2019

Contextual Bandit Modeling for Dynamic Runtime Control in Computer Systems

Jason Hiebel
Michigan Technological University, jshiebel@mtu.edu

Contextual Bandit Modeling for Dynamic Runtime Control in Computer Systems

By

Jason Hiebel

A Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In Computer Science

Michigan Technological University

2019

This dissertation has been approved in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Computer Science.

Department of Computer Science

Dissertation Co-Advisor: *Laura E. Brown*

Dissertation Co-Advisor: *Zhenlin Wang*

Committee Member: *Nilufer Onder*

Committee Member: *Allan A. Struthers*

Department Chair: *Linda Ott*

To my mother,

a woman with a great capacity

for tenacity and perseverance

that I strive everyday to myself achieve

Contents

List of Figures

LIST OF FIGURES

List of Tables

Preface

Chapter 3 contains material previously published in the Proceedings of the 47th International Conference on Parallel Processing (ICPP '18) [61]:

> Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Constructing dynamic policies for paging mode selection. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, pages 72:1–72:9, 2018, doi:10.1145/3225058.3225082.

Chapter 4 contains material previously published in the Proceedings of the 48th International Conference on Parallel Processing (ICPP '19) [62]:

> Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Machine learning for fine-grained hardware prefetcher control. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 3:1–3:9, 2019, doi:10.1145/3337821.3337854.

The material described in Chapter 5 has been submitted for review and publication.

Acknowledgements

I would first like to give my sincere appreciation and thanks to my advisors, Dr. Laura E. Brown and Dr. Zhenlin Wang, for their continued instruction and guidance. Through these past seven years, their counsel and enduring patience helped me learn, grow, and accomplish more than I imagined I could at the start of this journey. I would also like to recognize both Dr. Nilufer Önder and Dr. Allan A. Struthers. Their cheerful investment in my work was a fount of motivation to improve.

As the Turkish proverb goes, "always the trees that bear fruit are stoned." I'm glad that they saw potential in myself and my work and challenged me to succeed and excel.

Finally, I would like to extend my deepest gratitude to my mother, Mary Hiebel, father, Larry Clemo, and step-mother, Shannon Clemo, for their love and support. It was upon this bedrock that I was and continue to be able to build towards my future. From my first exposure to computer science in the shadow of my father, to the parallel struggles of working through graduate education with my mother, my family has been an indispensable source of comfort and inspiration.

Abstract

Modern operating systems and microarchitectures provide a myriad of mechanisms for monitoring and affecting system operation and resource utilization at runtime. Dynamic runtime control of these mechanisms can tailor system operation to the characteristics and behavior of the current workload, resulting in improved performance. However, developing effective models for system control can be challenging. Existing methods often require extensive manual effort, computation time, and domain knowledge to identify relevant low-level performance metrics, relate low-level performance metrics and high-level control decisions to workload performance, and to evaluate the resulting control models.

This dissertation develops a general framework, based on the contextual bandit, for describing and learning effective models for runtime system control. Random profiling is used to characterize the relationship between workload behavior, system configuration, and performance. The framework is evaluated in the context of two applications of progressive complexity; first, the selection of paging modes (Shadow Paging, Hardware-Assisted Page) in the Xen virtual machine memory manager; second, the utilization of hardware memory prefetching for multi-core, multi-tenant workloads with cross-core contention for shared memory resources, such as the last-level cache and memory bandwidth. The resulting models for both applications are competitive in comparison to existing runtime control approaches. For paging mode selection, the resulting model provides equivalent performance to the state of the art while substantially reducing the computation requirements of profiling. For hardware memory prefetcher utilization, the resulting models are the first to provide dynamic control for hardware prefetchers using workload statistics. Finally, a correlation-based feature selection method is evaluated for identifying relevant low-level performance metrics related to hardware memory prefetching.

Chapter 1

Introduction

Modern operating systems and microarchitectures rely on a vast set of algorithmic choices, parameterizations, and heuristic models to facilitate performant resource allocation and program execution. Design decisions often offer a trade-off, improving the performance of some workloads while impairing the performance of others. When these design decisions and parameterizations can be affected at runtime, the system can be tuned or reconfigured to operate in a manner advantageous to the performance of the currently executing workload.

The opportunity for runtime control and configuration is ripe. Since the Nehalem microarchitecture (2008–), Intel has publicly exposed a set of four hardware memory prefetchers which can be enabled or disabled at runtime on each core [132, 67]. IBM POWER7 and later POWER microarchitectures (2010–) expose a highly configurable engine for hardware memory prefetcher control which further offers opportunities to configure prefetcher depth and stride [121]. Intel's Resource Director Technology is an emerging toolkit for hardware monitoring and resource allocation, available for the Xeon microarchitecture, which further expands the available system control mechanisms to include the ability to partition and assign cache ways to specific programs or threads (Cache Allocation Technology) and measure and throttle memory bandwidth usage per-core (Memory Bandwidth Monitoring, Memory Bandwidth Allocation). Effective utilization of these tools is an active body of research [70, 62, 69, 140, 100, 141, 142]. Additionally, there are many bespoke and application-specific examples of system control available in the computer systems literature, including paging mode utilization in virtual machine memory managers [15, 136, 80, 61], thread and data-center scheduling [120, 131, 41], power consumption control [38, 120], and feedback-directed optimization in virtual machines [7, 110, 32, 34, 109].

In *static* runtime control utilizes a fixed configuration for the duration of a workload's execution. In contrast, *dynamic* runtime control makes use of fine-grained profiling to affect system operation and adapt system capabilities in favor of the current system and workload characteristics. Developing effective models for dynamic runtime control can be challenging due to limited feedback. Performance measurements obtained through profiling only provide partial information, limited to the system configuration under which the profiling occurred. Feedback for alternative configurations can not be measured simultaneously with the same execution. One common method for providing comparative profiling relies on identifying representative regions of program execution or developing micro-benchmarks which are representative of certain types of workload behavior. Enumerative profiling of representative regions for the full set of available configurations provides full-information feedback, which is directly comparable within that region.

Determining which profiling metrics which are relevant to a runtime control decision presents a similar challenge of limited feedback. The Performance Monitoring Unit (PMU) [66, 6], available on most modern architectures, is a ubiquitous mechanism for measuring and characterizing system and workload behavior [52, 18, 146, 83]. The PMU exposes a large number of architecture events which can be measured at runtime using a small set of performance counters. Modern microarchitectures expose thousands of unique events but only provide up to eight performance counter registers with which to measure those events. There is significant pressure for these performance counter registers, as the number of performance events has far out-paced the number of events which can be measured simultaneously [149]. This is further complicated by the nature of those performance events. Which events are exposed by a particular system varies substantially both between vendors and between microarchitectures of the same vendor. Statistical sampling can allow for larger sets of performance events to be sampled at the cost of measurement error [14]. However, even with statistical sampling, measuring the full suite of performance events is both impractical, due to the incurred measurement error, and unnecessary, as a substantial number of events will be irrelevant or redundant to the application.

Instead, a subset of relevant and representative performance events should be chosen to drive runtime control. However, selecting relevant and representative performance events is often a laborious

process. Reasoning about the relationship between low-level performance events and the resulting effect on configuration performance is often challenging even for domain experts. Performance event documentation, when available, is often terse, vague, and in some cases incorrect. Many performance events describe components or behaviors which are specific to a microarchitecture, and there are few performance events which are standardized across microarchitectures. Even for performance events which are consistently available, the relationship between those events and performance will also depend on microarchitecture design and the interaction between components.

This dissertation presents a framework for the uniform modeling of fine-grained, dynamic runtime control problems which are informed by measurable statistics of microarchitecture and workload behavior. This framework provides a simple and direct method for constructing effective runtime control models while mitigating the time cost and domain expertise required to achieve that performance. More specifically, the framework models fine-grained, dynamic runtime control as a contextual bandit [12]—a mathematical model which describes sequential decision making with so-called *bandit feedback*, which is representative of the limited feedback produced when profiling performance due to a control decision. At each iteration, the bandit observes some contextual information (workload behavior, according to performance event measurements), and uses that *context*, as well as existing domain knowledge about the control mechanism, to select an *action* (system configuration). In response, the bandit receives a *reward* (performance) dependent on both the context and selected action. By exploiting established off-policy contextual bandit methods (e.g., Binary-Offset [21]), profiling data that is obtained from random system control can adequately and efficiently capture the relationships between workload behavior, system configuration, and performance.

This work focuses on two motivating applications. The first application, paging mode selection, considers the trade-off in performance between common virtual memory abstractions (Shadow Paging, Hardware-Assisted Paging) in the Xen [16] virtual machine memory manager. The performance of each paging mode will favor certain types of memory access patterns. Shadow Paging introduces additional overhead to page table activity, which will, in turn, adversely affect the performance of workloads with larger memory working sets. Conversely, Hardware-Assisted Paging introduces additional overhead to the translation-lookaside buffer, which will, in turn, adversely affect the

performance of memory-intensive workloads. The second application, hardware memory prefetcher utilization, considers the configuration of existing hardware prefetchers. Prefetching is an effective tool for mitigating the cost of accessing DRAM. Future memory accesses are predicted and requested in advance of their potential use, ensuring that the requested memory is cached or in-flight by the time the memory is required. While prefetching is overwhelmingly effective for single core workloads, the added memory utilization due to prefetching can increase pressure for memory resources, such as the last-level cache and memory bandwidth, which are shared by multiple cores. This contention can be destructive to system-wide performance on multi-core systems. These two applications represent a scaffold in difficulty and complexity. In paging mode selection, there is a binary choice between paging modes, directed by the behavior of specific memory components (page table, translation-lookaside buffer) behavior. In hardware memory prefetcher utilization, there is a combinatorial set of binary choices, selecting to enable or disable each of the four prefetchers on each core. Performance is not dictated by a single decision for a single core, but rather by the interaction of multiple decision, across multiple cores, through the shared last-level cache and memory bandwidth. This is further complicated by the complex interactions between cache and memory components which result in system-wide performance. Identifying which performance events are relevant and effective for dynamic prefetcher control presents a distinct challenge.

The main contributions of this work are three-fold. First, a mapping between dynamic runtime control and off-policy contextual bandits is developed. Leveraging the Binary-Offset algorithm [21], dynamic runtime control models are learned from profiling data acquired by utilizing random runtime control decisions over time. Second, the framework is evaluated for the two motivating applications, paging mode selection and hardware memory prefetcher utilization, with the scaffolded difficulty introducing additional modeling features. Third, a correlation-based feature selection method is described for selecting relevant performance events from the logged random profiling data, and is evaluated for hardware memory prefetcher utilization. The selected performance events are further analyzed in the context of available documentation to show that the events are substantiated by domain expertise.

The remainder of this work is organized as follows. Chapter 2 presents background material and

related work relevant to the system configuration and resource allocation, the contextual bandit, machine learning, and feature selection. Chapter 3 introduces the contextual bandit framework for runtime control, and details the application of this framework to the dynamic selection of performant paging modes in the Xen virtual machine monitor (Hiebel et al. [61]). Chapter 4 further details the application of the framework to the dynamic utilization of hardware memory prefetchers in multi-tenant workloads which suffer from contention for shared memory resources (Hiebel et al. [62]). Chapter 5 presents correlation-based feature selection for selecting performance events relevant to the hardware memory prefetcher utilization. Finally, Chapter 6 summarizes this work and describes several avenues for future work.

Chapter 2

Background

2.1 Performance Monitoring

The Performance Monitoring Unit (PMU) is a commonly available component which allows for microarchitecture event occurrences to be measured at runtime with hardware assistance [66, 6]. The PMU consists of a small number (4–8) of configurable performance counters per CPU. Each performance counter is implemented as a pair of Model Specific Registers (MSRs), with one MSR for measurement and one MSR for configuration. Each counter is configured to observe an event through a two byte identifier: the first byte identifies the event, and the second byte identifies a mask. The event value identifies a distinct, high level event that can take place within the architecture, e.g., branch instructions retired (Intel event `0xC4`, mnemonic `BR_INST_RETIRED`), and the mask specifies some subset of that behavior, e.g., near call branches that are taken (mask `0x20`, mnemonic `BR_INST_RETIRED:NEAR_TAKEN`). When enabled, the processor will increment the performance counter whenever the configured event occurs. Additionally, the PMU offers a small collection of fixed-function performance counters which measure specific events on each core, including instructions that retire execution (`INST_RETIRED.ANY`) and core cycles while the processor core is not in a halt state (`CPU_CLK_UNHALTED.THREAD`, `CPU_CLK_UNHALTED.CORE`). The fixed-function and programmable counters operate independently of one another, freeing the user of using programmable counters to measure instruction throughput (measured as Instructions per Cycle, or IPC).

In addition to measuring event counts directly, the PMU can also be used to facilitate instruction-level profiling. In event-based sampling, a counter is configured to trigger an interrupt after a fixed number of occurrences for a specified event. When that interrupt occurs, the performance

monitor interrupt service routine will then record the PMU and processor state, including the current instruction counter, to a buffer. The result is a sub-sample of instruction pointer values where the sampling interrupt was triggered, which can, in turn, be used to identify code segments which frequently trigger the event of interest. Practically, event-based sampling is accomplished by setting the performance counter of interest to the maximum value, less the desired number of occurrences. When the counter overflows, the PMU will trigger an overflow interrupt which can be used to record the system state. Due to out-of-order execution and interrupt delays, the reported processor state can suffer from "skid": the reported state is several instructions offset from the instruction that triggered the interrupt. Intel's Precise Event-Based Sampling (PEBS) [66] and AMD's Instruction-Based Sampling (IBS) [6] provide low-latency event-based sampling directly in hardware, reducing the frequency of hardware interrupts for collecting sampling results and allowing for a more rich collection of processor state to be recorded (including branch status, data cache status, and load latency). Both PEBS and IBS minimize and bound the potential skid in program state.

Over time, the number of exposed performance events has grown substantially, exceeding a thousand available events on some recent Intel microarchitectures, while the number of performance counters has remained consistent [14, 42, 149]. As a result, only a small number of performance events can be sampled at any given time. Some performance events are limited to specific counter subsets. In many cases, high-level metrics require the measurement of multiple events to calculate. Ratios of events, such as the miss rate of a cache, are a common example which would require at least two events to calculate. The result is an increased demand for the (already scarce) set of programmable counters. Statistical sampling using time-based multiplexing is a common method for providing a larger set of logical performance counters by measuring subsets of performance events on the physical counters in shorter time slices [14]. However, multiplexing will omit some sampling error depending on the number of performance events measured and the sampling period size. Even with multiplexing, it is untenable to measure the full set of performance events simultaneously due to the resulting error. Alternatively, multiple sets of performance events can be sampled over several program executions, and the resulting traces can be merged. However, this is time-consuming of offline analysis and ill-suited for online analysis. Asynchronous events, such as interrupts and I/O events, can cause significant time drifts between individual runs, which, in-turn, complicates the process of merging

offline traces.

Selecting meaningful performance events can itself be a challenge. The set of available events varies substantially between vendors, and even between microarchitectures from the same vendor. Performance events may also describe the behavior of components or operations that are specific to a microarchitecture, and there are few events which are standardized across microarchitectures. Often, there are complex interactions between architectural components which can obscure the meaning of performance events. This is further complicated by poor, and in some cases incorrect, documentation. The terse event descriptions that are published are often difficult to dissect. Without ample documentation (which is often unavailable), it can be challenging, even for a domain expert, to understand the translate the meaning of low-level performance measurements to a high-level impact on application and system behavior [96, 14, 40, 95, 143].

Nevertheless, the PMU is a popular tool for modeling and characterizing runtime system behavior, and affecting system behavior at runtime. A large number of commercial and open source software tools and APIs provide standardized cross-architecture interfaces for managing and operating the PMU. Popular examples include Intel vTune [91], PAPI [27], Perfsuite [82], and Perfmon2 [47]. For high-performance and parallel computing, specialized software suites, such as HPCTOOLKIT [2], PerfExpert [31], and Periscope [53], address scalable performance measurement and analysis for parallel systems and workloads.

Models for runtime modeling and prediction of performance [102, 18, 139, 49, 83] and energy consumption [72, 68, 38, 120, 19, 97, 56, 139] are plentiful, and target a breadth of architectural targets, runtime environments, and workloads. Methods such as Bubble-Up [92, 81] and ADP [146] characterize high-level performance descriptions by using low-level performance data obtained from the PMU. These models assist in translating low-level performance data into high-level, user understanaable descriptions of program behavior. These descriptions can, in-turn, be used to affect system configuration or program implementation in order to optimize performance or reduce energy consumption. In addition to providing insight into the performance characteristics of software, the ability to control the PMU at runtime allows software to self-assess performance and self-tune operation. Runtime control, directed by performance measurement, is a common usage of the PMU

with applications spanning hardware memory prefetcher control [70, 62], efficient bandwidth allocation [69, 141], cache partitioning [140, 100, 142], paging mode utilization in virtual machine memory managers [15, 136, 80, 61], thread and data-center scheduling [120, 131, 41], and power consumption control [38, 120].

2.2 Phase Detection

A program will experience phases—periods of execution in which hardware metrics, including cache misses, branch mispredictions, energy consumption, and instruction throughput, are relatively stable. A phase change is an instance in a program's execution in which the behavior of a program undergoes a significant and noticeable change. For example, a program may be I/O-bound during one period of its execution, and once data has been serialized into memory, the program may become cpu-bound in a subsequent phase. Program phases can be observed at multiple granularities, with metrics showing stability over periods of tens of millions to tens of billions of instructions. Further, changes in phase typically occur across several hardware metrics simultaneously, suggesting that the characteristic behavior of the program is changing at those times. Phases can, and often will, reoccur multiple times during a program's execution [118]. Sherwood and Calder [114] illustrate that all but one benchmark program from SPEC CPU95 [124] either exhibited constant behavior for a majority of execution, or exhibited a cyclic, repeatable pattern of phases during that time.

Dhodapkar and Smith [43] identify phase changes by detecting changes in the instruction working set (segments of utilized memory regions) between multiple periods of execution. Basic Block Distribution Analysis (BBDA) (BBDA) [116, 117, 119] estimates the frequency in which each basic block is executed during a period. The result is a Basic Block Vector (BBV) describing a fingerprint of basic block utilization. By comparing the vector difference between BBVs in sequence over time, discovering phase changes amounts to a signal processing problem. Further, BBVs can be clustered in to identify repeated phase behavior. In addition to offline analysis and discovery, phase detection can also be performed online [98].

Alternatively, phase detection can be formulated as change-point detection applied to a signal of

performance characteristic sampling, such as IPC measurements at fixed intervals. A change-point is a time at which the statistical properties of a signal change. The segments between change-points will consist of stable periods of homogenous measurements corresponding to phase-like behavior. While change-point detection can broadly identify phases as periods of stable performance characteristics, further analysis would be required to identify periods corresponding to repeating or cyclic phase behaviors.

Consider a sequence of performance measurements $(y_{0:n}) = y_0, \ldots, y_{n-1}$ and a sequence of ordered indices $\tau_0, \tau_1, \ldots, \tau_{m-1}$ ($\tau_0 = 0$ and $\tau_{m-1} = n$). A common approach to change-point detection is to determine the indices $\tau_{0:m}$ which, when segmenting $(y_{0:n})$, minimize the penalized cost

$$\Big[\sum_{i=1}^{m-1} C(y_{\tau_{i-1}:\tau_i}) \Big] + f_\beta(m) \tag{2.1}$$

where C is the cost function for a segment (statistical critera), and $f_\beta(m)$ is a penalty to guard against overfitting [78]. The cost function describes the statistical properties of a segment determined by two change-points. The more probable that the distribution of the segment changed at some internal point, the most costly the segment should be weighted. Common examples of cost functions include the negative log-likelihood [64], cumulative sum of squares [65], and quadratic loss. The penalty is typically linear with respect to the number of change points, $f_\beta(m) = \beta\, m$. The relative weight of each change-point's penalty, β, determined by some information-theoretic criterion based on the number of parameters p which are introduced when a change-point is introduced: Akaike's Information Criterion (AIC) [5], $\beta = 2p$, Bayesian Information Criterion (BIC) [111], $\beta = p \log n$, Modified Bayesian Information Criterion (BIC) [150], $\beta = \frac{3}{2}\, p \log n + \frac{1}{2n} \sum_{i=1}^{m-1} (\tau_i - \tau_{i-1})$.

