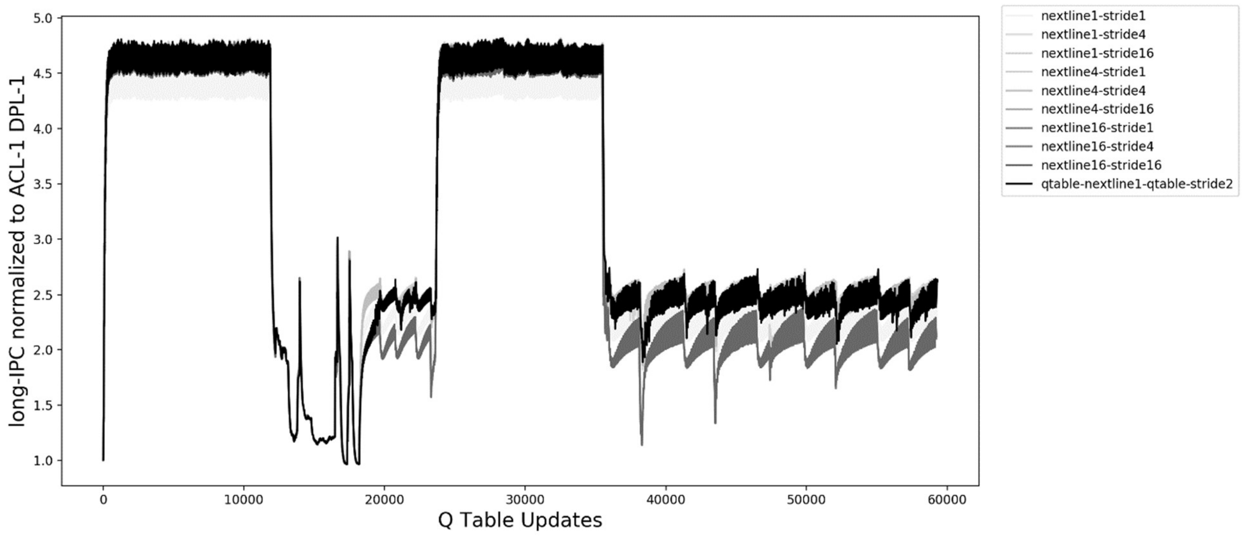
507.cactuBSSN_r

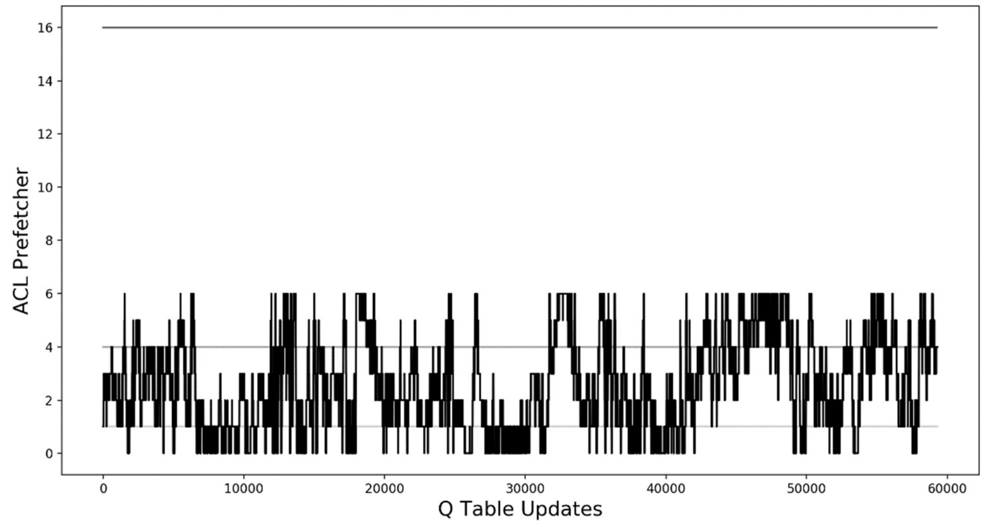


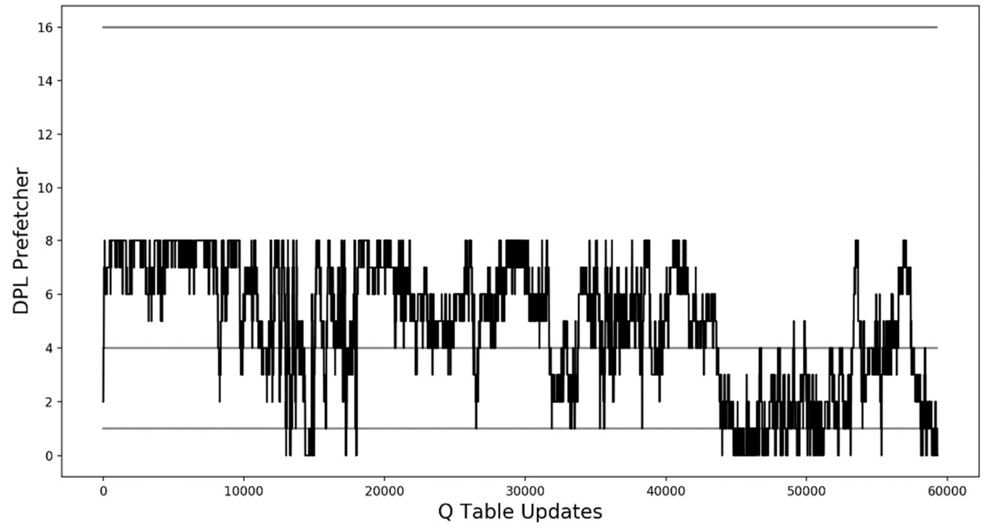




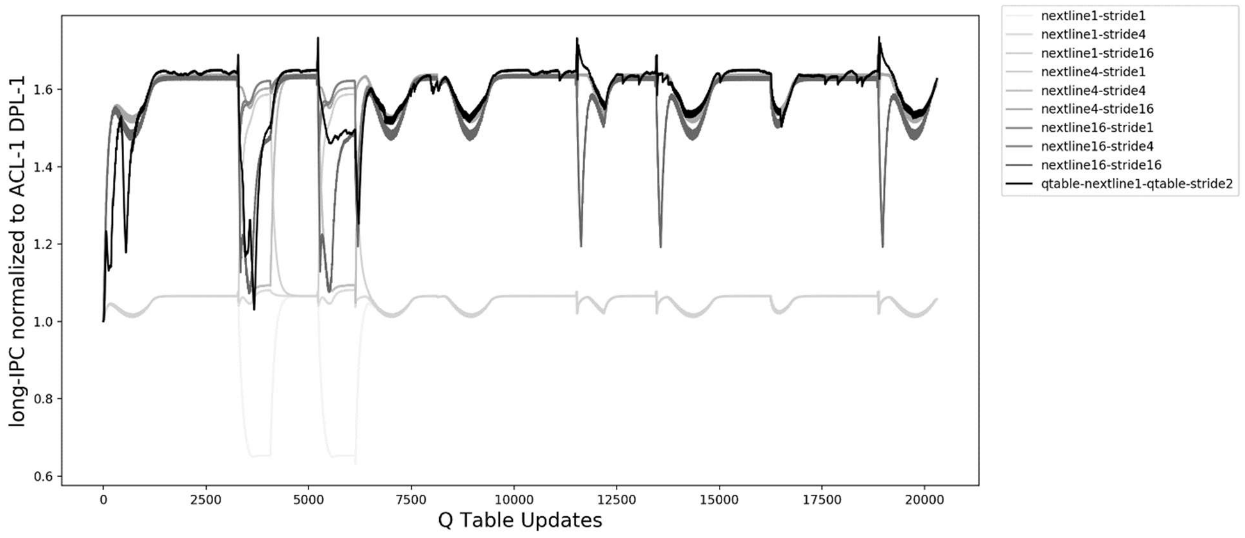
510.parest_r

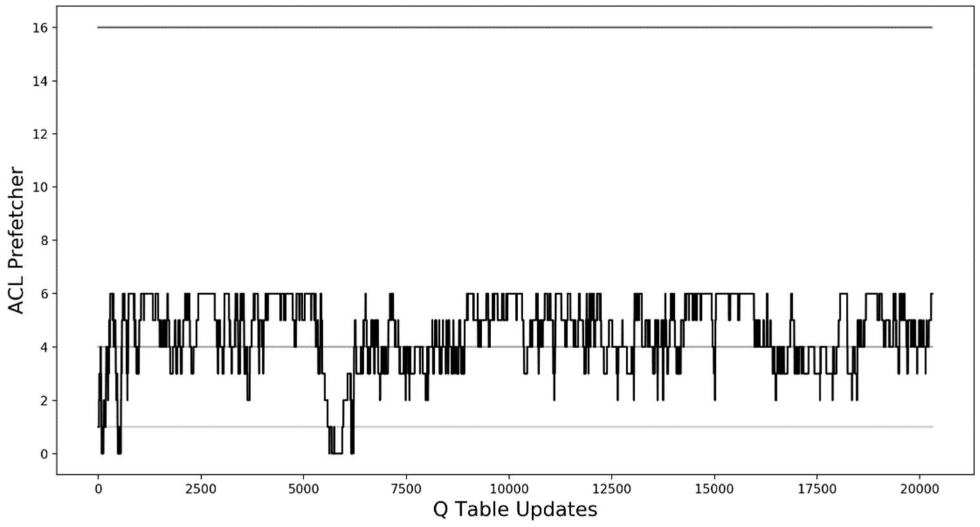

long-IPC normalized to ACL-1 DPL-1
Q Table Updates
nextline1-stride1
nextline1-stride4
nextline1-stride16
nextline4-stride1
nextline4-stride4
nextline4-stride16
nextline16-stride1
nextline16-stride4
nextline16-stride16
qtable-nextline1-qtable-stride2

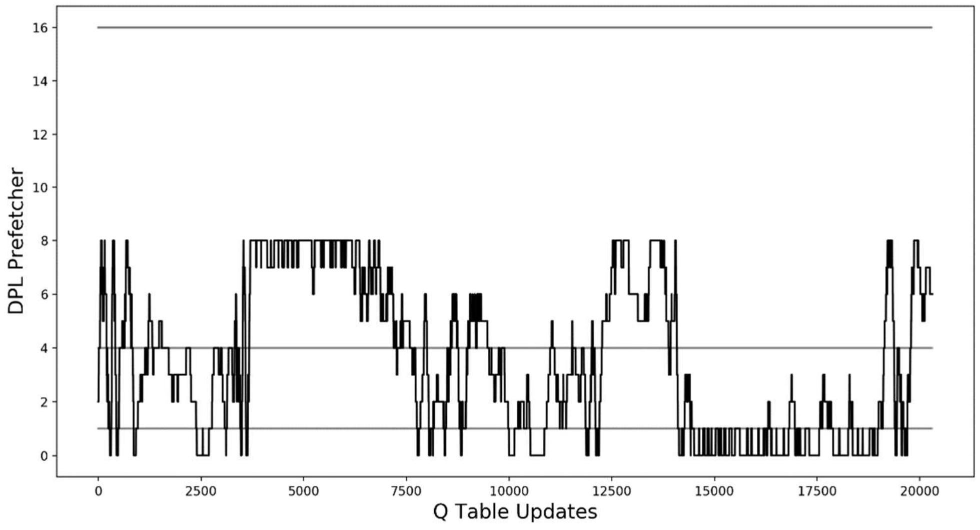

ACL Prefetcher
Q Table Updates


DPL Prefetcher
Q Table Updates

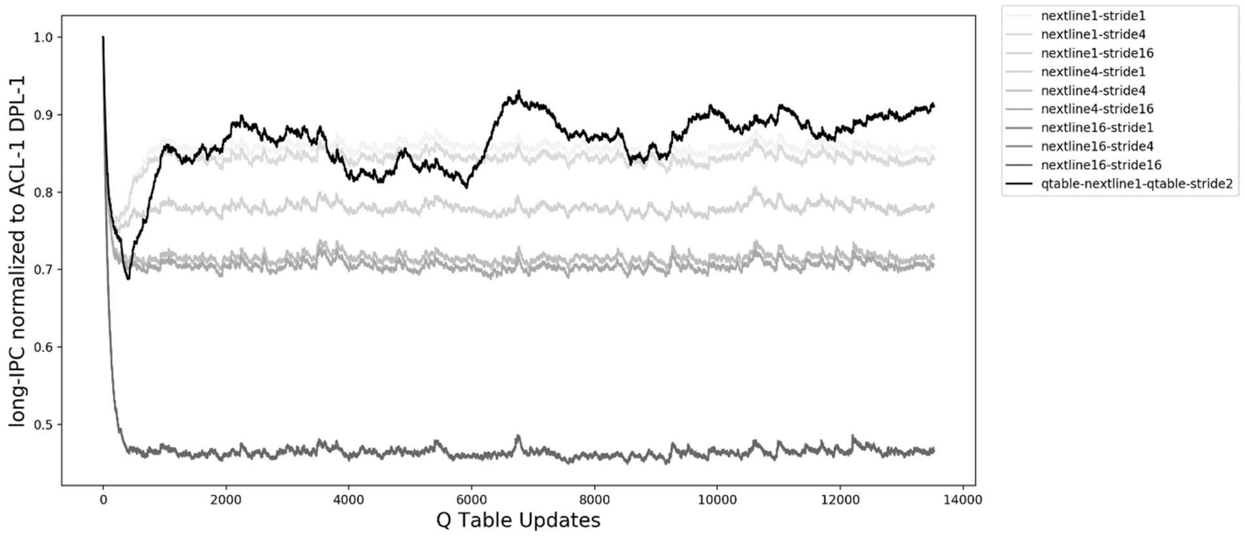
519.lbm_r

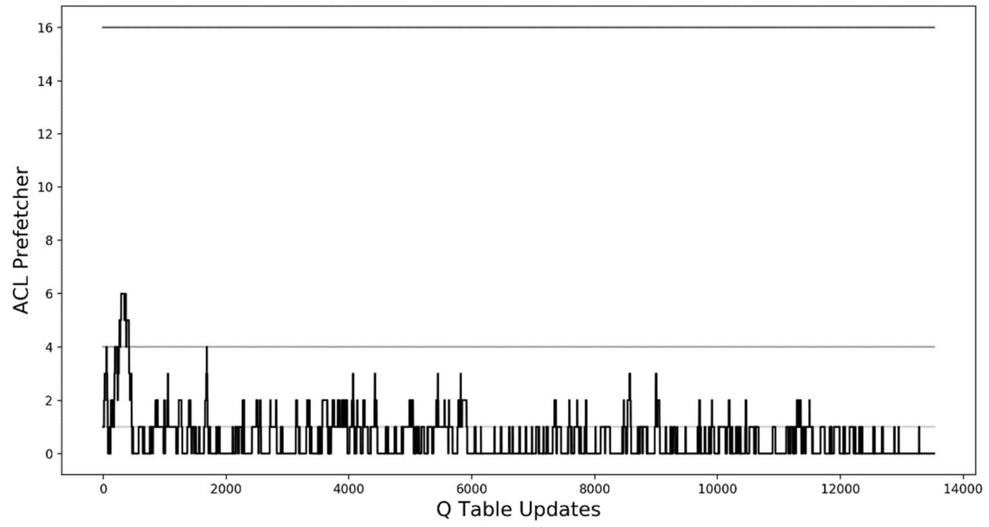

long-IPC normalized to ACL-1 DPL-1
Q Table Updates
nextline1-stride1
nextline1-stride4
nextline1-stride16
nextline4-stride1
nextline4-stride4
nextline4-stride16
nextline16-stride1
nextline16-stride4
nextline16-stride16
qtable-nextline1-qtable-stride2