Binary Segmentation [112] is an approximate recursive method with repeatedly considers the single change-point form of Equation 2.1. Segment Neighborhood [13] and Pruned Exact Linear Time (PELT) [78] solve Equation 2.1 exactly using dynamic programming. Segment Neighborhood requires an upper limit on the maximum number of change-points, whereas PELT dynamically determines the number of change-points. With regards to phase detection, there is a reasonable expectation that for some programs the number of phase changes will depend on the execution time

of the program. As such, PELT is well-suited to phase change detection.

2.3 Memory Virtualization

Virtualization technology is a key component for data center management which, by simulating the functionality of hardware, allows multiple operating systems and applications to operate concurrently on the same physical machine. A virtual machine (VM) is a software container which provides hardware emulation. A virtual machine manager (VMM), or hypervisor, such as Xen [16] or VMWare [133] manages a collection of independent VMs (guests) and facilitates the illusion of direct native hardware access to each. Shadow structures are used to replicate the primary structures used by the guest, such as the page table; however, the additional layer of abstraction will unavoidably introduce overhead compared to the performance of a native system. In fully virtualized systems, the guests run without modification and with no knowledge that the guest is executing on a VM. Privileged operations are "trapped" by the VMM, so that the VMM can gain control of the system and emulate the operation before returning control to the guest. In paravirtualized systems, the guests are modified to directly call VMM-specific code to facilitate operations that require hardware emulation.

The memory management unit (MMU) is responsible for translating the virtual address space made available to a process to the physical address space in the hardware. The virtual and physical address spaces are divided into pages: fixed-size ranges of addresses, commonly 4 KB. A page table manages the mapping between virtual pages and physical pages, and the MMU consults the page table in order to translate virtual addresses into physical addresses. Due to the cost of walking the page table structure to find the desired mapping, the MMU will cache recent translations using the translation lookaside buffer (TLB). With virtualization, the physical memory of the guest is itself a virtual address space. The VMM must virtualize the MMU and facilitate a translation from either the virtual or physical addresses of the guest into machine addresses for use on the hardware.

Both Shadow Paging (SP) and Hardware-Assisted Paging (HAP) are common memory virtualization techniques for fully virtualized systems. Both techniques utilize an additional paging structure in

the VMM which manages the mapping of guest addresses (virtual or physical) to machine addresses. The performance of either paging mode is dependent on workload, as both are subject to different types of overhead costs [24, 54, 1, 136].

In Shadow Paging, the VMMM maintains a shadow page table in parallel with the page table maintained by the guest. The shadow page table maps virtual addresses in the guest directly to machine addresses, bypassing the virtual to physical translation of the guest entirely. The shadow page table maps virtual addresses in the guest directly to machine addresses, bypassing the virtual to physical address translation of the guest all together. The shadow page table supersedes the guest page table, and the VMM installs the shadow page table. As updates to the guest's page table must be reflected in the shadow page table, expensive VM exits are required to maintain page table synchronization. This in turn increases the overhead of page table activity. This will have a significant, negative impact on workloads which suffer from a large number of page faults.

In Hardware-Assisted Paging, the VMMM maintains an extended page table (EPT) [54] or nested page table (NPT) [23] in sequence with the guest page table. Hardware support in the MMU performs the sequential mapping, first by translating virtual addresses into physical addresses using the guest page table, and then translating the physical addresses into machine addresses using the the extended/nested paging table. Unlike SP, page table updates do not require synchronization and expensive VM exits; however, the two-layer paging structure increases the cost of page table walks which, in turn, increases TLB miss latency. This will have a significant, negative impact on workloads with poor locality and a large working set (which exceeds the size of the TLB), as cached mappings will be evicted from the TLB before they are reused.

2.4 Hardware Memory Prefetching

Hardware memory prefetching is an effective technique for mitigating memory access latency. By observing and exploiting patterns in a program's memory accesses at runtime, a hardware prefetcher can generate memory requests ahead of the true request so that the desired memory is available (in the cache) or in-flight when demanded by the program. Hardware prefetching effectiveness

is determined by the coverage (proportion of misses that are eliminated because of prefetching), accuracy (proportion of prefetch targets which resulted in a hit), and timeliness (the latency between a prefetch targets availability vs reference) of the predicted prefetch targets. Inaccurate and untimely prefetching can place undue stress on memory resources, increasing memory bandwidth utilization and polluting the cache unnecessarily. It is also possible that untimely prefetch targets may be evicted from the cache before their use, prompting the memory to be fetched an additional time. Hardware prefetchers can struggle to obtain coverage in the presence of short streams, where the prefetcher does not have the opportunity to detect the direction and distance of the stream or stride, or when memory is accessed in irregular patterns [87]. While prefetchers are often effective in predicting memory accesses in a single-threaded setting, the increased utilization of, and contention for, shared memory resources such as memory bandwidth and the last-level cache can be destructive to multi-core performance [77, 93].

Prefetcher aggressiveness prevents a tradeoff in coverage, accuracy, timeliness, and resource utilization. An aggressive prefetcher will attempt to work well ahead of a detected memory access stream, relying on speculation in order to hide as much access latency as possible. The result is a likely increase in coverage and timeliness, at the expense of lower accuracy and an increase in memory bandwidth utilization and cache pollution due to the traffic and cache allocation of the incorrectly predicted prefetch targets. In contrast, a conservative prefetcher will attempt to operate with less speculation and more directly in response to current memory accesses. The result is a likely increase in accuracy, without the added cost of increased memory bandwidth utilization and cache pollution, but at the expense of lower coverage and worse timeliness.

Broadly, hardware prefetchers exploit both spatial and temporal locality in order to determine prefetch targets [94]. Stream prefetchers detect fixed-stride access patterns and on a cache miss will fetch one or more subsequent blocks along that stride, under the assumption that those cache lines will likely contain targets of future memory requests [73, 103, 35]. Correlation-based and Markov prefetchers [71, 115, 123] predict targets that may be the result of complex array accesses or pointer-chasing, allowing for a greater coverage on a broader set of memory access patterns. Prefetcher aggressiveness can be dynamically directed in hardware using feedback regarding the

accuracy, lateness, and cache pollution due to hardware prefetching [39, 123, 36, 106].

Since Nehalem (2008–), Intel microarchitectures have come equipped with four configurable prefetchers which can be enabled or disabled at runtime [132, 67]. Each prefetcher is configured independently of one another, and independently on each core, using the first four bits of each core's Model Specific Register (MSR) `0x1A4`, with `0` indicating the at the corresponding prefetcher should be enabled, and `1` indicating that it should be disabled. By default, all four hardware prefetchers are enabled across all cores. The first pair of prefetchers, the *Data Prefetch Logic* (DPL) and *Adjacent Cache Line* (ACL) prefetchers, operate on the L2 cache. The DPL is a stream prefetcher which is capable of detecting both ascending and descending sequences of accesses issued from the L1 cache within 4 K page boundaries. The prefetcher is capable of detecting and maintaining up to 32 data access streams, with up to one forward and one backward stream per page. Recent microarchitectures have refined the operation of the prefetcher in order to better address memory contention concerns: when there are few outstanding memory requests, the DPL will operate up to 20 lines ahead of the most recent load request in the stream; when there are many outstanding memory requests, the DPL will operate in a more restricted fashion and will only cache the prefetched memory in the last-level cache. The ACL is a spatial prefetcher which fetches adjacent cache lines which form a 128-byte aligned pair. While more restrictive than the DPL, the ACL prefetcher does not require a detected access stream to operate. The second pair of prefetchers, the *Data Cache Unit* (DCU) and *Instruction Pointer* (DCU IP) prefetchers, operate on the L1 cache. The DCU is an ascending stream prefetcher which reacts to ascending accesses in recently loaded data. The accesses are assumed to be part of access stream and the immediately following line is prefetchers. The DCU IP is an ascending/descending stride prefetcher which operates on half page (2 K) limits. Both prefetchers are only triggered under a restricted set of conditions, including a low load miss rate and the lack of a memory barrier in the pipeline.

Since POWER7 (2011–), IBM POWER microarchitectures have come equipped with a highly configurable hardware prefetching engine for an L1 stream prefetcher [121, 122, 58, 70]. The stream prefetcher is capable of detecting and exploiting up to 16 independent data streams resulting from memory requests in the L1 cache. Prefetching can be configured to enable or disable the detection

of load and store streams, and can independently be configured to enable or disable the detection of streams with non-unit strides. In addition, the depth of the stream buffer can be specified as one of six broad categories, from "shallowest" to "deepest". By default, load streams are enabled, both store streams and non-unit stride streams are disabled, and the prefetcher depth is set to "deep" (below "deeper" and "deepest"). Recent improvements in the POWER7+ and POWER8 microarchitectures have included configuration for prefetcher urgency to direct how aggressively the prefetcher will operate to attain the specified depth when a stream is detected.

2.5 Multi-Armed Bandits

First developed in Robbins [107], the multi-armed bandit [107, 55, 33, 28] describes a sequential decision process with limited feedback. The bandit selects some actions (or arms) to play in sequence, and in response to each action the bandit receives a (potentially stochastic) reward for that action. The rewards for the remaining actions are unobserved (so called "bandit feedback"). The goal of the multi-armed bandit is to select actions which will maximize the cumulative reward received. To start, the bandit has no knowledge of how rewarding each action will be, and must balance exploration (selecting actions to model the rewards of each action) and exploitation (selecting actions which are strongly believed to be optimal). The origin of the term multi-armed bandit comes from the slang "one-armed bandit", describing an old-style slot machine operated by pulling a long handle at the side.

Clinical trials are a historical motivation for the multi-armed bandit. In a clinical trial, each patient can be assigned only one treatment (action), and only the result of that treatment can be measured for a particular patient (bandit feedback). There are two conflicting goals involved in this process: first, to correctly identify the best treatment (requiring exploration), and second, to provide the best standard of care to the patients in the trial (requiring exploitation). As the trial continues, there is an obligation, especially in the case of a severe disease, to dynamically adjust treatment selection so that the selections favor more rewarding options. However, less rewarding treatments must still be utilized (with less frequency) to prevent the trial from greedily exploiting a suboptimal treatment

due to non-representative samples early on.

Formally, the multi-armed bandit problem is defined by a set of $K \geq 2$ possible actions and sequences $(X_a) = X_{a,1}, X_{a,2}, \ldots$ of rewards for each action a. At each time step $t = 1, 2, \ldots$, the bandit will select some action a_t, and will collect reward $X_{a_t,t}$ in response. The behavior of a bandit is determined by the action selection strategy. The quality of a particular strategy is expressed in terms of the *regret*, or lost reward, accumulated by selecting suboptimal actions,

$$R_n = \max_a \big(\sum_t^n X_{a,t} \big) - \sum_t^n X_{a_t,t} \tag{2.2}$$

as the difference between the total reward of the best performed action (the optimal strategy) and the total reward obtained by by the bandit over n selections. In practice, this from of regret is not practical to estimate, as the reward may be chosen according to some stochastic or adversarial process. Instead, the *pseudo-regret*,

$$\overline{R}_n = \max_a \ \mathbb{E}\big[\sum_t^n X_{a,t} - \sum_t^n X_{a_t,t} \big] \tag{2.3}$$

measures the regret compared to the action which has the optimal expected reward, as opposed to measuring the regret over the selected rewards rewards. The goal is to determine a strategy which minimizes the potential regret of the bandit.

The structure of the rewards will strongly influence action selection strategy. Rewards can be drawn stochastically or adversarially. In the stochastic bandit case, the rewards for each arm, (X_a), are independent and identically distributed (*i.i.d.*) according to some distribution ν_a. In the adversarial (non-stochastic) bandit case, the rewards are assumed to be generated by some adversary. The adversary is allowed to assign rewards with full knowledge of the bandit's selection process, but must make the reward assignments before the bandit selects and reveals its action (otherwise, the adversary could simply assign an arbitrary reward to the selected action). The adversarial setting illustrates the need for minimizing regret, as opposed to maximizing reward. If the adversary were simply attempting to minimize the bandit's accumulated reward, then it could simply set the poor rewards for every action. Instead, like in a rigged casino, the adversary attempts to maximize the

reward that the bandit could have accumulated with optimal selections.

Whereas the standard multi-armed bandit formulation considers an exploration-exploitation trade-off, pure exploration bandits [29, 9, 30, 76] instead consider a distinct exploration phase which is constrained to a fixed number of action selections. During the exploration phase, the bandit selects actions and obtains bandit feedback in order to identify the optimal action with high confidence, so that the recommended action can be exploited in the subsequent phase. Only the regret of the recommended action is considered. An example application for pure exploration bandits, given in Audibert and Bubeck [9], considers channel allocation in mobile networks. Before transmission, a transmitter can first explore the set of (noisy) channels, for a brief period, to identify which channel will be the best over which to communicate. Transmission is then performed over the channel that the bandit believes to be the best.

2.5.1 Selection Strategies

Action selection strategies are responsible for carrying out the exploration-exploitation balance of a bandit. Several actions election strategies appear commonly in the literature, introduced and modified to bound the regret of the bandit in some theoretical capacity. For the stochastic bandit, this includes strategies such as ϵ-Greedy [129], upper confidence bounds [11], and Thompson Sampling [4]; for the adversarial bandit, the exponential-weight algorithm for exploitation and exploitation [12].

ϵ-Greedy

The ϵ-Greedy approach balances exploration and exploitation at random. The bandit tracks the mean observed rewards for each action. Most of the time, the bandit selects the action with the greatest mean reward. Otherwise, with some small probability ϵ, the bandit instead selects an action to take at random. For the stochastic multi-armed bandit, in the case of a fixed ϵ, the regret of

ϵ-Greedy is grows linearly with respect to the horizon:

$$\begin{aligned}
\overline{R}_n &= \max_a \mathbb{E}\big[\sum_t^n X_{a,t} - \sum_t^n X_{a_t,t}\big] \\
&= n\,\mu^* - \sum_t^n \mathbb{E}\big[\mu_{a_t}\big] \\
&= n\,\mu^* - n\left[(1-\epsilon)\,\mu^* + (\frac{\epsilon}{K})\sum_a \mu_a\right] \\
&= n\,(\frac{\epsilon}{K})\sum_a (\mu^* - \mu_a)
\end{aligned} \tag{2.4}$$

where μ_a are the mean rewards drawn from ν_a and $\mu^* = \max_a \mu_a$. With careful annealing of the ϵ value, by scaling ϵ inversely proportional to time, the regret can instead be bounded logarithmically [11].

Upper Confidence Bounds

Upper Confidence Bound (UCB) methods facilitate the exploration-exploitation tradeoff by selecting actions according to bounds on each action's reward which hold with high probability. The bandit tracks the mean observed rewards, $\hat{X}_a$, and the total number of plays, T_a, for each action. With each selection, the bandit strengthens the estimate of the selected action's mean reward. In response, the bound on the reward will shrink towards the estimate.

For the stochastic bandit, the reward observations for each action are *i.i.d.* random variables. When the rewards are further bounded to the unit interval $[0, 1]$, the difference between the estimated and true mean reward of each action can be bounded probabilistically according to Hoeffding's inequality [63]:

$$P(\mu_a \geq \hat{X}_a + U_a) \leq e^{-2\,T_a\,(U_a)^2}, \tag{2.5}$$

where μ_a are the mean rewards drawn from ν_a, and U_a are upper bounds on the estimates for each action. The probability is bounded by some small $p = e^{-2\,T_a\,(U_a)^2}$ such that it is unlikely the upper-bounded reward estimate exceeds the true reward. For any $p \ll 1$, the reward estimate falls

below the bound

$$\hat{X}_a + U_a = \hat{X}_a + \sqrt{-\log p / 2\, T_a}. \tag{2.6}$$

almost always. A similar bound can be found for sub-Gaussian distributions. If the observed rewards are not bounded, but instead are sub-Gaussian random variables with variance σ^2, then the difference between the estimated reward and the true reward can be bounded as

$$P(\mu_a \geq \hat{X}_a + U_a) \leq e^{-t\,(U_a)^2 / 2\,\sigma^2}, \tag{2.7}$$

which produces the upper-bound reward estimate

$$\hat{X}_a + U_a = \hat{X}_a + \sqrt{-2\sigma^2 \log p / T_a}. \tag{2.8}$$

UCB greedily selects the action a which maximizes the upper-bounded reward estimate $\hat{X}_a + U_a$. When selected, the upper-confidence estimate of the action will shrink, as the estimate of the sample mean is less likely to deviate from the true mean with the larger sample size. Actions with a smaller estimated reward will be selected on occasion, after the most rewarding action is selected sufficiently often to shrink the upper-bound estimate.

The UCB1 algorithm [11, 10] decreases the probability over time according to the schedule $p = t^{-4}$, which results in an upper confidence estimate of

$$\hat{X}_a + U_{a,t} = \hat{X}_a + \sqrt{2 \log t / T_a}. \tag{2.9}$$

Note that the upper-bound estimate is very generous, given the loose assumptions for Hoeffding's Inequality (bounded on $[0, 1]$). The pseudo-regret for UCB1 is bounded logarithmically:

$$\overline{R}_n \leq 8 \sum_{a\,:\,\mu_a < \mu^*} \Big[\frac{\log n}{\mu^* - \mu_a} + \big(1 + \frac{\pi^2}{3}\big)(\mu^* - \mu_a) \Big] \tag{2.10}$$

The first term indicates that each action will be selected a logarithmic number of times with respect to the number of plays, and actions which are close to optimal will be selected more often. The

second term indicates a small number of expected plays which are required to address unlikely cases. The asymptotic behavior of the pseudo-regret for UCB1 is bounded sub-linearly:

$$\overline{R}_n \in \mathcal{O}(\sqrt{K\, n \log n}). \tag{2.11}$$

Thompson Sampling

Thompson sampling is a probability matching technique which models the prior distribution of the mean rewards for each action. Actions are selected according to the corresponding posterior distributions—the bandit samples from each action's posterior distribution, and selects the action with the greatest sample mean. After the reward is observed, the prior distribution for the selected action is updated accordingly.

Consider a bandit with Bernoulli rewards, $X_{a,i} \sim Bernoulli(\mu_a)$. After a sequence of actions, the conjugate priors $\mu_a \sim Beta(\alpha_a, \beta_a)$ describe the distribution of the sample mean for action a, where $\alpha_a - 1$ denotes the number of successes (reward = 1) and $\beta_a - 1$ describes the number of failures (reward = 0). To begin, before any actions have been selected, the priors for every action are initialized to $Beta(1, 1)$. This corresponds to the uniform case, where each value of the mean is equally likely. The Bernoulli bandit generates samples of the mean from each action's prior, and chooses the action with the largest sample. The prior of the selected action is then updated according to the reward, $(\alpha_a, \beta_a) \leftarrow (\alpha_a + X_{a,t}, \beta_a + (1 - X_{a,t}))$. The asymptotic behavior of the pseudo-regret for Thompson sampling for the Bernoulli bandit is bounded sub-linearly:

$$\overline{R}_n \in \mathcal{O}(\sqrt{K\, n \log n}). \tag{2.12}$$

Exponential-Weight Algorithm for Exploration and Exploitation

In contrast to the previous selection strategies, the Exponential-Weight Algorithm for Exploration and Exploitation (EXP3) [12] considers action selection for the adversarial bandit with bounded rewards. Without loss, assume the actions are bounded to the interval $[0, 1]$. Instead of measuring

regret in terms of the accumulated reward, the EXP3 strategy measures regret in terms of the accumulated loss, $l_{a,t} = 1 - X_{a,t}$, inflicted by the adversary for choosing an action a. The bandit incorporates randomness to the action selection strategy in order to subvert the adversarial selection of rewards. With randomness, a sub-linear regret bound is achievable. Without randomness, the adversary can perfectly emulate the selection strategy and maximize the loss for the predicted action.