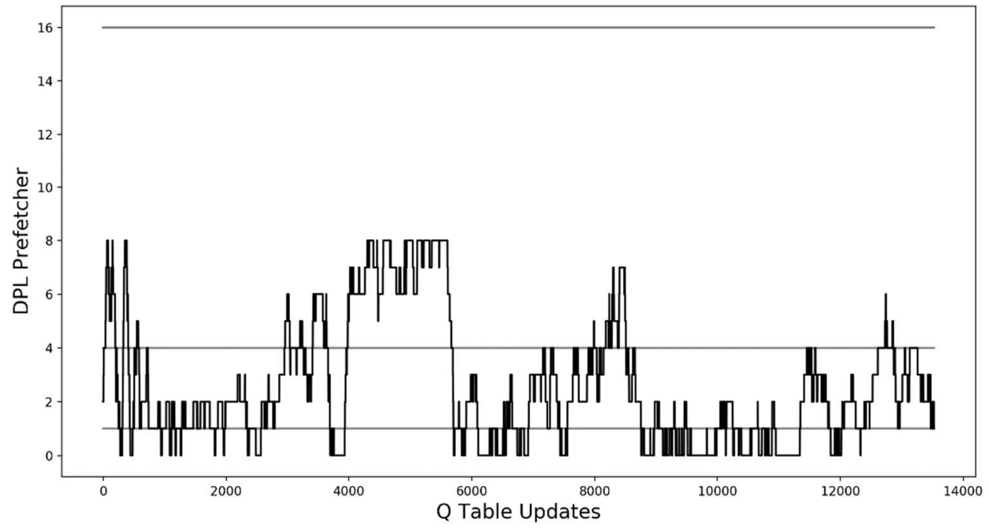

ACL Prefetcher
Q Table Updates


DPL Prefetcher
Q Table Updates

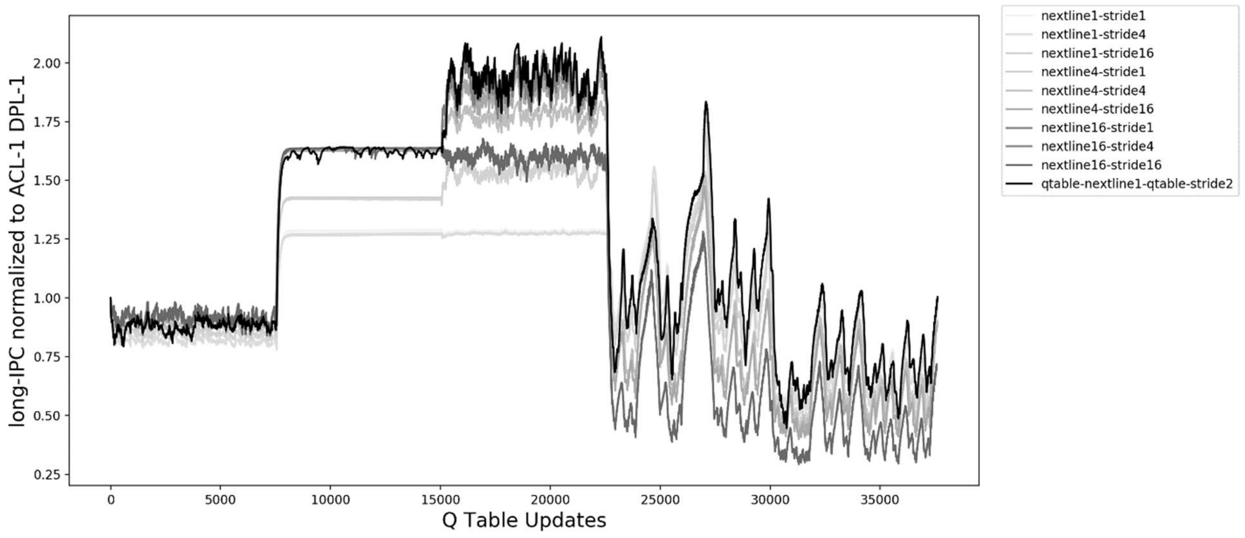
520.omnetpp_r

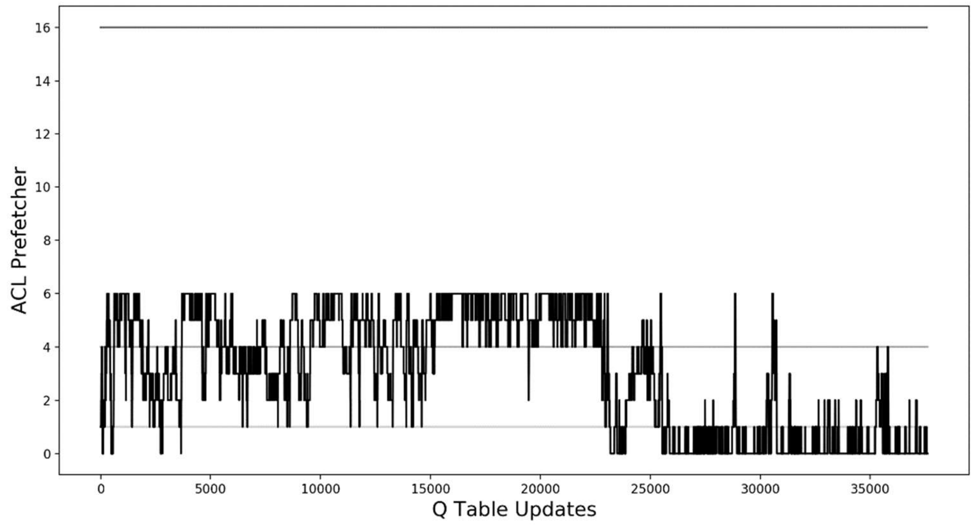