The bandit tracks estimates of the cumulative loss for each action, $\hat{L}_a$. Actions are selected randomly with probability proportional to an exponential weighting of the cumulative loss estimate,

$$p_a \propto e^{-\eta_t \hat{L}_a}, \tag{2.13}$$

where η_t is a non-increasing schedule of weights. Rather than use the losses imparted by the adversary directly, the bandit accumulates the unbiased estimation of the loss, $\hat{l}_{a_t} = {}^{l_{a_t,t}}/_{p_{a_t}}$. For the decreasing schedule $\eta_t = \sqrt{{}^{\log K}/_{t\,K}}$, the asymptotic behavior of the pseudo-regret for EXP3 can be bounded sub-linearly:

$$\overline{R}_n \in \mathcal{O}(\sqrt{n\,K \log K}). \tag{2.14}$$

2.5.2 Contextual Bandits

The contextual bandit [12, 22, 85, 89] (alternatively, the partial label problem [74], the associative bandit problem [127], bandits with side information [134, 135, 147], bandits with a concomitant variable [137], associative reinforcement learning [12]), extends the multi-armed bandit to include side-informaation in the decision procedure. Prior to selecting an action, the contextual bandit first perceives some information $\vec{x}_t$ about the environment (a context) in which the action will occur. The resulting reward observed by the contextual bandit depends on both the action and the context.

A common application of the contextual bandit is the personalized selection of internet advertisement. Websites have a wealth of logs which detail historical usage: observable quantities about the user, such as the visiter's history, queries, and provided personal data (context), the advertisements served to the visiter (action), and whether or not the visiter clicked on the advertisement (Bernoulli reward). The goal is to maximize the click-through rate of advertisements by selecting

advertisements which are appropriate to the visiter and thus rewarding to the website.

In order to manage the complexity introduced by the addition of contextual information, assumptions on the context space and reward are introduced. In the simplest case, when the contextual information specifies an element from a finite set of contexts, the contextual bandit can be interpreted as a set of independent, context-free bandits, indexed by the context [134, 135]. When the context is drawn from a vector space, a common constraint is to assume that the reward is linear [37, 3] or Lipshitz [89] in expectation. Regardless of the constraints, the selection strategies underpinning various solutions to the contextual bandit generally follow the selection strategies for the multi-armed bandit: greedy [85], upper confidence bound [37], Thompson sampling [3], and adversarial models [22].

Off-policy methods [86, 44] utilizes the partial-label results of a contextual bandit in order to construct an action selection strategy from logged data. Similarly to pure exploration bandits in the multi-armed case, exploration and exploitation are not interleaved; rather, exploration data is constructed from the exploration of a previous action selection strategy (as tuples of context, action, and reward), and the resulting log data is used to construct a selection strategy which is then exploited. The advantage of using log data is its ubiquity—many selection strategies can be evaluated against the logged exploration data without requiring *in situ* evaluation, which may be impractical and costly. In contrast to pure exploration bandits, the exploration procedure is not the focus. Due to the limitation of bandit feedback, log data is insufficient to directly simulate the result of some new selection strategy. One alternative is to estimate the contextual reward function directly using the available log data, and use the resulting regression model for simulating selection strategies. Another alternative is to utilize inverse propensity scoring (IPS) [108] to shift the proportion of actions between the log data and the selection's actions [86]. Combining both the direct and IPS methods addresses the deficiencies of each [44]. Instead of modeling the reward directly, the Offset-Tree [21] instead maps the logged partial label data to a weighted classification problem. The resulting weighted classification data is then amenable to a broad suite of machine learning techniques for feature selection, dimension reduction, and classification.

2.6 Supervised Learning and Classification

In machine learning, classification is the supervised learning task of categorizing new instances based on a training set of data containing observations (or instances) whose category membership is known. For example, determining a diagnosis for a patient given a set of diagnostic test results, categorizing emails as either spam or not spam given the contexts of the email, or labeling an image according to the object represented.

Consider a training set of labelled instances $(\vec{x}_1, y_1), (\vec{x}_2, y_2), \ldots, (\vec{x}_n, y_n)$, each described by a vector $\vec{x}_i$ of quantifiable features, and a label (class) y_i drawn from a finite set of categories. Each feature may be nominal, ordinal, or numerical. A nominal feature takes the form of a set of options which have no intrinsic ordering, e.g., sex, $\{male, female\}$, blood type, $\{A, B, AB, O\}$, or boolean-valued sets, $\{true, false\}$ or $\{spam, notspam\}$. This is sometimes also referred to as a categorical feature. A ordinal feature takes the form of a set of options that have an intrinsic ordering, e.g., sizes, $\{small, medium, large\}$, or letter grades, $\{F, D, C, B, A\}$. Numerical features are drawn from (subsets of) some number space, e.g., integers, $\mathbb{Z}$, or real numbers, $\mathbb{R}$.

Classifiers attempt to select some function, or hypothesis, $h(\vec{x})$ which categorizes instances by mapping the feature vector $\vec{x}$ to a predicted category y. Ideally, the classifier attempts to maximize the accuracy of the classes predicted by the hypothesis when presented with novel (unlabeled) instances.

Deterministic (non-probabilistic) classifiers, such as the support vector machine, separate the feature space in to regions and associate each region with a class. When presented with a new instance, the classifier returns the class associated with the region containing the instance. Probabilistic classifiers, such as Logistic Regression and Naïve Bayes, instead generate a distribution describing the probability that the instance is a member of each class. The instance can then be labelled according to the class with the largest probability. Alternatively, the classifier can abstain from providing a class if there is insufficient confidence for any of the classes. The collection of classifier methods is rich and detailed, and can not be completely described here.

The Support Vector Machine (SVM) is a non-probabilistic, binary classifier which models the clas-

sification boundary separating instances of the training data, according to class, with the largest margin. Consider a binary classification problem with class labels $\{+1, -1\}$ (positive and negative labels, respectively). In the hard-boundary case, where training data instances of each class are linearly separable, i.e., that there exists some linear hyperplane which separates the positive and negative instances, the support vector machine can be modeled by the constrained optimization problem

$$\begin{aligned} \underset{\vec{w}, b}{\arg\min} \quad & \frac{1}{2}\|\vec{w}\|^2 \\ \text{subject to} \quad & y_i \ (\vec{w} \cdot \vec{x}_i - b) \geq 1, \end{aligned} \tag{2.15}$$

where $\vec{w} \cdot \vec{x}_i - b \geq 1$ and $\vec{w} \cdot \vec{x}_i - b \leq 1$ represent the parallel linear functions comprising the margin and separating the positive and negative instances of the training data. The instances which fall directly on the margin, and thus constrain (or support) it, are referred to as support vectors. The margin can be completely determined by these support vectors.

More generally, when the training data instances are not linearly separable, a collection of slack variables can be added to allow for instances to violate the margin constraint at the expense of some error. The soft-margin SVM is be modeled by the constrained optimization problem

$$\begin{aligned} \underset{\vec{w}, b, \xi_i}{\arg\min} \quad & \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n} \xi_i \\ \text{subject to} \quad & y_i \ (\vec{w} \cdot \vec{x} - b) \geq 1 - \xi_i, \\ & \xi_i \geq 0, \end{aligned} \tag{2.16}$$

where ξ_1, ξ_2, ..., ξ_n are slack variables, and C is a hyper-parameter representing the relative weight between the margin size and slack variable error. The slack variables represent a form of hinge-loss, $\max(0, 1 - (y_i \ \vec{w} \cdot \vec{x}_I - b))$. When the margin would lead to correctly classifying a training instance, $\xi_i = 0$ and the instance contributes no additional loss to the optimization function. Otherwise, the instance accumulates loss linearly with regards to it's distance from the boundary.

As the soft-margin formulation satisfies the Karush-Kuhn-Tucker (KKT) conditions [], Equation 2.16 can be reformulated as the quadratic optimization problem (the so-called dual form),

$$\begin{aligned} \arg\max_{c_i} \quad & \sum_{i}^{n} c_i + \frac{1}{2} \sum_{i}^{n} \sum_{j}^{n} y_i c_i (\vec{x}_i \cdot \vec{x}_j) c_j y_j \\ \text{subject to} \quad & \textstyle\sum_{i}^{n} c_i y_i = 0, \\ & c_i \geq 0, \\ & c_i \leq \frac{1}{2n\lambda}, \end{aligned} \tag{2.17}$$

where c_i are Lagrange multipliers (or Kuhn-Tucker coefficients) [90]. Quadratic programming solvers allow for the efficient optimization of Equation 2.17. When the margin would lead to correctly classifying a training instance ($c_i = 0$), the instance contributes no additional loss to the optimization function. When $0 < c_i <= \frac{1}{2n\lambda}$, the training instance exists on or across the margin and the resulting instance is a support vector. The weight vector

$$\vec{w} = \sum_{i} c_i y_i \vec{x}_i \tag{2.18}$$

can be computed as a linear combination of the support vectors (as $c_i = 0$ for any training instances that are not support vectors).

In some problems, training instances may not be of equal value. Prior knowledge might dictate that the quality of instances differ in a quantifiable manner, or that the relative importance of instances, and thus the weight those instances should impact on the classifier, can vary. In an instance-weighted classification problem, each training instance is accompanied by a positive weight value. That weight will scale the loss contributed by the corresponding instance. The meaning of that weight, and whether that weight is constrained to a given range, will depend on the problem. Instance weights can be directly incorporated into the formulation of a classification algorithm, by using weights as coefficients in the classification loss (e.g., for SVMs [145]). Alternatively, the instance weights can be used to sample instances from the training data set, resulting in higher weighted training instances being selected with a proportionally higher probability than lower weighted training instances. The resulting, unweighted training data set is then amenable to the full suite of classical, unweighted classification methods. Typically, a collection of training data set samples are taken, and the resulting

classifiers constructed for each set are combined in to an ensemble. The probability can be taken as the normalized weight values. Zadrozny et al. [148] instead proportion the selection of weight according to the maximum weight value.

Convex loss classifiers, such as Logistic Regression and SVMs, are popular due to the existence of efficient numerical solvers. However, these methods are sensitive to the presence of class label noise. The loss of mislabeled instances will increase with respect to the distance to the decision boundary (e.g. logarithmically, or linearly with hinge loss). The result is a learned decision boundary which is necessarily skewed towards the label outliers in order to minimize the loss of those outliers. Generally, convex loss functions act as surrogates to the 0-1 misclassification loss, $\frac{1}{n}\mathbb{I}\big[h(\vec{x}_i) = y_i\big]$. While robust to label outliers, optimizing the 0-1 misclassification loss directly is NP-Hard [17]. Label outliers can be addressed by either using a classifier which is robust to label noise, perhaps accepting a non-convex loss function which is less efficient to optimize, or filtering instances which appear to be mislabeled [51].

2.7 Feature Selection

Feature selection is a technique for identifying and removing features which are redundant or irrelevant to an outcome [79, 57]. Focusing the attention of a supervised learning to the subset of useful features has a number of advantages. Most notably, feature selection helps mitigate the effect of the curse of dimensionality: the amount of training data needed to learn grows exponentially with the number of features. There are also implications for training time, as fewer features reduces the set of parameters which must be learned, and for data exploration, as the prominent features can be more easily visualized and interpreted in a lower dimensional space. Feature selection methods can broadly be categorized as filter methods, wrapper methods, and embedded methods:

Filter methods are a computationally efficient class of feature selection algorithms which operate directly on the characteristics of the training data set, relating features to the corresponding classes. As such, they can be thought of as a preprocessing step, first identifying the relevant features before using just the selected features for the learning task. Filter methods typically rank and select features

according to univariate and multivariate measures of those features. For example, Correlation-Based Feature Selection (CFS) [59] selects feature subsets which maximize the heuristic of that subset's *merit*,

$$Merit_S = \frac{k\,\overline{r_{fc}}}{\sqrt{k + (k-1)\,\overline{r_{ff}}}}, \tag{2.19}$$

where S is a subset of k features, $\overline{r_{fc}}$ is the average correlation between the features of the subset and the corresponding classes, and $\overline{r_{ff}}$ is the average correlation between each pair of features in the subset. The merit heuristic rewards feature sets with a high average relevancy (feature-class correlation), and penalizes feature sets with a high average redundancy (feature-feature correlation). Features are selected through a search of the space of all possible subsets. A direct, combinatorial search of the space of feature subsets is computationally intractable, even for a small number of features. Instead, features are selected through greedy search maximizing the selection merit, either through forward selection (starting with the empty set of features and adding new features), or backward elimination (starting with the full set of features and removing features), until merit no longer improves. The result is a nested subset of features, in the order of selection or elimination.

Wrapper methods use the supervised learning algorithm as a black-box method for scoring subsets of features. A wrapper will perform an iterative search of feature subsets, evaluating and guiding the search using the performance characteristics of the resulting models. As with CFS, wrapper methods can make use of both greedy searches, forward selection and backward elimination, to identify nested subsets. The result is a feature selection method which is simple, and incorporates the characteristics of the learning algorithm in a ubiquitous fashion.

Embedded methods directly incorporate the task of feature selection in to the supervised learning algorithm. Unlike both filter and wrapper methods, embedded methods permit alternatives to nested subset searches, and unlike wrapper methods, embedded methods do not require repeated model construction (as is the case with wrapper methods). The Least Absolute Shrinkage and Selection Operator (LASSO) [130] is an example of an embedded method which directly affects the optimization objective instead of performing a nested subset search. LASSO adds an additional L_1 regularization penalty, $\|\vec{w}\|_1$, to the loss function based on the total weights of the resulting linear model $\vec{w}$. This penalty will encourage coefficients $\vec{w}$ corresponding to unimportant features to drop

to 0, and those features are effectively pruned from the resulting model.

Chapter 3

Paging Mode Selection

Virtualization technology is a key component for data center management which allows for multiple users and applications to share a single, physical machine. Modern virtual machine monitors utilize both software and hardware-assisted paging for memory virtualization, however neither paging mode is always preferable. Previous studies have shown that dynamic selection, which at runtime selects paging modes according to relevant performance metrics, can be effective in tailoring memory virtualization to program workload. However, these approaches require low-level manual analysis, or depend on prior knowledge of workload characteristics and phasing.

This chapter introduces the contextual bandit framework for dynamic system control, and considers an application of the framework to dynamic paging mode selection. Paging mode selection presents a controlled first step towards developing the framework, as the action space is binary and the underlying, relevant features are well known and well studied in related work [15, 136, 80]. Technical challenges, such as changing performance characteristics (according to program phase), are used to motivate off-policy contextual bandit methods. The Binary-Offset algorithm [21] and random profiling are presented as effective techniques for constructing a dynamic selection model with equivalent performance to the state-of-the-art ASP-SVM method [80], while requiring substantially less profiling time (2.5 hours compared to over 24 hours) to achieve that performance.

The material contained in this chapter was previously published in the Proceedings of the 47th International Conference on Parallel Processing (ICPP '18) [61].

3.1 Introduction

Virtualization is an essential technology for cloud computing, providing a mechanism for performance isolation and resource utilization. A virtual machine monitor, such as Xen [16] or VMWare [133], presents guest operating systems with a virtual abstraction of a physical machine, while providing mappings between the virtual machine resources and actual hardware. The additional layer of abstraction can introduce performance overhead in many ways. For memory virtualization, there are two techniques taken by modern virtual machine managers: Hardware-Assisted Paging (HAP) and Shadow Paging (SP). Whether HAP or SP performs better depends on the memory access characteristics of a workload. Workloads with a large number of page faults will perform better using HAP. Memory intensive workloads will perform better using SP.

Previous work has proposed dynamic methods for selecting between HAP and SP at runtime depending on workload performance characteristics, using manual analysis and a hand-tuned model [136] or expensive enumerative profiling and machine learning [80]. Both cases show that dynamic selection can improve performance by matching, and in some cases beating, the performance of a static paging choice. While effective, both methods require time consuming data collection for model construction as well as manual intervention and/or domain expertise.

In this chapter, we present a dynamic selection procedure, DSP-OFFSET, for the dynamic paging mode selection problem. We map the problem of selective paging to the contextual bandit, a model for sequential decision making under limited feedback. With a single, random profiling execution of each benchmark in the SPEC INT2006 suite, using the Binary-Offset algorithm [21], we construct an effective dynamic paging mode selection policy which is competitive with the state-of-the-art ASP-SVM [80] while requiring substantially less profiling time. Unlike previous work, our profiling requires no prior knowledge of workload structure or phasing, and does not require extensive domain expertise or manual tuning. In addition, our dynamic selection framework has the potential to be applied to other system configuration problems.

The chapter is organized as follows. In Section 3.2, we review memory virtualization and summarize work related to dynamic paging mode selection. We also describe the contextual bandit and methods

for constructing selection policies from logged random data. Section 3.3 describes our application of the contextual bandit to the dynamic paging mode selection problem. Section 3.4 presents our methodology, experimental results, and analysis. Section 3.5 summarizes our conslusions, discusses the general applicability of our method, and describes possible future research directions.

3.2 Background and Related Work

We first provide an overview of memory virtualization techniques, and describe prior work for selecting paging modes dynamically at runtime by observing workload characteristics. We then introduce the contextual bandit, which will serve as the underlying model for dynamic paging mode selection. Finally, we describe Binary-Offset and the Weighted Support Vector Machine, which we will use to construct our dynamic selection model.

3.2.1 Memory Virtualization

In virtualized systems, the virtual machine memory manager (VMMM) is responsible for mapping virtual and physical memory addresses of guest operating systems to hardware addresses. Fully virtualized systems, which do not require modifications to guests, use either Shadow Paging (SP) or Hardware-Assisted Paging (HAP) for address translation. In Shadow Paging, the VMMM maintains a shadow page table in parallel with the page table maintained by the guest. The shadow page table maps virtual addresses in the guest directly to machine addresses (V2M), bypassing the virtual to physical address translation (V2P) of the guest all together. This requires updates to the guest page table to be reflected in the shadow page table, which results in expensive virtual machine (VM) exits and context switches in order to maintain the synchronization between the two tables. In Hardware-Assisted Paging, an extended page table [54] (EPT) or nested page table [23] (NPT) is maintained by the VMMM and maps a guest's physical addresses to machine addresses (P2M). An overview of the two methods is given in Figure 3.1. Page table updates in HAP do not require synchronization and expensive VM exits; however, address translation must access both the guest page table and the extended/nested page table, resulting in more memory accesses and longer latency.

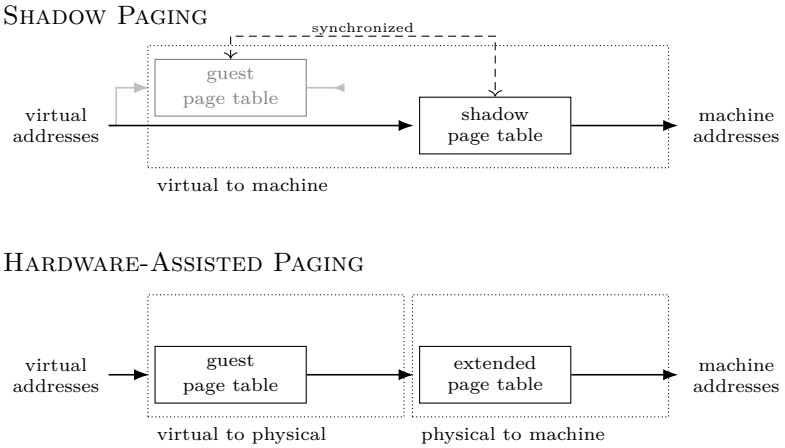


Figure 3.1: A comparison of Shadow Paging and Hardware-Assisted Paging using extended/nested page tables.

The performance of either paging mode is dependent on workload, and both SP and HAP have cases in which they are preferable [24]. Gillespie [54], Adams and Agesen [1], Wang et al. [136] characterize the advantages of SP and HAP according to workload behavior. Workloads which encounter a large number of page faults, and thus a large number of page table updates, will favor HAP, as hardware virtualization does not incur the penalty of page table synchronization. Workloads which are memory intensive will favor SP, as page walk overhead is substantially reduced. This suggests that VM exits, page faults, and translation lookaside buffer (TLB) misses are effective metrics for quantifying workload behavior with regards to memory virtualization.

To address these trade-offs, a number of dynamic paging mode selection schemes have been proposed. These methods choose to utilize hardware or software paging when appropriate based on runtime performance metrics for the current workload. Bae et al. [15] present a heuristic model for the Palacios [84] VM which selects between hardware and software paging at regular intervals according to a pair of dynamic thresholds, for VM exits and for data TLB misses. Wang et al. [136] conduct an extensive manual analysis of page fault and data TLB miss counts for workloads executed using the Xen [16] VM, and present a set of hand-crafted and system-dependent thresholds for paging mode selection. However, both of these methods involve subjective construction by domain experts.

Kuang et al. [80] describe a procedure for labeling program phases according the performance gain

associated with each paging mode, and utilize machine learning to construct a decision procedure. They enumerate over each phase of a program, comparing the performance of selecting HAP for that phase (and SP for the remaining phases) with the baseline performance of SP; similarly, they enumerate and compare SP with the baseline performance of HAP. This enumerative profiling approach is effective, but requires extensive computation. The authors suggest that the profiling required for the SPEC INT2006 [60] required over 24 hours.

3.2.2 Contextual Bandits

We model the dynamic paging mode selection problem as a contextual bandit. The contextual bandit is a method for sequential decision making in environments which provide limited feedback [12, 86, 85, 44, 21, 128]. At each iteration, a contextual bandit observes some contextual information $\vec{x} \in X$ and uses $\vec{x}$ and existing knowledge about the environment in order to select an action $a \in A$. In response to taking action a, the bandit receives a reward r dependent on both the taken action and the associated context; the rewards for actions not taken remain unobserved. This is referred to as bandit feedback. The goal of the bandit is to learn some policy for action selection which maximizes the cumulative reward earned by the learner.

Classic approaches to the contextual bandit are online and dynamically adjust the selection of actions to adapt to both the estimates of each action's reward and the confidence of those reward estimates [12, 85]. These methods are said to balance exploration, selecting an action to improve the estimate of its reward, and exploitation, selecting the action believed to be optimal. However, these methods are generally not amenable to low-level implementation, e.g., in the Xen virtual machine manager, because they require expensive numerical optimization, linear algebra, and statistical procedures.

Alternatively, offline evaluation and construction for contextual bandits can be performed using logged data [86, 44, 21, 128]. Here, exploration and exploitation are not interleaved; rather, exploration occurs for a fixed duration during a training phase, and the resulting logged data is used to construct a policy which is then exploited. The logged data can be obtained by selecting actions uniformly at random, or from carefully constructed deterministic action selections. These methods

Algorithm 1 Binary-Offset [21]

Input set of contextual bandit instances $S = \{(\vec{x}, a, r)\}$
$S' = \varnothing$
for each $(\vec{x}, a, r) \in S$ **do**
  $(\vec{x}, y, w) = (\vec{x}, \text{sign}\,(a \cdot r), |r|)$
  $S' = S' \cup (\vec{x}, y, w)$
end for
return weighted classification instances S'

are also referred to as exploration scavenging [86], as they attempt to utilize the logged data gained from executing some other policy as a form of exploration.

Here, we focus on the Binary-Offset algorithm [21], given in Algorithm 1, which requires binary actions $A = \{-1, +1\}$. Binary-Offset is a method for transforming contextual bandit data $(\vec{x}, a, r)$ obtained from a random policy into weighted data $(\vec{x}, y, W)$, where the class y represents an estimate of the better performing action and the weight W represents the degree to which that action improves from the baseline. For paging mode selection, the context could take the form of relevant performance metrics measured over a sampling period and the action would indicate whether SP or HAP should be selected for the subsequent period. Workload throughput or speedup could both be considered as useful reward metrics.

The resulting weighted classification instances are amenable to a broad suite of machine learning techniques for feature selection, dimension reduction, and classifier construction. Classifiers which directly incorporate instance weights exist in the literature [145, 46, 50, 104]. Alternatively, using the 'Costing' method [148], weighted classification instances can be sampled in proportion to their weight in order to construct a standard, unweighted labeled data set.

We use the Weighted Support Vector Machine (WSVM) [145] to construct a dynamic selection model from the weighted classification instances generated by Binary-Offset. For a set of n weighted instances of the form $(\vec{x}_i, y_i, W_i)$, the (linear) WSVM attempts to find the classifier

$$f(\vec{x}) = \text{sign}(\vec{w} \cdot \vec{x} + b) \tag{3.1}$$

with the largest margin separating the positive and negative instances. This can be found using the

constrained optimization problem

$$\begin{aligned} \underset{\vec{w}, b, \xi_i}{\arg\min} \quad & \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{N} W_i \xi_i \\ \text{subject to} \quad & y_i \; f(\vec{x}) \geq 1 - \xi_i, , \\ & \xi_i \geq 0, \end{aligned} \tag{3.2}$$

where C is a hyper-parameter indicating the relative importance of the margin size and the weighted misclassification error. This resulting linear classifier is simple to implement in a virtual machine manager.

3.3 Dynamic Paging Mode Selection

We formulate dynamic paging mode selection as a contextual bandit, wherein the virtual machine monitor selects between Shadow Paging (SP) and Hardware-Assisted Paging (HAP) at regular intervals depending on relevant performance metrics in order to optimize workload performance. The contextual information will take the form of page fault and data translation lookaside buffer (DTLB) miss counts, as they characterize the performance of the two paging modes. The action space contains both SP and HAP. The reward will be a measure of workload performance, based on the number of instructions retired per cycle count (IPC) over an observation interval, for the selected paging mode.

Here we present two methods. The first is a simple, context-less bandit model, DSP-SAMPLE, which selects paging modes by comparing the IPC of HAP and SP directly at runtime without taking advantage of page faults, DTLB misses, or any other performance metrics. The second is a contextual bandit model, DSP-OFFSET, which exploits both page fault and DTLB miss counts in order to select paging modes which provide a speedup compared to a random baseline. However, unlike DSP-SAMPLE, DSP-OFFSET requires offline profiling and training.

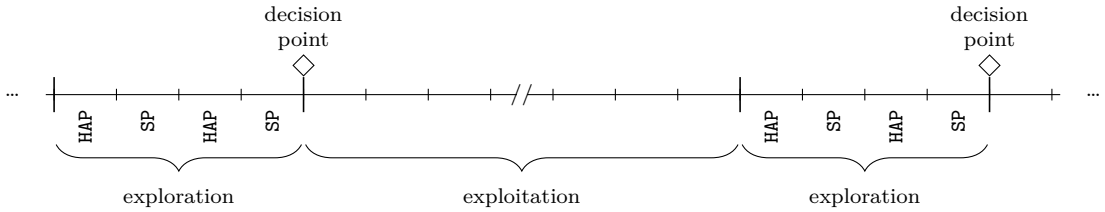


Figure 3.2: Design and parameters of DSP-SAMPLE.

3.3.1 Direct Sampling (DSP-SAMPLE)

DSP-SAMPLE is a simple, direct sampling approach which operates in two stages. The first stage alternates between selecting HAP and SP several times in order to discover which paging mode provides the highest IPC. The second stage selects the paging mode which was found to provide the best performance on average and utilizes that paging mode for a time. The two stages alternate, timed appropriately to balance constructing a confident estimate of performance, utilizing the best identified paging mode, and adapting to changing workload characteristics. This can be described as a method which balances exploration (sampling the performance of each paging mode), and exploitation (utilizing the best performing paging mode) — similar to the contextual bandit, but without contextual information. A similar model is used in Jiménez et al. [70] for dynamic hardware memory prefetcher utilization.

The design of DSP-SAMPLE is summarized in Figure 3.2. This method is parameterized by the length of the observation interval, as well as the number of intervals in both the exploration and exploitation periods. A longer exploration period can provide a better estimate of performance, which can lead to fewer poor exploitation period selections. However, a longer exploration period will also incur more overhead from paging mode switching. A longer exploitation period will reduce the frequency of exploration, but shifting workload characteristics can cause the selected paging mode to no longer be desirable. Parameter tuning is required for DSP-SAMPLE to be effective.

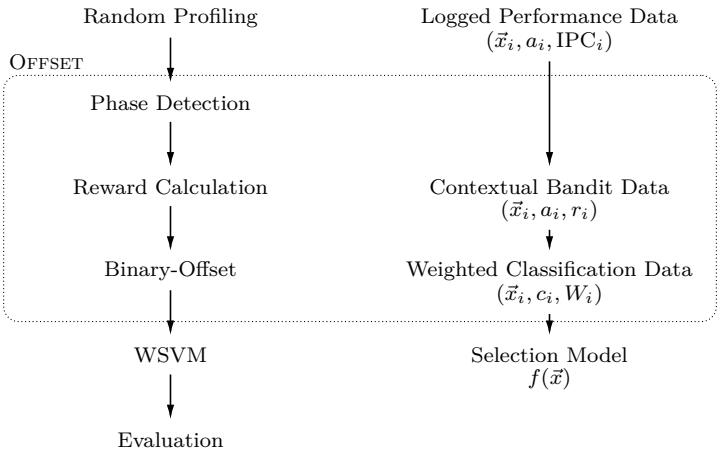


Figure 3.3: Overview of the Binary-Offset model construction and Binary-Offset model evaluation workflows for paging mode selection and the associated data transformations.

3.3.2 Contextual Bandit Model (DSP-OFFSET)

To construct the DSP-OFFSET model, we first must obtain logged data from random paging mode selections for workloads of interest. Next, the logged data must be converted into a form which is usable to Binary-Offset. This includes identifying phasing structure and defining a useful reward function. Finally, we transform, via Binary-Offset, the logged data into weighted data and use the WSVM in order to construct the DSP-OFFSET model. This construction is illustrated in Figure 3.3.

As with previous work [136, 80, 1], we rely on page faults and DTLB miss counts to characterize the relative performance of HAP and SP. The frequency of DTLB misses is correlated with the frequency of page walks, and the frequency of page faults is correlated with page table updates; therefore, we expect HAP to outperform SP during periods of frequent page faults and SP to outperform HAP during periods of frequent DTLB misses. However, effective switching requires determining the trade-off for workloads with mixed characteristics. As page faults and DTLB misses characterize the relative performance of HAP and SP, we assume that the relative performance of the two paging modes otherwise remains unchanged by other, unobserved performance characteristics, as well as from the historical behavior of both page faults and DTLB misses. As we find that the distribution of both page fault and DTLB miss counts over fixed sampling intervals are heavy tailed, we consider

the binary logarithm of both counts instead of using the counts directly. This has the effect of leveling out the distribution of each metric and reducing the effect of outlier behavior.

Training data is obtained from workloads by executing a random paging mode policy. At regular sampling intervals, Xen measures relevant performance metrics, including page faults, DTLB misses, and IPC, and selects HAP or SP uniformly at random for use during the next sampling interval. If the system is already using the selected paging mode, no change happens. Otherwise, the system switches to the new paging mode, incurring the associated cost. We associate the performance characteristics used to make a selection (page fault and DTLB miss counts over the interval which just ended) with the performance resulting from that selection (IPC of the following interval).

The logged training data must now be transformed into contextual bandit data, i.e., context, action, and reward. We consider the speedup of a paging mode selection compared to average workload performance as a reward. However, many applications exhibit phasing behavior [113, 116, 118]. Shifting performance characteristics, either between workloads or between phases of a workload, can skew the weighting of our instances toward certain phases. Both `milc` and `xalancbmk` from the SPEC CPU2006 [60] suite skew our results if we consider IPC as a reward directly, as both contain small phases of high IPC that would be more strongly weighted towards despite providing little opportunity for improved performance. Any possible imbalance between HAP and SP due to random sampling during these phases can amplify the effect.

To account for these extraneous effects, we consider phases of the logged performance data. Using the change-point detection algorithm PELT [78], we partition each random profiling execution into a set of phases based on the sequence of IPC values. PELT is an efficient dynamic programming algorithm for identifying changes in the distribution of a time series, such as identifying changes to the mean and variance of a workload's IPC over time. PELT optimizes the number and position of the change points given an information criterion penalty. Given a set of change-points c_j, we segment our training data into phases $[c_j, c_{j+1}]$. These phases simply represent periods of consistent workload performance. An alternative would be to specify these phases manually, however we find that PELT is sufficient for identifying meaningful periods and does not rely on domain expertise.

Reward Functions

The reward is calculated based on the logarithmic speedup of each instance's performance (IPC) against the average performance of the phase containing that instance. Instances which cause a speedup in comparison to random are given a positive reward. Instances with no speedup or slowdown compared to the average performance of the random selections have a zero reward as they represent the baseline behavior. For Binary-Offset, instances which cause a slowdown compared to random should be treated as instances of the opposite paging mode with the reciprocal speedup. Therefore, for a instance $i \in [c_j, c_{j+1}]$, we calculate the reward as

$$r_i = \log \frac{IPC_i}{\overline{IPC}_{[c_j, c_{j+1}]}}, \tag{3.3}$$

where $\overline{IPC}_{[c_j, c_{j+1}]}$ is the average IPC for the phase containing instance i. Measuring speedup (per phase) avoids the problem of high IPC phases having a stronger weighting, as the weighting is now relative to the average performance of the phase. Figure 3.4 (top and middle) illustrates the transformation from IPC to reward.

Learning Methods

Using the contextual information $\vec{x}_i$ (page fault and DTLB miss counts), actions a_i (SP and HAP, mapped to +1 and -1 respectively), and rewards r_i calculated according to Equation 3.3, we transform the contextual bandit data $(\vec{x}_i, a_i, r_i)$ into weighted data $(\vec{x}, y_i, W_i)$ using Algorithm 1. This transformation is illustrated in Figure 3.4 (middle and bottom). The weighted data describes, for some set of performance metrics $\vec{x}_i$, which paging mode y_i is expected to provide a speedup and how strongly it is expected, i.e., the weight W_i. We apply a linear WSVM (Equation 3.2) to the weighted instance data in order to construct a linear decision function which maps page fault and DTLB miss measurements to a paging mode selection. Other algorithms (e.g., weighted logistic regression, weighted sampling [148]) were considered but WSVM provided the best performance.

To prevent rapid switching between SP and HAP, a potential source of performance loss due to the switching overhead, we define a margin around the decision function. Any workload which is

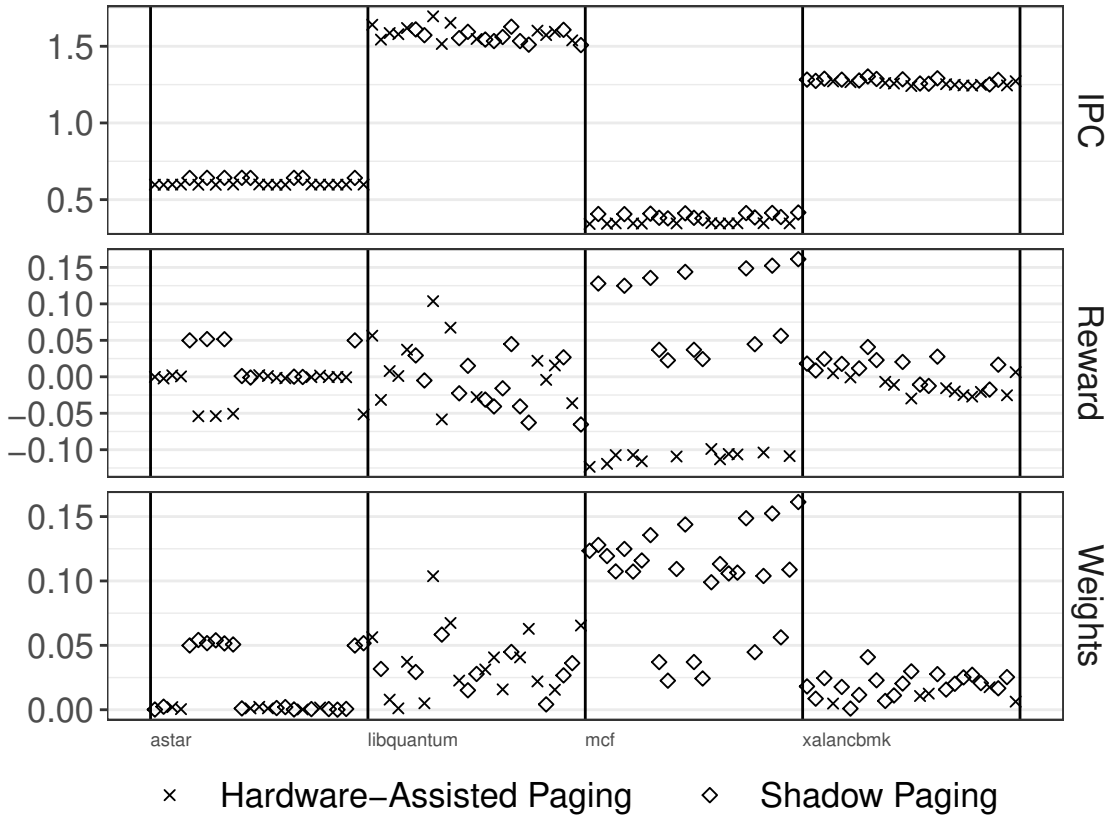


Figure 3.4: IPC to instance weight transformation: Top; traces of IPC and paging mode using a random selection policy for a subset of select workloads. Middle; IPC transformed to reward. Bottom; Binary-Offset transformation to weights.

Table 3.1: Hardware Configuration

CPU	Memory	Cache			DTLB	
		L1	L2	L3	L1	L2
2.8 GHz	4 GB	64 KB 4-way	512 KB 8-way	8192 KB 16-way	64 entries 4-way	512 entries 4-way

operating inside of the margin does not trigger a switch, as we assume that the potential performance advantage will not outweigh the cost of switching. We find that a quarter of the WSVM margin results in good performance:

$$\vec{w} \cdot \vec{x} + b > +0.25\text{: if necessary, switch to SP,}$$
$$\vec{w} \cdot \vec{x} + b < -0.25\text{: if necessary, switch to HAP.}$$

Thrashing behavior which occurs because a workload alternates between two extremes, and thus alternates outside of the margin, would not be prevented. However, this does not happen in practice for the workloads we investigated.

3.4 Evaluation

This section describes our experimental methodology and presents our results. We evaluate the performance of both DSP-SAMPLE and DSP-OFFSET, and compare both models against the state-of-the-art ASP-SVM [80]. To conclude, we discuss the advantages of DSP-OFFSET with respect to profiling cost (Section 3.4.4).

3.4.1 Experimental Environment

Experiments are conducted on a 1st generation Intel Core i5 processor (Nehalem microarchitecture), running at 2.8 GHz, with Intel Turbo Boost and other adaptive clock cycle technology disabled. The hardware configuration is summarized in Table 3.1. A 64-bit host OS running Linux 2.6.18 (CentOS 5.4) is configured to run a modified version of Xen 3.3.1 which implements the paging mode selection mechanism for the Xen hypervisor as described in [136]. A 32-bit guest OS, also running Linux

2.6.18 (CentOS 5.4), is provided with 3 GB of memory and is constrained to a single core, for which it has sole affinity. Policies are evaluated using the SPEC CPU2006 [60] benchmark suite as the benchmarks show a variety of memory behavior. The benchmarks are compiled for the guest OS using GCC 4.1.

At regular intervals, Xen measures relevant performance metrics, including page faults and DTLB misses, and identifies if the system should utilize SP or HAP for the following interval according to the current policy. To measure page faults, a kernel module in the guest OS notifies the Xen hypervisor of a shared memory address in which the guest OS records the page fault count. To measure DTLB misses, instructions retired, and clock cycles, the Xen hypervisor configures and accesses the Performance Monitoring Unit [66] directly. A programmable counter is configured to measure DTLB misses (mnemonic `DTLB_MISSES.WALK_COMPLETED`) and IPC is measured using the fixed-function counters for retired instructions and core clock cycles. The page fault and DTLB miss counts are transformed using a simple fixed-point arithmetic binary logarithm; alternatively, these features could be approximated by identifying the number of leading zeros in the counts.

3.4.2 Experimental Design

We evaluate DSP-SAMPLE with a sampling rate (observation interval length) of 100 ms. For the exploration period, the algorithm measures the IPC of SP and HAP three times each (for a total of 0.6 s), and then selects the better performing paging mode to exploit for 50 observation intervals (for a total of 5 s). This is approximately a 1:10 exploration to exploitation ratio. We also attempted other possible parameter settings, but found no particular setting which was effective in all cases.