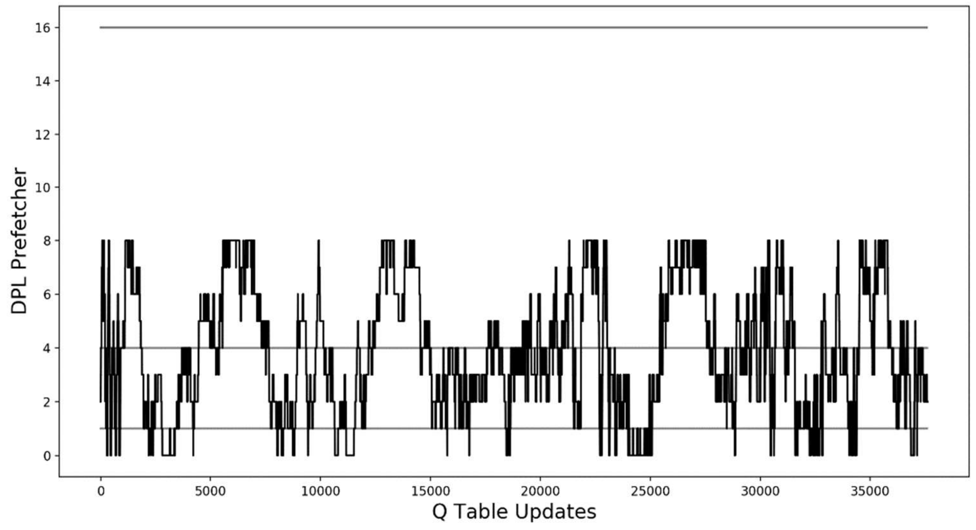




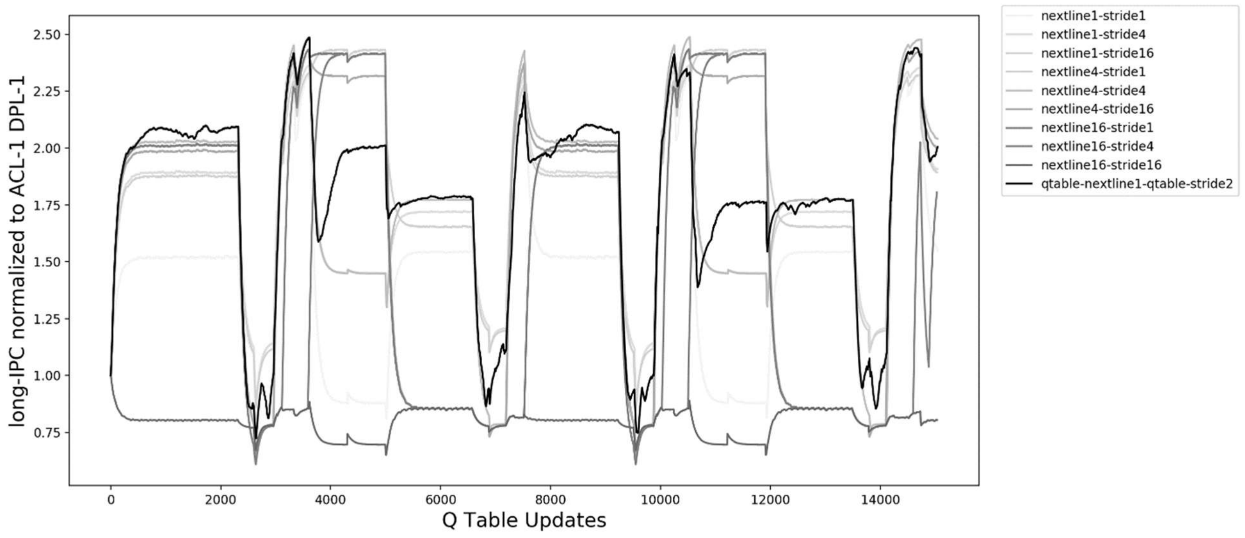
523.xalancbmk_r

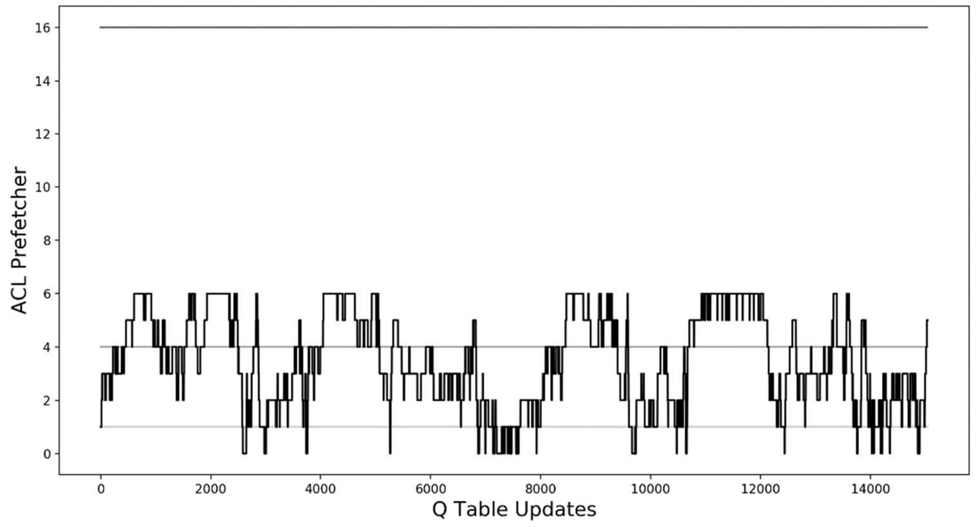


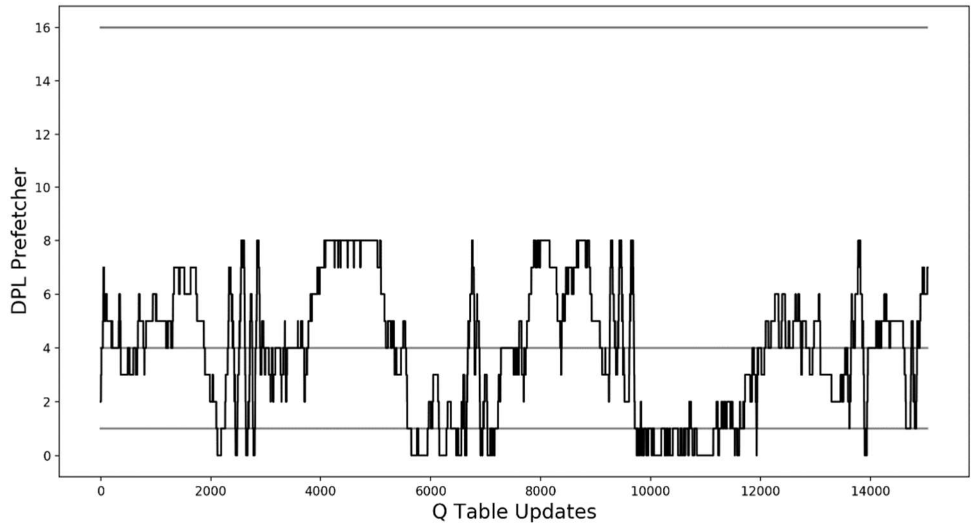




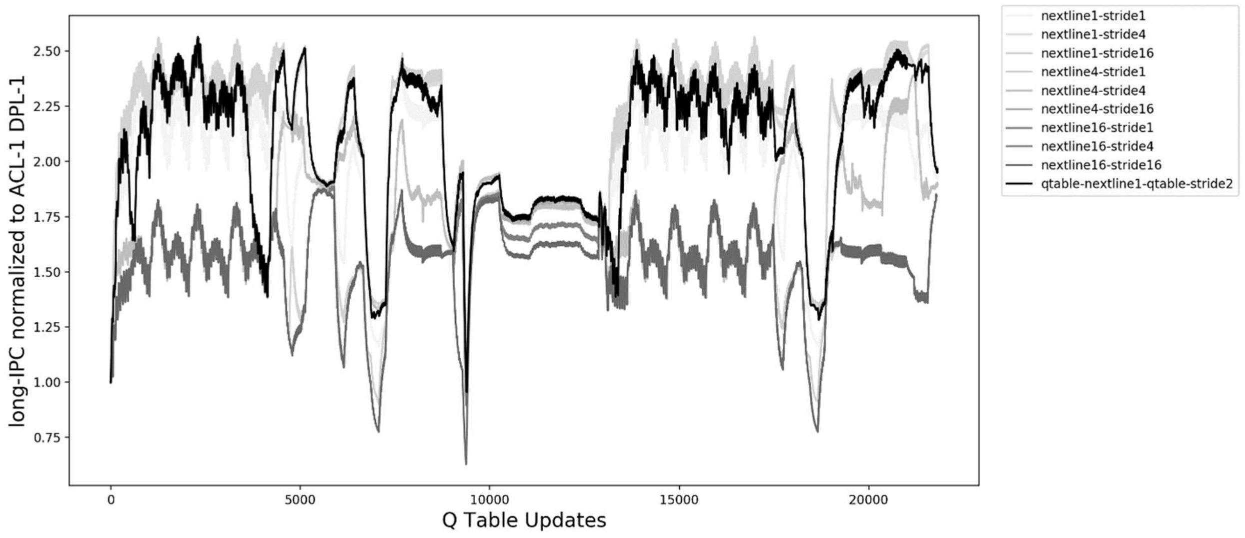