We evaluate DSP-OFFSET for both a benchmark-specific and benchmark-agnostic setting. In the benchmark-specific case, we train a DSP-OFFSET model for each benchmark using a single random profiling execution from that benchmark. Each model is then evaluated on the benchmark for which it was trained. This evaluates the performance of DSP-OFFSET when constructed on a wide range of training data sizes with varying workload characteristics. In the benchmark-agnostic case, we construct a single DSP-OFFSET model by aggregating data from the SPEC INT2006 benchmarks. This evaluates the effectiveness of DSP-OFFSET to model a broad range of workload characteristics

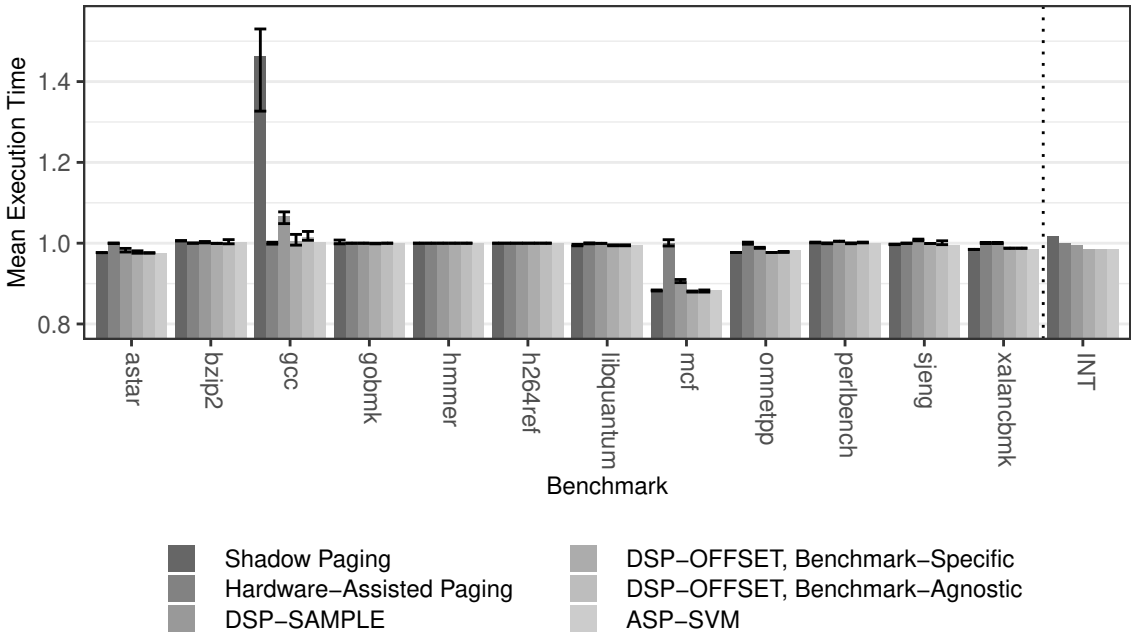


Figure 3.5: Mean normalized execution time for Hardware-Assisted Paging, Shadow Paging, and dynamic selections including DSP-SAMPLE, DSP-OFFSET (benchmark-specific, benchmark-agnostic), and ASP-SVM [80] on SPEC INT2006. Error bars indicate minimum and maximum normalized times.

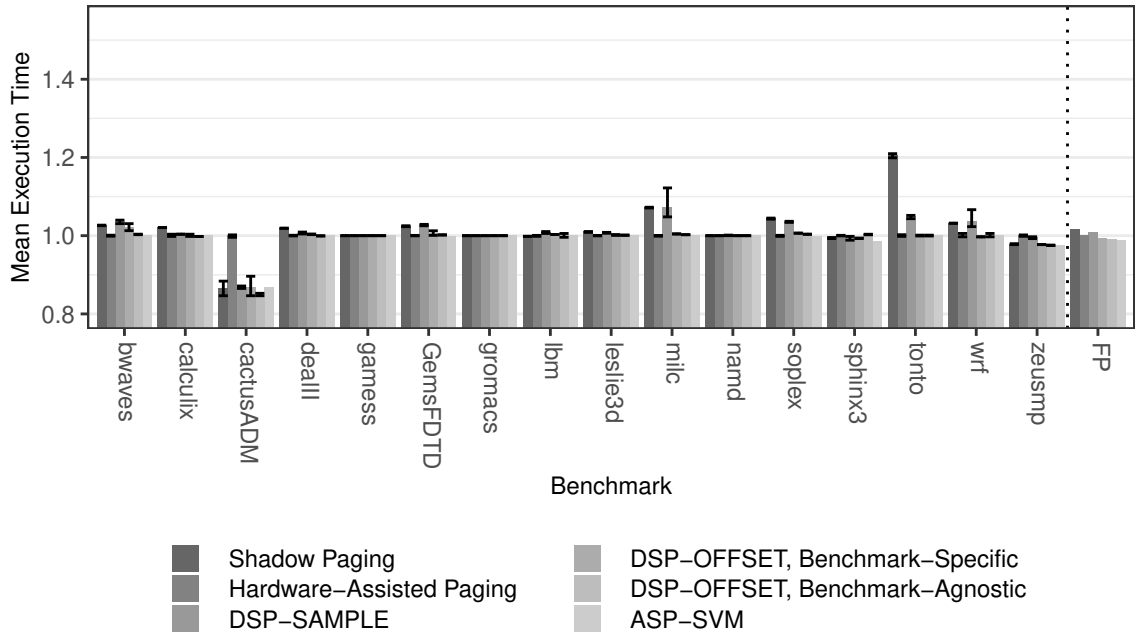


Figure 3.6: Mean normalized execution time for Hardware-Assisted Paging, Shadow Paging, and dynamic selections including DSP-SAMPLE, DSP-OFFSET (benchmark-specific, benchmark-agnostic), and ASP-SVM [80] on SPEC FP2006. Error bars indicate minimum and maximum normalized times.

and to generalize to other workloads not included as part of the training data. In both cases, we use a sampling period of 1 s for both random profiling and evaluation, and we select the hyper-parameter C for the WSVM (Equation 3.2) using a simple grid search. We considered sampling periods of 2 s, 1 s, and 100 ms and found that the differences in the resulting policies and performance were small.

3.4.3 Results

Figures 3.5 and 3.6 summarize the mean execution times of the static HAP and SP policies and the dynamic DSP-SAMPLE, DSP-OFFSET, and ASP-SVM policies, normalized to the mean execution time of HAP, for the SPEC INT2006 and FP2006 benchmark suites, respectively. For HAP, SP, DSP-SAMPLE, and DSP-OFFSET, we report the min, mean, and max ratios of three runs. For ASP-SVM, we report the mean of five runs. The results for `povray` are omitted for ASP-SVM, as they were not reported in [80].

Several benchmarks have notable differences in performance between SP and HAP: `cactusADM` and `mcf` favor SP (13%, 12% gain); `gcc` and `tonto` favor HAP (46%, 21% loss with SP, respectively). On average, SP presents a performance loss of 1.6% compared to HAP (1.6% for SPEC INT2006 and 1.5% for FP2006), and many benchmarks show no difference in performance between the two static policies.

Direct Sampling

DSP-SAMPLE presents an overall performance loss of 0.2% compared to HAP (0.5% gain for SPEC INT2006 and 0.7% loss for PF2006). While the performance of DSP-SAMPLE can be similar to the performance of the best static policy, as is the case for `gcc`, `tonto`, `cactusADM`, and `mcf`, there are some cases for which the performance of the dynamic procedure is no better than the worst static policy. For `bwaves`, `milc`, and `wrf`, DSP-SAMPLE has roughly an equivalent average performance loss to SP (3.5%, 7.3%, 3.8% loss, respectively) and for `milc` and `wrf` there is significant variability in the performance across multiple runs. The performance of DSP-SAMPLE may be tailored, through careful parameter selection, to better suit certain types of workloads. However, this can in turn negatively affect other workloads.

Benchmark-Specific Models

As DSP-OFFSET utilizes the contextual information (performance metrics) available, we anticipate that each benchmark-specific model should provide effective performance on the workload in which it was trained. For nearly all benchmarks, the performance of the benchmark-specific DSP-OFFSET model constructed from a single random profiling execution matches the performance of the best static policy. The notable exception to the favorable performance of DSP-OFFSET is `bwaves`, which performs 2.1% worse than the static HAP policy but on average better than the static SP policy. On average, the DSP-OFFSET models present a 1.1% performance gain (1.6% for SPEC INT2006 and 0.8% for SPEC FP2006).

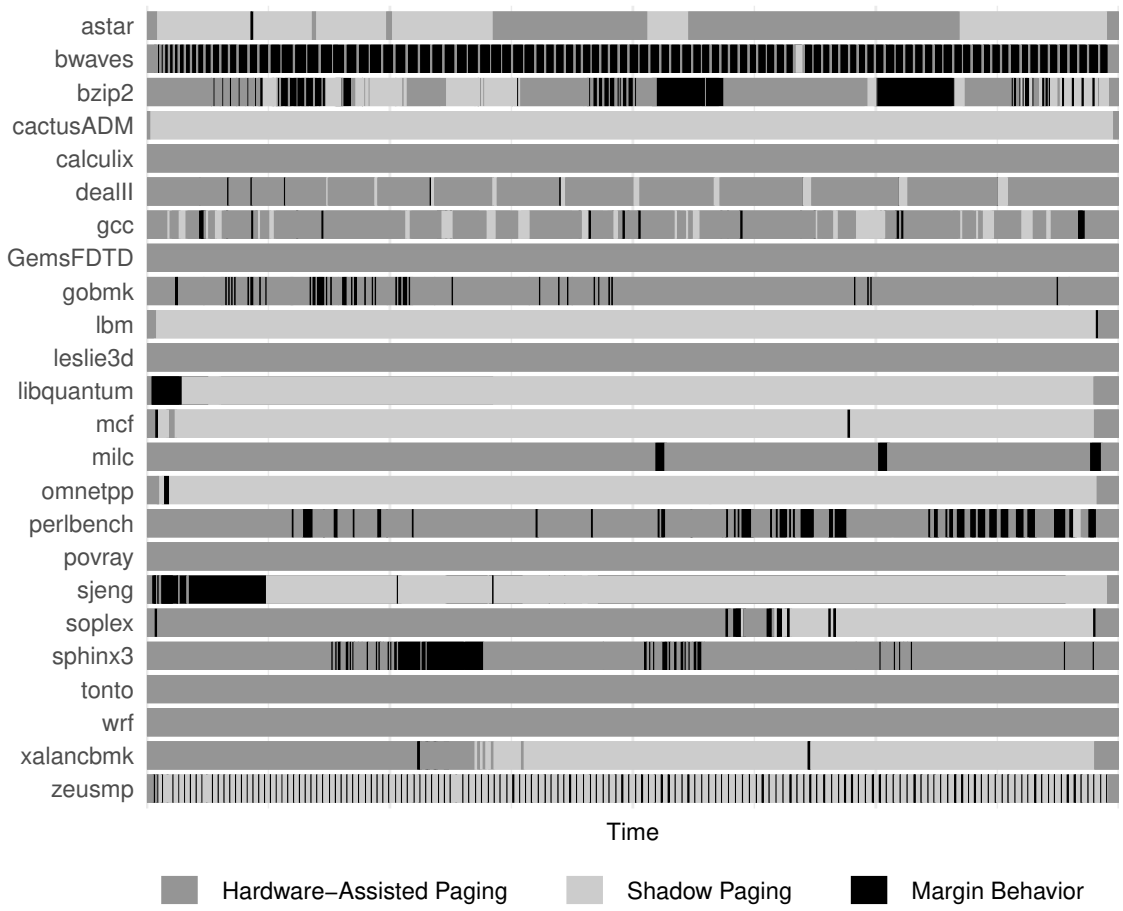


Figure 3.7: Paging modes selected over time for SPEC CPU06 benchmarks using the benchmark-agnostic DSP-OFFSET constructed on SPEC INT06.

Benchmark-Agnostic Model

Whereas in the benchmark-specific case we constructed separate models for each benchmark, here we construct a single benchmark-agnostic model for the full suite. The benchmark-agnostic DSP-OFFSET model presents a 1.2% performance gain compared to HAP (1.4% for SPEC INT2006 and 1.0% for FP2006). In comparison, ASP-SVM presents a 1.3% performance gain (1.6% for SPEC INT2006 and 1.1% for FP2006). Overall, both DSP-OFFSET and ASP-SVM have similar performance gains over the static policies. Again, we stress that the aggregate data used to train DSP-OFFSET contains only a single random execution for each integer benchmark.

The paging mode selections for the benchmark-agnostic DSP-OFFSET model are summarized in Figure 3.7, including periods in which the model would have triggered a switch but did not due to the margin. Workloads for most benchmarks cause a single paging mode to be selected almost always during the course of the benchmark's execution. For benchmarks which execute primarily in SP, we observed periods at the beginning and end of the profiling run in which HAP was utilized. These periods coincide with the initialization and tear-down of the SPEC tools as well as with the start and end of program execution. A larger than average number of page faults are to be expected during these periods, and thus these periods would favor HAP as hardware paging avoids the cost of page table synchronization. Margin behavior only affects `bwaves` and `zeusmp`. For `zeusmp`, the margin prevents thrashing behavior that would otherwise cause the model to switch between HAP and SP every two or three seconds. For `bwaves`, we observe that the benchmark's workload is predominantly inside of the margin.

3.4.4 Profiling Cost

Collecting training data using random selection is no more expensive than running the benchmarks using the worst of their static paging modes. Moreover, a single random evaluation for each benchmark is sufficient to obtain performance equivalent to ASP-SVM. In contrast, ASP-SVM requires an average of six separate executions of each benchmark in the collection of the training data. The reported data collection time for ASP-SVM was over 24 hours; in comparison, random profiling for SPEC INT2006 requires less than 2.5 hours for DSP-OFFSET, and the full SPEC CPU2006 suite requires less than 6.5 hours.

While our profiling time is reduced in comparison to ASP-SVM, the dataset for DSP-OFFSET is several orders of magnitude larger. For DSP-OFFSET, with a 1 s sampling period, there are approximately 25000 data samples across the twelve integer benchmark executions (one data sample per sampling period); for ASP-SVM there are 60–67 samples. This is noisy data, both due to variable workload characteristics as well as the random selection of paging modes. There are periods of a benchmark's execution which will be under-sampled. In some cases, random selection may also lead to periods where one paging mode is sampled almost always. This leads to outliers in the contextual

measurements (page fault and DTLB miss counts) as well as in the labels and weights we eventually generate using Binary-Offset. The enumerative profiling approach taken in [80] encodes knowledge and assumptions regarding workload structure in order to address this noise/variation, which is the source of their profiling cost. We instead compute this structure after the fact using the random logged data.

3.5 Discussion and Conclusion

In this chapter, we present DSP-OFFSET, an effective procedure for dynamic paging mode selection which utilizes a simple, random profiling method. Dynamic paging mode selection policies are capable of balancing the trade-off between Hardware-Assisted Paging and Shadow Paging at runtime by dynamically switching the paging mode at runtime according to performance metrics. We evaluate our approach on the SPEC CPU2006 benchmark suite and compare our approach with an existing machine learning method. DSP-OFFSET achieves speedups up to 44% compared to static paging mode selections and matches state-of-the-art performance. In addition, our method requires substantially less profiling, an 90% reduction in profiling time.

While we chose to apply our method specifically to paging mode selection, the framework we present is generally applicable to a range of dynamic configuration problems for computer systems. One particular example is that of hardware prefetching. Modern Intel systems are equipped with four hardware prefetchers which can be enabled or disabled at runtime [132]. IBM POWER7 systems are equipped with a highly configurable prefetch engine that allows prefetchers to be parameterized (e.g., prefetching depth and stride) [121]. Liao et al. [88], Rahman et al. [105] propose prefetcher configuration recommendation methods; however, these are static, and not dynamic approaches. A single, fixed configuration is selected for a given program after a window of profiling. Jiménez et al. [70] propose a direct sampling method, similar to the DSP-SAMPLE approach given in Section 3.3.1, without using contextual information to guide their selection.

Paging mode selection can be seen as a small and well understood instance of a dynamic configuration problem. Performance can be described by a small number of features identified by domain

knowledge (page faults, DTLB misses), with only two configurations (Hardware-Assisted Paging, Shadow Paging). Hardware prefetching is an interesting application as it presents the challenge of larger action sets (16 in total for Intel systems) and action sets which are combinatorial in nature (4 independent hardware prefetchers). The Binary-Offset method can be expanded into an Offset-Tree [21], providing for larger action spaces. Hardware prefetching can also present the opportunity to expand the contextual information used to include additional performance metrics (our framework has no explicit limit on the number of attributes).

While our application of Binary-Offset substantially reduces profiling time for training, validation of the resulting dynamic selection procedures still requires execution of the model in situ. Methods for evaluating deterministic policies, using random or deterministic data, are available for the contextual bandit [86, 44], and may be amenable to the problem setting. We hope to apply these methods in order to provide offline evaluation, in addition to offline model construction.

Finally, we note that the application of Binary-Offset still required careful attention in order to address problems such as label noise. Standard convex-loss methods are sensitive to label outliers in the data [144, 99]. We hope to investigate the use of more robust machine learning methods which are capable of addressing this problem.

Chapter 4

Hardware Memory Prefetcher Utilization

Modern architectures provide hardware memory prefetching capabilities which can be configured at runtime. On Intel microarchitectures, this control takes the form of four hardware prefetchers which can be enabled or disabled independently of one another, independently on each core. While hardware prefetching can provide substantial performance improvements for many programs, prefetching can also increase contention for shared resources such as last-level cache and memory bandwidth. The interaction in memory resource utilization across multiple cores, and the corresponding cross-core contention, can degrade system-wide performance in multi-core workloads.

This chapter considers an application of the contextual bandit framework to dynamic hardware memory prefetcher utilization. Hardware prefetcher control introduces a potential combinatorial growth in action space size compared to paging mode selection—the binary state of each hardware prefetcher on each core can be controlled independently. System-wide prefetcher control is achieved by utilizing a set of identical and cooperating binary-action controllers each operating one prefetcher on one core. The controllers are presented with cross-core workload behavior statistics, and the reward for each hardware prefetcher configuration accounts for both local and cross-core effects on system-wide performance. The Binary-Offset algorithm [21] and random profiling are again presented as effective techniques for constructing dynamic selection models. In addition, Smooth 0-1 Loss Approximation (SLA) [99] is introduced to address the challenges of label noise in the resulting data.

The material contained in this chapter was previously published in the Proceedings of the 48th International Conference on Parallel Processing (ICPP '19) [62].

4.1 Introduction

Hardware memory prefetching is an effective way to ameliorate memory latency. While hardware prefetching is productive for single-threaded programs, added memory requests due to prefetching can pollute or saturate shared resources such as the last-level cache and memory bandwidth. On multi-core workloads, the increase in resource contention can cause hardware prefetching to be contraindicated. Modern microprocessors expose runtime controls of certain hardware prefetchers. These runtime controls offer users and administrators the opportunity to tailor prefetcher usage to workload behavior.

Examples of existing frameworks for runtime hardware prefetcher control include course-grained recommendation systems and fine-grained enumerative sampling. In the former, workload characteristics are mapped to a prefetcher configuration which is utilized over the course of a full workload execution [88, 105]. In the later, prefetcher configurations are sampled regularly and the best performing configuration is exploited for a period of time [70].

In this chapter, we describe a method for learning per-core hardware prefetcher control policies which provide prefetcher configuration recommendations for multi-tenant workloads on a fine-grained scale. These dynamic control policies are reflexive, in that a policy responds directly to the performance characteristics of the currently executing workload in order to select a prefetching configuration. We construct policies for systems with a range of memory characteristics in order to verify the efficacy of our approach. Our models outperform a typical baseline, which leaves all prefetchers enabled system-wide, by up to 4.3% on average for a system with limited memory bandwidth. By utilizing workload-specific policies, tailored to the performance characteristics of individual workloads, our models outperform the same baseline by up to 5.1% on average for the same system.

The chapter is organized as follows. Section 4.2 reviews hardware memory prefetching and summarizes work related to software control of hardware prefetchers. Section 4.3 describes the contextual bandit model, and formalizes the application of the contextual bandit to the problem of prefetcher control. Sections 4.4 and 4.5 discuss our methodology, experimental results, and analysis. Section 4.6 summarizes the framework, describes our conclusions and motivates future work.

4.2 Background and Related Work

Modern microprocessors utilize hardware memory prefetching in order to reduce memory access latency. Hardware prefetching predicts cache lines which are likely needed by the processor in the near future, and fetches that data into the cache early. While prefetching can be effective in reducing memory latency, it can be contraindicated in multi-core systems due to the increased contention for shared memory resources, e.g., last-level cache and off-chip memory bandwidth [75]. Prefetching misprediction further pollutes the shared cache and wastes memory bandwidth.

Hardware prefetchers can often be configurable in software. Intel microarchitectures are equipped with four independent and configurable hardware prefetchers which can be enabled or disabled on a per-core basis [132]. Prefetcher configuration is controlled by the first four bits (bits 0–3) of Model-Specific Register (MSR) `0x1A4` on each core. Each bit controls the state (enabled or disabled) of the four exposed prefetchers: (0) *Data Prefetch Logic* (DPL), which detect streaming requests and fetches streams of instructions and data from memory to the L2 cache; (1) *Adjacent Cache Line* (ACL), which fetches a paired cache line to form a 128-byte aligned chunk; (2) *Data Cache Unit* (DCU), which attempts to recognize streaming access due to multiple loads from the same cache line and will fetch the next cache line into the L1 data cache; and (3) *Instruction Pointer* (DCU IP), which attempts to detect stride accesses in a fixed memory window for L1 data cache prefetching [88, 105, 66, 132]. Both the DPL and APL prefetchers are associated with the L2 cache, whereas the DCU and DCU IP prefetchers are associated with the L1 cache. Likewise, IBM Power systems feature a hardware prefetching engine which supports rich software configuration support [121].

In addition to hardware-based solutions [123, 45], a number of approaches have been proposed to mitigate the destructive effects of hardware prefetching in multi-core workloads through software control of hardware prefetchers. Liao et al. [88], Rahman et al. [105] construct recommendation systems for hardware prefetcher configurations. Both methods utilize an understanding of workload characteristics, and relate those workload characteristics using machine learning to a static configuration which is likely to perform best for that workload on future executions. Rahman et al.

[105] utilize a feature extraction technique in order to identify relevant performance measures for hardware prefetcher recommendations; Liao et al. [88], on the other hand, utilize domain expertise to identify these measures.

In contrast to static prefetcher configuration recommendations, Jiménez et al. [70] describe a method for dynamic prefetcher configuration by periodically sampling workload performance directly for a variety of different configurations and exploiting the best performing configuration for a period of time. Their method is applied to IBM POWER7 systems, which are equipped with a highly configurable prefetch engine that allows prefetchers to be parameterized (e.g., for prefetching depth and stride) [121]. As sampling the full set of configurations is not generally feasible for the POWER7 prefetching engine, the authors identify a subset of configurations that cover a broad range of workload behavior characteristics. Jiménez et al. [69] present a similar dynamic scheme for the POWER7 system by directly incorporating memory bandwidth measurement, adjusting the aggressiveness of prefetching on cores which inefficiently utilize a significant amount of added bandwidth due to prefetching compared to the corresponding increase in performance.

Ortega et al. [101] exploit runtime systems for shared memory programming models in order to directly configure hardware prefetchers for software-defined parallel regions. Similarly to Jiménez et al. [70], Jiménez et al. [69], this approach samples hardware prefetcher performance and exploits the configuration with the best performance. However, instead of an uninformed approach which polls the system periodically, this approach is informed by software constructs.

4.3 Contextual Bandit Framework

In this work, we consider the problem of constructing decision policies for fine-grained hardware prefetcher control. At short, regular intervals, a controller observes system behavior (using performance monitoring). In response to the performance characteristics of the current workload, the controller then decides according to some policy function which hardware prefetcher configuration to use during the following interval. An effective policy will tailor the use and aggressiveness of prefetching to the workload, disabling prefetching on one or more cores when the added resource

Table 4.1: Performance Monitoring Events for Contextual Information

Mnemonic	Architectures	Description [66]
L1D:ALLOCATED_IN_M	SB	allocations of modified L1D cache lines
L1D:M_EVICT	SB	modified lines evicted from the L1 data cache due to replacement
L1D:REPLACEMENT	KL BW	lines brought into the L1 data cache
L2_LINES_IN:ANY	SB KL BW	lines allocated in the L2 cache
LD_BLOCKS:STORE_FORWARD	KL BW	loads blocked by store buffer overlapping that cannot be forwarded
LD_BLOCKS:NO_SR	KL BW	split load operations blocked temporarily due to all resources for handling the split accesses being in use
LD_BLOCKS:ALL_BLOCK	SB	loads blocked (without DCU miss)
DTLB_LOAD_MISSES:WALK_COMPLETED	SB KL BW	miss count in all translation lookaside buffer levels which results in a completed page walk of any page size
L3_LAT_CACHE:MISS	SB KL BW	cache miss condition count for references to the last level cache
OFFCORE_REQUESTS:DEMAND_DATA_RD	SB KL BW	demand data read requests set to uncore
BR_MISP_RETIRED:ALL_BRANCHES	SB KL BW	mispredicted branch instructions at retirement

contention is destructive. We relate the problem of fine-grained control to the contextual bandit, and describe a method for learning decision control policies from the contextual bandit model.

The contextual bandit [12, 85] is a form of sequential decision process. At each time step t, an agent is presented with a *context* $\vec{x}_t$ describing the state of the world. The agent selects an *action* a_t, given the context, according to some policy function $\pi(\vec{x}_t)$. Then, the agent receives a (potentially stochastic) *reward* r_t as feedback for taking action a_t with context $\vec{x}_t$. It is important to note that the agent is limited to the feedback for the selected action ("bandit" feedback) and the rewards for actions not taken are not revealed. The goal of the bandit model is to maximize the total reward over a sequence of interactions.

The contextual bandit is typically evaluated in sequence, with the agent selecting an action and then directly refining the policy function according to the feedback. Alternatively, the policy function can also be learned using log data [21, 86]. Log data is composed of context-action-reward tuples $(\vec{x}_t, a_t, r_t)$ generated on past decisions according to some fixed, known action selection policy (e.g., uniformly random selection). The advantage of using log data is two-fold: the availability of log

data is often ubiquitous, and non-adaptive methods built and evaluated offline may be preferable to adaptive methods on production systems.

To effectively apply contextual bandit methods, one needs to identify the set of actions, select the relevant contextual information, and construct a reward function. Additionally, with the use of log data we must select a classifier with which to learn a decision policy.

4.3.1 Action Selection

Across N cores, there are a total of 2^{4N} possible system-wide prefetcher configurations on Intel microarchitectures. Naively, each system-wide prefetcher configuration could represent a distinct action, however, the exponential growth limits the application of this method to many-core systems. In order to reduce the complexity of system-wide configuration, we instead consider per-core and per-prefetcher decisions independently. Per-core control of prefetchers can be myopic compared to coordinated, system-wide control over all cores [45]. However, the inclusion of system-wide metrics as part of the contextual information will allow for indirect cooperation between cores and can mitigate potential performance loss due to decoupled decision making.

4.3.2 Context Selection

We utilize the Performance Monitoring Unit (PMU) [66] to obtain contextual information relevant to prefetching performance, including translation look-aside buffer, cache, memory, and branch predictor behavior. Selected events, detailed in Table 4.1, are drawn from domain knowledge and used to good effect in related work [88, 138]. As performance events can vary across different architectures, we include similar events where available. Each event is measured on each core independently while several programs, each isolated to a core, are executed. These measurements form the basis of the contextual information. In addition to per-core measurements, we provide additional contextual information measuring off-core behavior. Off-core measurements include aggregate measures of each performance event across all other cores as a measure of global system behavior.

Absent from our list of performance measures is memory bandwidth. While memory bandwidth is

strongly related to hardware prefetching performance, direct bandwidth measurement is only available on recent Intel Xeon microprocessors, and bandwidth measurement is per-socket, as opposed to per-thread or per-core. To facilitate prefetcher configuration on a wide range of architectures, we instead rely on `OFFCORE_REQUESTS:DEMAND_DATA_RD` as a surrogate measure, which measures a subset of memory bandwidth behavior.

We also omit any performance measurements which directly describe hardware prefetcher behavior. Many microarchitectures expose performance events which directly measure the behavior and efficacy of hardware prefetchers at various cache levels. While it may be tempting to incorporate these events as contextual information, these events will be zero when the associated prefetcher(s) are disabled (will not provide meaningful response) and non-zero otherwise.

4.3.3 Reward Function

Consider the log data for execution of a workload W of n programs, each executing on isolated cores $c_1, c_2, \ldots, c_n$, using random prefetcher configuration selections. The log data is generated by a controller which, independently on each core, measures the contextual information for prior sampling periods, randomly selects between two prefetcher configurations, and associates the resulting performance (Instructions per Cycle, IPC) to the selected configuration.

We measure workload performance for a prefetcher configuration policy π as the average speedup of each core's performance compared to the performance of the baseline policy 0 (enabling all prefetchers on all cores):

$$R(W, \pi) = \frac{1}{n} \sum_i \frac{IPC_i^\pi}{IPC_i^0} \tag{4.1}$$

where IPC_c^π and IPC_c^0 denote the IPC over the entire program execution running on core c using the specified policy.

We translate the logged performance data for each core by estimating the average performance each prefetcher configuration has system-wide. This includes the direct effect a configuration has on the core it was taken, as well as the indirect effect on all other cores. In order to establish meaningful

measures of average performance, we first partition traces into periods of consistent performance (program phases), and estimate average performance over those phases (as program behavior can vary greatly over time due to phasing behavior). Estimating speedup per-phase normalizes measurement across different phases — the reward will be relative to the average performance of the phase.

Phase Detection

From log data, we apply Pruned Exact Linear Time [78] (PELT) change-point detection, independently for each core, on the IPC trace for that core. PELT determines change points, i.e. points in which the distributional properties of the data change, in sequence data. While IPC measurements for a given program will depend on the prefetcher configurations used both on that core and system-wide, and while there may be drastic differences in performance due to those configuration changes, a visual inspection suggests that PELT is effective in discovering meaningful change-points in a program's performance. We consider the periods between change-points to be independent phases of program behavior.

Reward Calculation

A simple definition of reward could be defined as the speedup of an action (prefetcher configuration) against the the estimated average performance over the current program phase $p_{i,t}$,

$$R_{i,t} = \frac{1}{n} \sum_{i} \frac{IPC^{\pi}_{i,t}}{\overline{IPC}^{0}_{i,p_{i,t}}} - 1, \tag{4.2}$$

where $IPC^{\pi}_{i,t}$ is the IPC of the chosen action and $\overline{IPC}^{0}_{i,p_t}$ is the average performance on core c_i for every instance of action 0 during the phase. However, there are two unfortunate side-effects to using Equation 4.2 as the reward function. First, the reward function does not categorize the individual effect each core's configuration has on performance. Instead, the reward categorizes the performance effect of the system-wide prefetcher configuration, to which each particular core's configuration contributes, and assigns that same reward to each core's configuration for that sampling

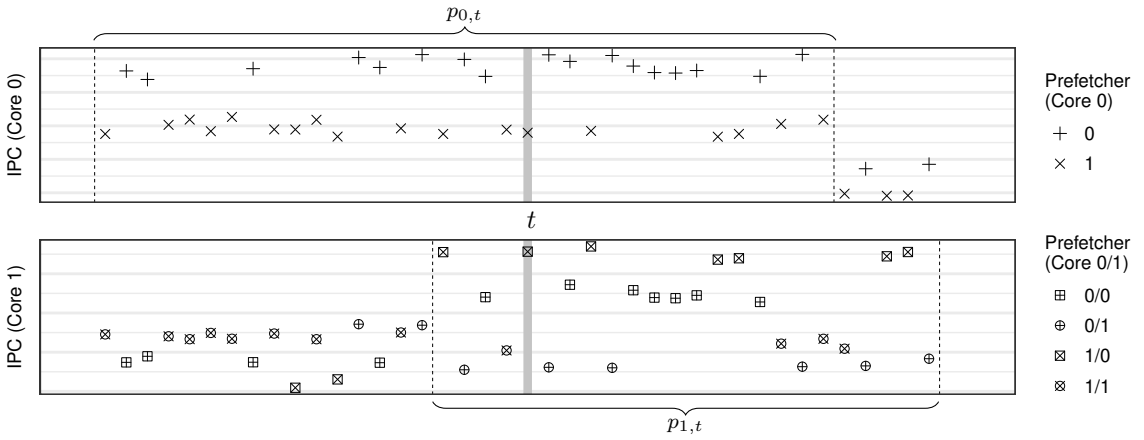


Figure 4.1: A small sample segment of log data from a random execution of a two-core workload.

period. Second, configurations which are close to the baseline will receive (approximately) no reward. Instead of rewarding the use of the baseline, when desirable, this reward function effectively indicates that these samples should be ignored.

In order to address these two problems, we consider the following two modifications to our reward calculation. First, we calculate the reward of each core's configuration independently, as an estimate of the average speedup that a particular core's configuration will cause system-wide. Second, instead of comparing the performance to a fixed baseline, we compare against the average performance of the random data.

To calculate the effect an action $a_{i,t}$ has on core c_i, we measure the speedup of that action against the average IPC over the phase $p_{i,t}$ containing $a_{i,t}$,

$$R_{(i),t} = \frac{IPC_{i,t}}{\overline{IPC}_{i,p_{i,t}}}, \tag{4.3}$$

where $IPC_{i,t}$ is the IPC of the action on core c_i at time t and $\overline{IPC}_{i,p_{i,t}}$ is the average IPC of all actions on core c_i during the phase containing time t. Figure 4.1 (Core 0) illustrates this calculation. At time t, we calculate the effect of action $a_{0,t} = 1$ (disable prefetching) by comparing its IPC with the average IPC on core c_0 of all actions in the phase containing t.

To calculate the effect an action $a_{i,t}$ has on some other core c_j, where $i \neq j$, we measure the speedup

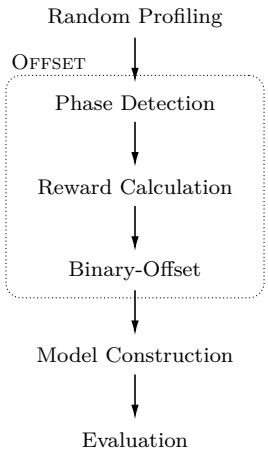


Figure 4.2: Overview of the Binary-Offset model construction and model evaluation workflow for hardware memory prefetcher utilization.

of that action on core c_j against the average IPC over the phase $p_{j,t}$ containing $a_{j,t}$, for all samples sharing that action $a_{j,t}$,

$$R_{(i,j),t} = \frac{IPC_{j,t}}{\overline{IPC}_{j,p_{j,t}|a_{j,t}}}. \tag{4.4}$$

where $\overline{IPC}_{j,p_{j,t}|a_{j,t}}$ is the average IPC of all actions on core c_j during the phase containing time t that used action $a_{j,t}$. Figure 4.1 (Core 1), we illustrates this calculation. At time t, the observed IPC on core c_1 indicates the performance of taking $a_{0,t} = 1$ and $a_{1,t} = 0$. This is divided by the average IPC on core c_1 of all actions in phase containing t that are identical to action $a_{1,t}$.

The calculated reward for each action $a_{i,t}$ is then the average estimated speedup that action has locally on the core it was taken (Equation 4.3) and remotely on all other cores (Equation 4.4),

$$R_{i,t} = \frac{1}{n}\left(R_{(i),t} + \sum_{i \neq j} R_{(i,j),t}\right) - 1. \tag{4.5}$$

If action $a_{i,t}$ shows an average speedup across all four cores, then the reward is positive; if the action provides an average slowdown across all four cores, then the reward is negative.

4.3.4 Policy Construction

Figure 4.2 summarizes the workflow for constructing decision policies from log data. Log data is generated from profiling executions which select hardware prefetcher configurations randomly over time and record system behavior (context), prefetcher configuration (action), and system performance (reward). The log data is transformed using the Binary-Offset algorithm [21], summarized in Algorithm 1, and a policy is constructed from the transformed data using machine learning. The result is a policy which classifies system behavior measurements according to the prefetching configuration which is predicted to provide optimal performance. Binary-Offset has been successfully applied to fine-grained paging mode selection in virtual machines [61], although that application focused on single core workloads.

At a high level, Binary-Offset can be described as a transformation procedure which maps the context-action-reward tuples $(\vec{x}, a, r)$ of log data for random action selections to weighted classification tuples $(\vec{x}, \hat{y}, w)$ where $\hat{y}$ is the expected superior action and w is the weight of that expectation. When an action has a positive reward, the chosen action a becomes the label $\hat{y}$ and is weighted according to the reward. When an action has a negative reward, the opposing action becomes the label $\hat{y}$ and is weighted according to the opposite of the reward. A weighted classifier uses the transformed dataset to generate a policy function $\pi(\vec{x})$ which describes which action is preferred given an observed context. Alternatively, the data can be resampled according to its weight [148], which allows standard, non-weighted classifiers to also be used.

Due to noise and unobserved system phenomenon, variation in system performance measurements can lead to substantial noise in the labels assigned by Binary-Offset. These label outliers can be problematic for standard, quadratic error classifiers such as logistic regression, Support Vector Machines, etc., and their weighted counterparts. To that end, we use a weighted variant of the Smooth 0–1 Loss Approximation (SLA) algorithm [99], which is robust to label outliers, to construct a classifier from the weighted classification data resulting from Binary-Offset. SLA refines the (linear) classifier from a (weighted) Support Vector Machine (SVM) using a mixture of gradient decent, pattern search, and hill-climbing according to a differentiable approximation of the robust

Table 4.2: Hardware Configuration

	CPU	Cores	Memory	Cache		
				L1	L2	L3
Broadwell Xeon E5-2620 v4	2.1 GHz	8	4x32 GB DDR4-2400	32 K 8-way	256 K 8-way	20 M 16-way
Sandy Bridge Core i5-2500	3.3 GHz	4	2x2 GB DDR3-1333	32 K 8-way	256 K 8-way	8 M 12-way
Kaby Lake Core i7-7700	3.6 GHz	4	4x8 GB DDR4-2400	32 K 8-way	256 K 8-way	8 M 16-way

0–1 loss. As SLA is a linear classifier, we expand the context vector to include pairwise interaction terms in order to capture some non-learner interactions between system behaviors. Using the SLA classifier, a policy can direct hardware prefetcher configuration on each core according to per-core and system-wide performance measures.

4.4 Methodology

We evaluate our method for constructing prefetching control policies on three machines, as detailed in Table 4.2. These three environments present a broad range of system configurations which in turn present a broad range of hardware prefetcher performance on multi-core workloads. For convenience, we refer to each machine according to its microarchitecture code-name: Broadwell, Kaby Lake, and Sandy Bridge. In all cases, we disable turbo boost and energy saving features. We additionally disable hyper-threading, as hardware prefetchers are shared between virtual cores.

4.4.1 Workload Selection

In order to evaluate the effect of our dynamic hardware prefetcher controller, we generate 60 four-core workloads by combining selected benchmarks from the SPEC CPU2006 [60], SPEC CPU2017 [126], and PARSEC [25] benchmark suites. In order to generate workloads which exhibit a variety of performance characteristics due to hardware prefetcher configurations, we limit our consideration to

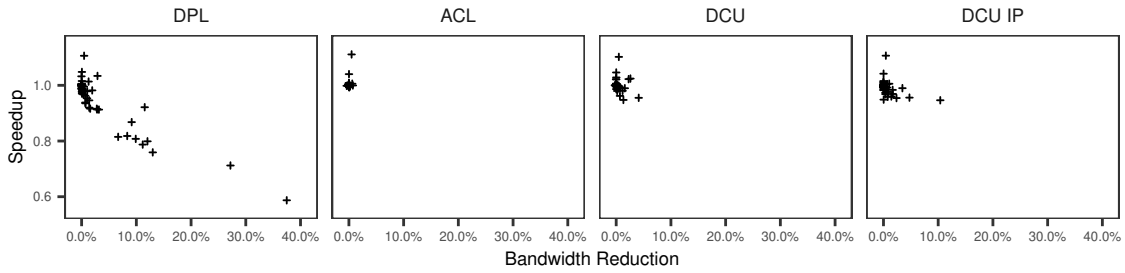


Figure 4.3: Change in prefetcher performance and memory bandwidth utilization for benchmarks from SPEC CPU2006, SPEC CPU2017, and PARSEC.

a subset of exemplars which are sensitive to hardware prefetching. Our selection criterion include substantial changes in performance and memory bandwidth consumption when prefetching is enabled or disabled. As such benchmarks should exhibit noticeable trade-offs in multi-core, co-tenant environments with shared memory resources.