549.fotonik3d_r

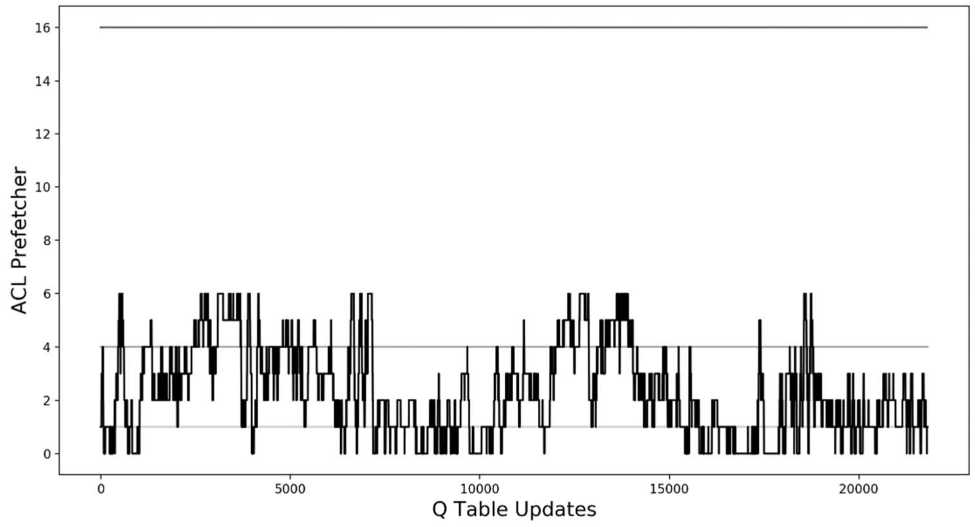


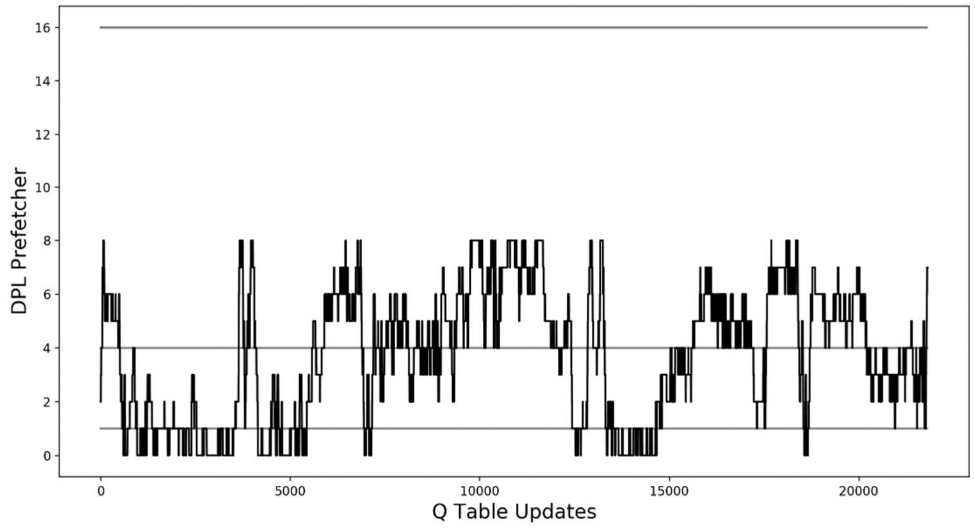




554.roms_r


long-IPC normalized to ACL-1 DPL-1
Q Table Updates
nextline1-stride1
nextline1-stride4
nextline1-stride16
nextline4-stride1
nextline4-stride4
nextline4-stride16
nextline16-stride1
nextline16-stride4
nextline16-stride16
qtable-nextline1-qtable-stride2


ACL Prefetcher
Q Table Updates


DPL Prefetcher
Q Table Updates