Bandwidth and performance sensitivities for each hardware prefetcher and benchmark are given in Figure 4.3. For each hardware prefetcher, we execute each benchmark in isolation with that hardware prefetcher enabled and disabled (while the remaining prefetchers are enabled). Using Memory Bandwidth Monitoring (MBM) tools available on recent Xeon microprocessors, we measure performance sensitivity as the speedup of the benchmark when prefetching is disabled and bandwidth reduction as the change in memory bandwidth usage between the enabled and disabled cases as a percentage of total available memory bandwidth. On Broadwell, benchmarks are most sensitive to the usage of the DPL prefetcher in terms of both performance and memory bandwidth usage, with benchmarks such as `libquantum` and `fotonik3d_r` exhibiting significant performance loss (41%, 29%) and reduced memory bandwidth (37%, 27%), respectively, when L2 stream prefetching is disabled. In contrast, the ACL, DCU, and DCU IP prefetchers do not have as prominent of an effect on any benchmark in any benchmark suite. As MBM is unavailable on the remaining evaluation environments, we utilize the Broadwell measurements exclusively to conduct benchmark selection.

A subset of 20 benchmarks, detailed in Table 4.3, are selected according to the performance and memory bandwidth changes for the DPL prefetcher. A benchmark is selected if there is more than a 1% reduction in bandwidth usage when the DPL prefetcher is disabled. A total of 60 workloads

Table 4.3: Benchmark Selections by Suite

SPEC CPU2006	`bwaves`, `gcc`, `GemsFDTD`, `lbm`, `leslie3d`, `libquantum`, `mcf`, `milc`, `omnetpp`, `soplex`, `wrf`, `xalancbmk`
SPEC CPU2017	`bwaves_r`, `fotonik3d_r`, `gcc_r`, `lbm_r`, `mcf_r`, `omnetpp_r`, `roms_r`
PARSEC	`fluidanimate`

are constructed by selecting, with replacement, four benchmarks from the set of exemplars. The 60 workloads are shared across all three experimental machines.

4.4.2 Workload Execution

Workloads are executed by repeatedly executing their constituent programs, each isolated on a distinct core, until each program has completed execution at least once. A simple Python controller orchestrates the execution, performance monitoring, and hardware prefetcher configuration. The controller measures performance events (contextual information) over short sampling periods. At the end of each sampling period, the controller updates the prefetcher configuration according to some policy using the collected contextual information. This includes static policies which fix each prefetcher to a predetermined state during the workload's execution, or dynamic policies which adjust the prefetching configuration according to the workload characteristics described by the context. While we utilize a user-space controller for convenience, a kernel-space implementation is also viable within the constraints of our framework.

We restrict our evaluation to 1 s sampling periods, however any small period should be sufficient; in practice, we find similar results utilizing 10 ms and 2 s sampling periods. Contextual information is obtained using `libpfm4` and hardware prefetchers are configured by directly writing MSR `0x1A4` on each core. Due to a lack of availability for certain performance events on some architectures, the event sets for each evaluation machine differ slightly (see Table 4.1). However, the underlying structures described by the chosen events remain the same across all three machines. In all cases, each performance event is normalized per the number of instructions executed during the same interval.

4.4.3 Experimental Design

To establish baseline performances, we evaluate each workload by enabling and disabling a prefetcher of interest on all cores (while the remaining prefetchers are enabled). On both Sandy Bridge and Kaby Lake, there is at most a 6% difference between the enabled and disabled performance for the APL and DCU prefetcher across all 60 workloads. In comparison, there is up to a 30% difference between the enabled and disabled performance for the DPL prefetcher on Sandy Bridge and up to a 12% difference for the DCU IP prefetcher. As such, we choose to focus our evaluation on the DPL and DCU IP prefetchers where the performance change is marked. On Broadwell, disabling any prefetcher across all cores leads to a performance loss on the selected workloads. The same is true for eight-core workloads, chosen from the same set of exemplar benchmarks.

We consider the performance of executing the workloads using the following static policies:

All Enabled All four prefetchers are enabled on all cores.

[Prefetcher] Disabled The given prefetcher, DPL or DCU IP, is disabled on all cores; the remaining prefetchers are enabled on all cores.

Best Static The given prefetcher is either enabled or disabled on all cores, so as to give the best static performance.

We assume that All Enabled is a sensible default configuration, although this designation depends on workload characteristics and is often contraindicated by vendors for virtual environments [75]. The Best Static policy is a baseline measure of improvement, as it indicates the performance of an oracle which is limited to utilizing either the All Enabled or the Disabled policy for specific benchmarks.

Dynamic prefetcher configuration policies are constructed independently for the DPL and DCU IP prefetchers using Binary-Offset on small subsets of training workloads (this process was detailed in Section 4.3). Training workloads are selected uniformly according to the performance of the DPL Disabled and DCU IP Disabled policies, respectively. The training dataset is constructed by compiling the Binary-Offset transformations for exactly one random execution (randomly enabling

or disabling the hardware prefetcher of interest at each sampling period) of each training workload. To measure the effect of training workload size and specificity, we consider three training cases for policy construction:

Binary-Offset (Ind) Policies are constructed specific to each workload using only a given workload's training data and each policy is evaluated only on the training workload.

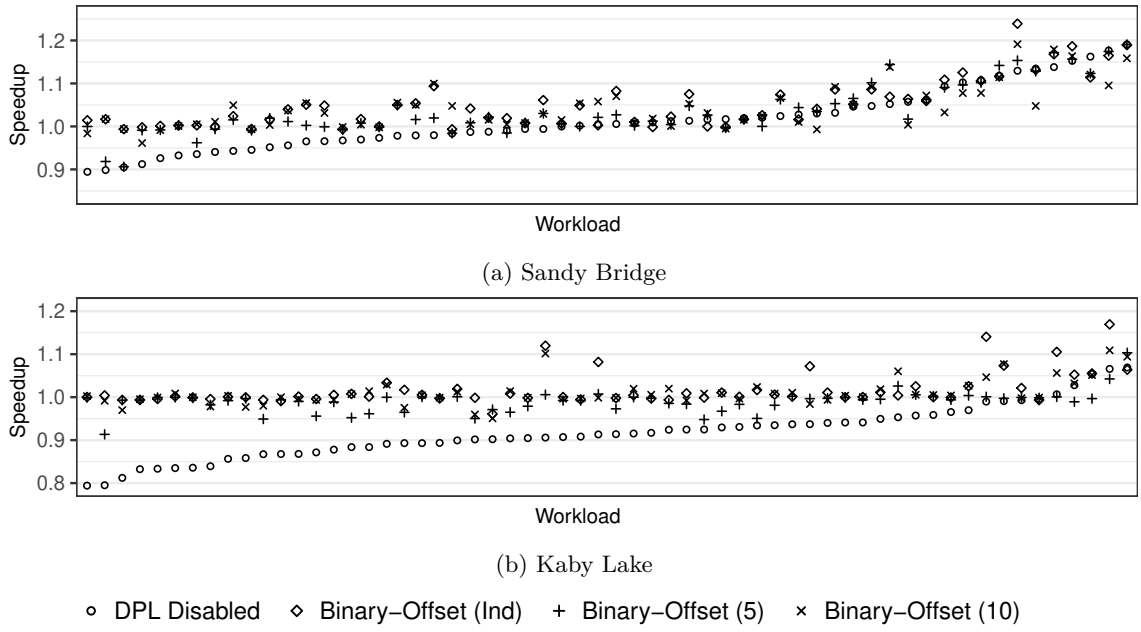
Binary-Offset (5) A single, general-purpose policy is constructed using the profiling data from 5 training workloads and is evaluated on the full workload suite.

Binary-Offset (10) A single, general-purpose policy is constructed using the profiling data from 10 training workloads and is evaluated on the full workload suite.

4.5 Results

We first examine the performance of the static and learned, dynamic Binary-Offset policies for the DPL prefetcher. Figure 4.4 details the average speedup of the DPL Disabled and Binary-Offset policies on Sandy Bridge and Kaby Lake. In both cases, the All Enabled static policy is used as the baseline against which all four policies are compared. With the exception of Binary-Offset (5) on Kaby Lake, the Binary-Offset policies outperform both the All Enabled and DPL Disabled static policies on a majority of workloads (up to 11% on Sandy Bridge and 14% on Kaby Bridge). In all but a small handful of workloads, the Binary-Offset policies never perform worse than both the All Enabled and DPL Disabled static policies, and the performance loss is no greater than 1% in all cases. On Broadwell (not shown), the Binary-Offset policies enable the DPL prefetcher almost always, likely due to the lack of resource contention.

The average performance for each policy across the full workload suite is detailed in Figure 4.5. On Sandy Bridge, disabling the DPL prefetcher produces better average performance (1.2%) across all 60 workloads; on Kaby Lake, disabling the DPL prefetcher substantially degrades performance (−8.5%). With the exception of Binary-Offset (5) on Kaby Lake, the Binary-Offset policies outperform not only the All Enabled and DPL Disabled policies, but the Best Static policy



(a) Sandy Bridge

(b) Kaby Lake

Figure 4.4: Workload performance for DPL prefetcher related policies on Sandy Bridge and Kaby Lake experimental environments, relative to baseline All Enabled (all prefetchers enabled on all cores).

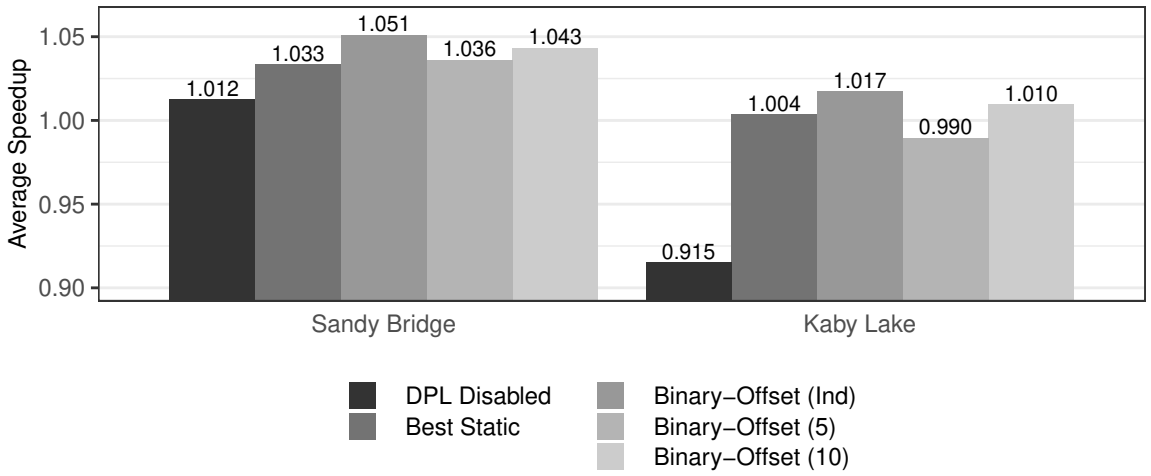


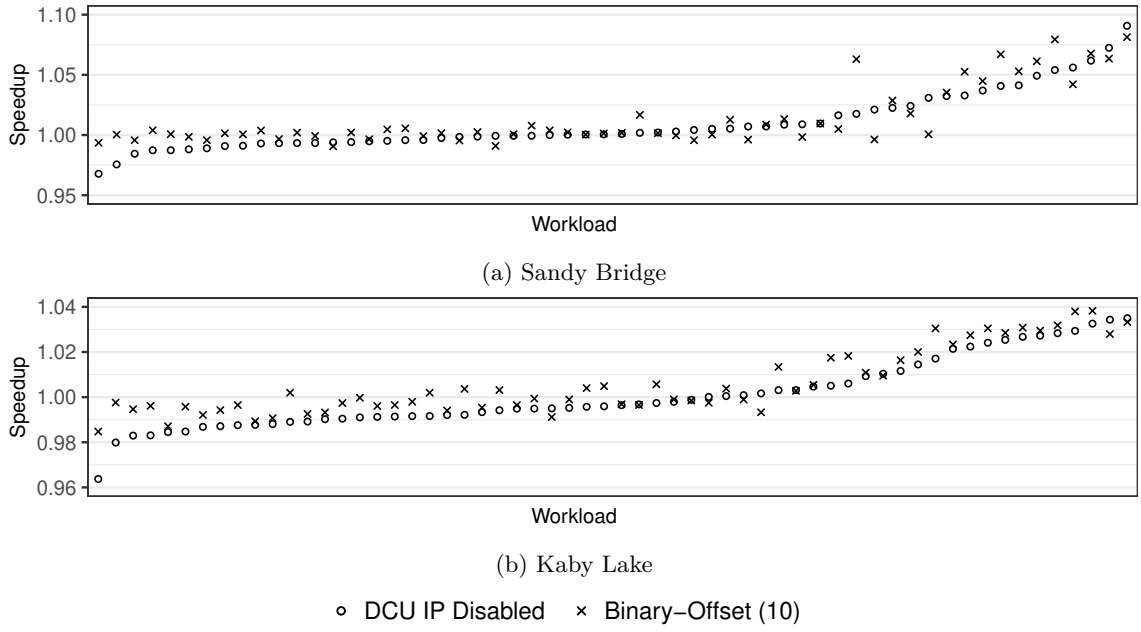
Figure 4.5: Comparison of (geometric) mean policy performance on both the Sandy Bridge and Kaby Lake for the DPL prefetcher.

as well. The Binary-Offset (10) policies improve upon the baseline All Enabled policy by 4.3% and 1.0% on Sandy Bridge and Kaby Lake, respectively, and improve upon the Best Static policy by 1.0% and 0.6%, respectively. Generating policies with increased specificity, as in the Binary-Offset (Ind), further improves average performance by tailoring prefetcher configuration to a narrower set of characteristics.

To better understand the behavior of policies resulting from Binary-Offset, we examine a workload consisting of benchmarks `libquantum`, `wrf`, and two instances of `lbm_r`, which achieves an average performance increase of over 10% when using either the Binary-Offset (Ind) or Binary-Offset (10) model. When executed on Kaby Lake, disabling the DPL prefetcher across all cores causes `libquantum` to receive a small (3.5%) performance gain, while the remaining programs each suffer a substantial (20%) performance loss. In contrast, the Binary-Offset policies enable the DPL prefetcher for `libquantum` and `wrf` (gaining 65% and 20% performance, respectively) while disabling the DPL prefetcher for the both `lbm_r` instances during a majority of the execution (losing 25% performance for both). As `libquantum` is a significant consumer of memory bandwidth and is very sensitive to DPL prefetch usage, enabling prefetching while eliminating contention for the less sensitive `lbm_r` instances provides a substantial increase in average workload performance.

Figure 4.6 details the average speedup of the DCU IP Disabled and Binary-Offset (10) policies on Sandy Bridge and Kaby Lake. As with the DPL prefetcher case, we use the All Enabled static policy as a baseline against which the other policies are compared. While the DCU IP prefetcher is less impactful on performance, we observe similar characteristics to the DPL prefetcher with regards to the Binary-Offset (10) policy performance. In over 55% of workloads on Sandy Bridge and 40% of workloads on Kaby Lake, the Binary-Offset learned policy outperforms both the All Enabled and DCU IP Disabled static policies. In all by a small handful of workloads, the Binary-Offset (10) policy never performs worse than both the All Enabled and DCU IP Disabled static policies, and the performance loss is no greater than 1% in all cases.

The average performance for each policy across the full workload suite is detailed in Figure 4.7. On Sandy Bridge, disabling the DCU IP prefetcher produces better average performance (0.9%). On Kaby Lake, there is little difference in average performance, as the number of cases which favor



(a) Sandy Bridge

(b) Kaby Lake

Figure 4.6: Workload performance for DCU IP prefetcher related policies on Sandy Bridge and Kaby Lake experimental environments, relative to baseline All Enabled (all prefetchers enabled on all cores).

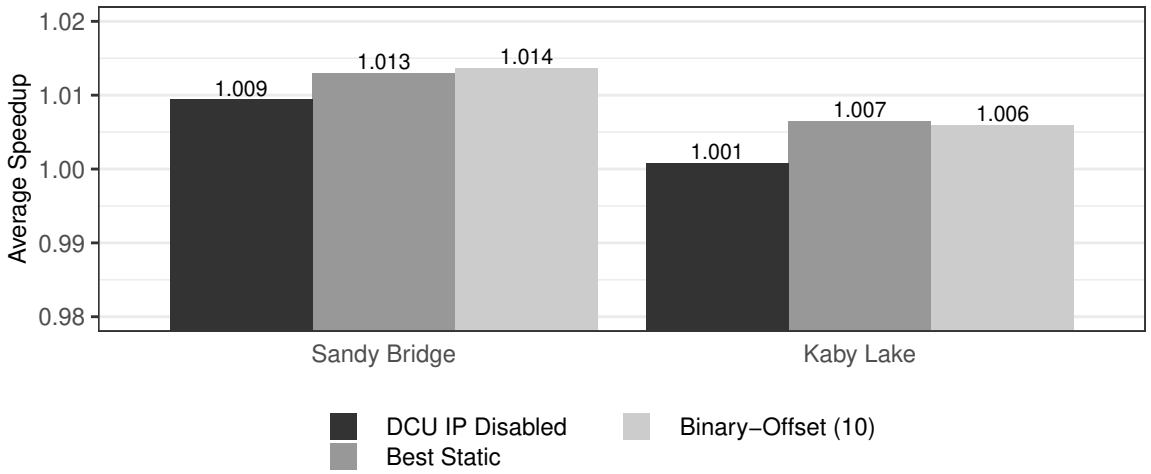


Figure 4.7: Comparison of (geometric) mean policy performance on both the Sandy Bridge and Kaby Lake for the DCU IP prefetcher.

enabling and disabling the prefetcher are relatively balanced. On both Sandy Bridge and Kaby Lake, the Binary-Offset (10) policy improves performance compared to both baselines, and has similar performance to the Best Static policy. While there are many instances, especially on Sandy Bridge, in which performance can be substantially improved compared to both the All Enabled and DCU IP Disabled policies, there are many cases in which disabling the prefetcher is advantageous yet the Binary-Offset policy enabled the prefetcher across all cores.

4.6 Discussion and Conclusion

For our contextual bandit model of fine-grained hardware prefetching control, we focused on simplified selections for actions and contextual information, and focused our attention to a particular reward function and training methodology. However, each of these aspects can be relaxed accordingly. For action selection, our approach focused on prefetcher control for independent hardware prefetchers and cores, operating in concert on a multi-core system. Preliminary results suggest that policies for both the DPL and DCU IP hardware prefetchers can likewise be applied in concert with a synergistic effect, providing independent but simultaneous control of the two prefetchers on all cores. In the future, methods to support a larger sets of actions could be applied, allowing for joint control of a prefetcher over a set of cores, or joint control of all prefetchers on a specific core.

Context selection remains an open question. As the performance events exposed by the PMU can differ between manufacturer and microarchitecture, it can be difficult to establish a consistent set of relevant performance events for prefetching behavior. Due to terse official documentation, and potentially buggy hardware implementations which contradict documentation, choosing appropriate events is a challenging task even for domain experts. Further, an overly broad selection of events can impact classification accuracy. As such, incorporating automatic context selection would be a valuable addition to our framework.

The reward function derived here is motivated by the use of average speedup as an optimization target. However, there are several metrics which summarize the performance of a multi-core workload, including total throughput ($\sum_i IPC_i^\pi$), which seeks to optimize aggregate system IPC (often at the

expense of low IPC applications), and fair speedup ($n \sum_i IPC_i^0/IPC_i^\pi$), which rewards a uniform improvement in application speedup. In either case, a new reward function derivation would be required to target these metrics.

Selecting an informative set of benchmarks as training workloads is a difficult problem. While our approach utilizes workload profiling of static hardware prefetcher configurations to make an informed decision, the uniform selection of training workloads given static prefetcher performance is a safe but uninformed choice. In practice, it may be possible to tailor the training workloads to account for expected workload types. Ideally, one would construct several models using different training workload sets and evaluate each to determine which model provides the best performance on a general testing workload set. However, such an approach requires a significant time cost to perform random profiling across each training workload set and to evaluate each resulting model on a broad set of testing workloads.

Leveraging this contextual bandit model, we describe a method for learning control policies for hardware prefetching in multi-core systems using profiling data obtained through random prefetcher configuration selections. Despite operating independently, per-core, the resulting prefetcher control policies are capable of tailoring prefetcher usage which is advantageous system-wide, disabling or enabling to reduce resource contention and improve system performance. We evaluate our approach on multi-core workloads constructed from prefetching sensitive SPEC CPU2006, SPEC CPU2017, and PARSEC benchmarks. On a system with limited memory bandwidth, our learned L2 stream prefetching control policy outperforms a typical baseline, which leaves prefetching enabled on all cores, by up to 19% and by 4.3% on average across our workload suite. Using control policies tailored to specific workloads, the same baseline is outperformed by up to 24% and by 5.1% on average.

Chapter 5

Performance Event Selection

While modern microarchitectures expose access to hundreds of performance events, the number of events which can be simultaneously monitored is limited by the number of available measurement registers (typically four or eight). It is not practical to measure the full set of performance events without substantial error; instead, a subset of events must be identified for sampling. Manual event selection adds a substantial human cost to model construction, requiring domain expertise to identify which events are relevant to the control problem. This cost is compounded by poor documentation of performance events, and an event availability which changes substantially between microarchitectures.

This chapter presents Correlation-Based Feature Selection (CFS) as a method for identifying relevant performance events, and evaluates CFS for hardware memory prefetching control. Filter methods, such as CFS, identify relevant features through statistical relationships in training data. As such, these methods are immediately applicable to the (transformed) random profiling data generated by the off-policy contextual bandit. The effectiveness of the CFS event selections and the domain-expert event selection (of Chapter 4) are evaluated according to the performance of the models that result from using the respective features. In addition, the validity of the CFS event selections for hardware memory prefetcher control are analyzed using available microarchitecture documentation.

5.1 Introduction

Hardware memory prefetching can be an effective tool for mitigating the cost of memory latency by anticipating future memory accesses and requesting that memory in advance. While hardware

prefetchers are largely effective in single-threaded applications, prefetching can be detrimental for multi-tenant workloads as it can increase demand for shared memory resources such as a shared last-level cache or memory bandwidth. Inter-core contention for shared resources can have a destructive effect on performance, by evicting useful cache lines with prefetched memory, or by increasing pressure on memory bandwidth with additional traffic for prefetching.

On many microarchitectures, hardware memory prefetching can be controlled at runtime. Recent Intel microarchitectures expose controls for four hardware prefetchers, allowing each prefetcher on each core to be enabled or disabled at runtime. A number of prefetcher control schemes [105, 88, 62, 70] have been designed to take advantage of these mechanisms in order to optimize performance by affecting prefetcher usage beneficial to current workload behavior.

The Performance Monitoring Unit (PMU) is a common mechanism available on modern microarchitectures for measuring the behavior of hardware components [66, 105, 88, 62]. However, the capabilities of the PMU are limited. The number of events available for measurement has far outpaced the number of events which can be measured simultaneously [149]. Recent microarchitectures expose thousands of unique events, but allow for the simultaneous measurement of only a few events (4 or 8). Measuring the full suite of events is impractical, as multiplexing will introduce a source of measurement error [14], and unnecessary, as a substantial number of events will be irrelevant or redundant to the application or domain.

Relying on domain expertise to select relevant performance events places a significant human cost on developing effective, informed models for system characterization and optimization. Performance event availability depends on the microarchitecture and vendor. Many events describe components which are specific to the microarchitecture, and there are few events which are standardized across microarchitectures. This is further complicated by poor, and in some cases incorrect, documentation for performance events. The terse event descriptions that are published are often difficult to dissect. Without ample documentation (which is often unavailable) and domain expertise, it can be challenging to understand the relationships between performance events and the effects of these events to a given problem.

In this chapter, we evaluate Correlation-Based Feature Selection (CFS) as a method for selecting performance events which are relevant to dynamic hardware prefetching control [59]. For two systems, offering stark differences in performance event availability and memory resources, we use CFS to identify performance event selections which are effective for determining hardware prefetcher usage in the presence and absence of contention for the shared last-level cache and memory bandwidth. On a memory-limited system, dynamic hardware prefetcher control using CFS selected events improves performance by 5.6% compared to the baseline which enables hardware prefetching system-wide. Compared to the domain-expert selected events, the CFS selected events improve dynamic prefetcher control performance by up to 1.2% on both a memory-limited system and a system with ample memory resources.

The chapter is organized as follows. Section 5.2 reviews hardware memory prefetching and summarizes work related to software control of hardware prefetchers. Section 5.3 reviews feature selection and introduces Correlation-Based Feature Selection. Sections 5.4 and 5.5 discuss our methodology, experimental results, and analysis. Section 5.6 details work related to hardware memory prefetcher utilization and the selection of relevant performance events for that purpose. Section 4.6 summarizes the framework and describes our conclusions.

5.2 Dynamic Hardware Prefetcher Control

Intel microarchitectures expose four prefetchers for runtime control [132, 67]. Two prefetchers operate on the L2 cache and respond to memory requests from the L1 data and instruction caches: the Data Prefetch Logic (DPL) prefetcher will fetch anticipated cache lines for ascending and descending streams of memory accesses; the Adjacent Cache Line (APL) prefetcher will fetch cache lines which complete a 128 byte aligned cache line pair. Two prefetchers operate on the L1 cache and respond to memory requests from the execution engine: the Data Cache Unit (DCU) prefetcher will fetch anticipated cache lines for ascending (but not descending) streams of memory access; the DCU Instruction Pointer (DCU IP) prefetcher will track loads and fetch anticipated cache lines which comprise ascending or descending stride accesses of up to 2K bytes. Prefetchers can be independently

enabled or disabled at runtime, independently on each core. Model Specific Register (MSR) `0x1A4` takes a binary mask indicating the state, either enabled or disabled, for each of the four prefetchers on a core. A typical system-wide default enables all four prefetchers on all cores.

While hardware memory prefetching can be effective at mitigating the cost of memory accesses and ameliorating the memory wall effect, the added contention for shared memory resources can cause a destructive performance effect in multi-core systems. When memory bandwidth is constrained and several benchmarks must contend for that bandwidth, the DPL prefetcher is contraindicated. For example, the SPEC CPU2017 [126] benchmark `fotonik3d` is very sensitive to DPL prefetcher usage. When executed in isolation, disabling the DPL prefetcher can cause a substantial decrease in program throughput with a corresponding decrease in memory bandwidth usage [62]. In contrast, `omnetpp` has little sensitivity to DPL prefetcher usage. When executed together using the DPL prefetcher, the memory pressure of `fotonik3d` can have a substantive negative effect on `omnetpp` performance. In contrast, when memory bandwidth is plentiful, there is insufficient contention to negatively affect performance.

Runtime control of hardware prefetchers allows for dynamic prefetcher usage which responds to changes in system behavior and performance over time [70, 62]. Hiebel et al. [62] model dynamic prefetcher control as a sequential decision process, measuring workload behavior at regular intervals and choosing to enable or disable each prefetcher in response. Workload behavior is characterized by a number of metrics exposed by the PMU. This includes branch predictor, cache, translation look-aside buffer (TLB) and memory bandwidth behavior, identified as relevant to hardware prefetcher performance according to domain knowledge. The effectiveness of the model depends on the identified PMU events being sufficient to distinguish when prefetcher utilization is advantageous for system-wide performance.

5.3 Correlation-Based Feature Selection

Performance events are broadly used for the control and characterization of computer systems. However, limitations in design, lacking documentation, and complex inter-relationships between

hardware components complicate the effective use of performance event measurement for these outcomes. Identifying subsets of predictive events and filtering out irrelevant and redundant events can help mitigate PMU measurement error due to multiplexing [14].

Feature selection is a technique for identifying and removing predictors, or features, which are redundant or irrelevant to an outcome [57]. Filter methods are a computationally efficient class of feature selection algorithms which operate directly on the characteristics of the data, relating features (e.g., performance event measurements) to the corresponding classes (e.g., preferred hardware prefetcher usage). Filter methods typically rank and select features according to univariate and multivariate measures of those features.

Correlation-Based Feature Selection (CFS) [59] selects feature subsets which maximize the heuristic of that subset's *merit*,

$$Merit_S = \frac{k\,\overline{r_{fc}}}{\sqrt{k + (k-1)\,\overline{r_{ff}}}}, \tag{5.1}$$

where S is a subset of k features, $\overline{r_{fc}}$ is the average correlation between the features of the subset and the corresponding classes, and $\overline{r_{ff}}$ is the average correlation between each pair of features in the subset. The merit heuristic rewards feature sets with a high average relevancy (feature-class correlation), and penalizes feature sets with a high average redundancy (feature-feature correlation).

Direct combinatorial optimization of $Merit_S$ is impractical for large sets of features. Instead, event subsets are selected with a greedy search. This can either be through a forward search, starting with the empty set of features and progressively adding events which maximizes merit, until adding any additional events will only reduce the merit heuristic, or similarly through a backward search, starting with the full set and progressively removing events.

5.4 Methodology

We utilize the Sandy Bridge and Kaby Lake experimental environments detailed in Table 4.2. The two environments represent different Intel microarchitectures (Sandy Bridge and Kaby Lake), which have substantial differences in the available performance events. In addition, the two environments

have different memory characteristics, with Kaby Lake having significantly more available memory and memory bandwidth. In both cases, turbo-boost and energy-saving features are disabled. We additionally disable hyper-threading, as counter use is otherwise restricted.

For each system, performance events are identified using `libpfm4`, which provides a set of mnemonic designations and corresponding configuration details for each performance event exposed by the microarchitecture. From the complete list of available events, we prune a number of events according to the following criterion. We prune any event which is limited in use to a subset of the available performance counter registers. A small number of events on each microarchitecture are documented as being limited in this way. Similarly, we remove events which require additional, non-standard configuration, such as requiring additional MSRs to parameterize the events measured (e.g. `MEM_TRANS_RETIRED:LOAD_LATENCY` measures memory accesses which exceed a user-defined threshold specified with MSR `0x3F6`). The total number of (unique) event mnemonics reported by `libpfm4` for Sandy Bridge is 255. After pruning, we are left with 235 unique events. For Kaby Lake, the total number of event mnemonics reported is 275, and after pruning we are left with 236 events.

5.4.1 Workload Design and Execution

We evaluate hardware prefetcher control using CFS selected events on the 60 workloads presented in Hiebel et al. [62]. Each workload is comprised of four benchmarks, selected with replacement from a set of twenty prefetcher-sensitive benchmarks from the SPEC CPU2006 [125, 60], SPEC CPU2017 [126], and PARSEC [25] benchmark suites.

A workload is executed by isolating each of the four benchmarks to a unique core, and executing each benchmark repeatedly on the corresponding core until all benchmarks have finished execution at least once. These workloads represent a broad range of workload behaviors with respect to last-level cache and memory contention and include cases for both evaluation environments in which prefetching on each core is advantageous or disadvantageous to system-wide performance. A user-space controller manages workload execution, performance event measurement, and hardware prefetcher control system-wide at fixed intervals.

The performance of each benchmark is measured as the average throughput of instructions retired per cycle (IPC) on that benchmark's core. Workload performance is measured according to the average speedup of each benchmark's performance when executed with some static or dynamic hardware prefetcher controler (conf), compared to the performance of each benchmark when executed with the static baseline with all prefetchers enabled on all cores (base):

$$Speedup = \frac{1}{4} \sum_i \frac{IPC_i^{\text{conf}}}{IPC_i^{\text{base}}}. \tag{5.2}$$

5.4.2 Dynamic Hardware Prefetcher Control

Controllers for dynamic hardware prefetching can be constructed using training data derived from random profiling [62]. During workload execution, a controller, at regular (1s) intervals, chooses a random, system-wide prefetcher configuration. The workload behavior and resulting performance corresponding to each random selection are recorded for the entirety of the workload's execution. For our purposes, we only consider the effects of each prefetcher in isolation. During random profiling, the hardware prefetcher of interest is enabled or disabled at random on each core, which the remaining hardware prefetchers remain enabled and unchanging on all cores.

Workload behavior consists of a set of PMU measurements for a specified set of performance events, utilizing multiplexed measurement if necessary. The event count data for each interval is normalized to the cycle count of that interval. Performance is measured as the throughput (IPC) of each core during that interval.

The Offset translation converts random profiling data into weighted classification data which is amenable to a wide range of instance-weighted classifiers. This translation consists of three main components:

1. *Phase Extraction*: Program behavior will often follow patterns, or phases, of repeating behaviors. A phase change occurs when the program undergoes a noticeable and sudden change in program behavior. Change-point analysis can be used to identify phase changes by detect-

ing points in which the statistical properties of a program's performance (IPC) have shifted. Phase changes for each program (i.e., on each core) are identified independently using the Pruned Exact Linear Time (PELT) [78]. The resulting phases are used to establish a baseline performance metric for each prefetcher configuration.

2. *Reward Calculation*: The reward for each core's prefetcher configuration, either enabling or disabling the configuration of interest, as the estimated speedup that configuration has system-wide compared to the average performance of acting randomly. To estimate system-wide speedup, we average the effect that each core's configuration has locally (on the same core) and remotely (on each other core), while controlling for the prefetcher usage on the remote cores.

3. *Binary-Offset*: The resulting data, consisting of workload measurements, random prefetcher selections, and the calculated reward are further transformed according to the Binary-Offset algorithm [21]. This translates the context-action-reward tuples in to a set of weighted classification data that describes the (normalized) workload behavior, the predicted, ideal configuration, and the weight of that prediction.

As each core addresses the same underlying decision process, the resulting data for each core is combined into a single dataset of that workload.

Models for prefetcher control are constructed according to the Binary-Offset transformed data using instance-weighted supervised learning. For each prefetcher, a set of ten training workloads are chosen uniformly based on the speedup of the static policy in which the prefetcher of interest is disabled across all cores. This ensures that the training is illustrative of a wide extent of performance variation. The ten training workloads are aggreated to form the training dataset.

In addition to the set of events measurements, we additionally incorporate higher-order features. This includes aggregate measures of each event system-wide, ensuring that each per-core controller can respond to system-wide resource usage, and binary interaction terms between the event and aggregate event features to measure non-linear relationships. For a total of k performance events selected, there are $2k^2 + k$ features. We utilize an instance-weighted variant of the SLA binary

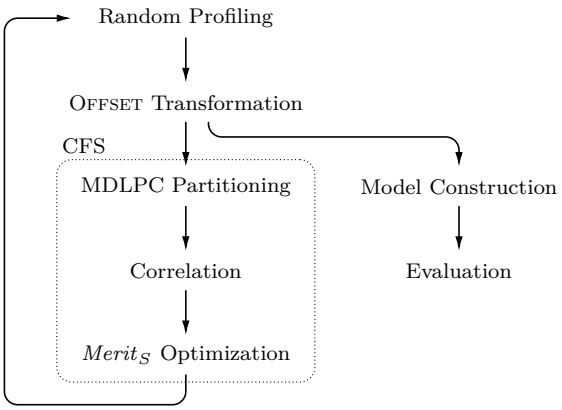


Figure 5.1: Overview of the CFS performance event selection, Binary-Offset model construction, and Binary-Offset model evaluation workflows for hardware memory prefetcher utilization.

classifier [99] to learn a linear model for hardware prefetcher control.

5.4.3 Event Selection

As a baseline, we use the domain-expert event selection of Liao et al. [88], Hiebel et al. [62], detailed in Table 4.1. The expert features are translated for Sandy Bridge and Kaby Lake using close analogs which are consistent across both microarchitectures. The events correspond primarily to cache (`L1D`, `L2_LINES_IN`, `L3_LAT_CACHE`) and memory behavior. This includes blocked loads (`LD_BLOCKS`), translation-lookaside buffer behavior (`DTLB_LOAD_MISSES`), and memory bandwidth (`OFFCORE_REQUESTS`). The set of eleven events is then reduced to eight events on each microarchitecture due to availability of `L1D` and `LD_BLOCKS` events. As both execution environments have eight performance counters, the resulting event selections require no multiplexing.

Figure 5.1 summarizes the workflow for performance event selection. On each system, random profiling is performed with the full set of (pruned) events, using multiplex sampling in order to estimate performance counts for the full set during each interval. Training data is created from the random profiling using the Offset transformation described in Section 5.4.2. The CFS event selection has three main components:

1. *MDLPC Partitioning.* In order to ensure a consistent comparison between the ordinal per-

formance event features and the nominal class descriptions (prefetcher enabled, disabled), we utilize the MDLPC partitioning algorithm to map each feature to a set of discrete values [48]. MDLPC recursively chooses the binary partition of the feature values which minimizes the information entropy induced by the partition. Partitioning continues while the information gain of the selected partition exceeds a condition based on the Minimum Description Length (MDL). The MDL criterion appeals to the regularity of the partitioned data: if the partition produces sub-sequences which are in total more regular than the original data, then that data can be described more compactly using the partition.

2. *Correlation.* After partitioning, the feature-class correlations (r_{fc}) and feature-feature correlations (r_{ff}) may be treated uniformly between nominal values. Symmetrical Uncertainty (SU), is a normalized, symmetric measure of mutual information,

$$\mathrm{SU}(r_1, r_2) = 2\left(1 - \frac{\mathrm{Ent}(r_1, r_2)}{\mathrm{Ent}(r_1) + \mathrm{Ent}(r_2)}\right), \tag{5.3}$$

   where Ent is entropy.

3. *$Merit_S$ Optimization.* We use the union of the forward and backward greedy searches as the events selected, "CFS Features". In order to provide a fair comparison to the domain-expert selection, we additionally consider a subset of eight events from the CFS selected features. We select this subset, "CFS Features (8)", using an enumerative search considering just the CFS features.

After selecting a subset of events, we repeat the process of random profiling and OFFSET transformation on only those events. A control model is now generated using SLA on the new training data set specific to the event selection. While the model could be trained on the initial set of trained data, using only the data for the selected features, we perform the additional profiling pass to ensure that the error of multiplex sampling is minimized.

Table 5.1: CFS Events for Sandy Bridge DPL Prefetcher

Mnemonic
BR_INST_EXEC:ALL_CONDITIONAL
BR_INST_EXEC:TAKEN_DIRECT_JUMP
BR_INST_RETIRED:CONDITIONAL
DTLB_STORE_MISSES:MISS_CAUSES_A_WALK
FP_ASSIST:SIMD_INPUT
ILD_STALL:IQ_FULL
ITLB_MISSES:MISS_CAUSES_A_WALK
L2_LINES_IN:E
L2_LINES_OUT:DIRTY_ANY
L2_RQSTS:ALL_DEMAND_RD_HIT
L2_STORE_LOCK_RQSTS:HIT_M
L2_TRANS:L2_WB
LD_BLOCKS:DATA_UNKNOWN
LD_BLOCKS:STORE_FORWARD
LSD:UOPS
MEM_LOAD_UOPS_RETIRED:L2_HIT
MISALIGN_MEM_REF:LOADS
OFFCORE_REQUESTS_OUTSTANDING:DEMAND_DATA_RD_GE_6
RESOURCE_STALLS:LD_SB
RESOURCE_STALLS:SB
UOPS_DISPATCHED_PORT:PORT_1
UOPS_DISPATCHED_PORT:PORT_5

5.5 Results

Following the analysis of [62], we restrict our focus to the DPL and DCU IP prefetchers. Sensitivity to these prefetchers is more substantial amongst the selected workloads compared to the ACL and DCU prefetchers.

5.5.1 DPL Prefetcher

The events selected by CFS for the DPL prefetcher on Sandy Bridge and Kaby Lake are given alphabetically in Tables 5.1 and 5.2. The eight event subsets are denoted in bold. Each feature selection is a combination of both the forward and backward greedy searches. However, we observe that there is substantial overlap between the two searches. On Sandy Bridge, with 22 total events in the union, the forward search produces 17 events and the backward search produces 21 events.

Table 5.2: CFS Events for Kaby Lake DPL Prefetcher

Mnemonic
CYCLE_ACTIVITY:STALLS_L3_MISS
DTLB_LOAD_MISSES:WALK_COMPLETED_4K
DTLB_STORE_MISSES:MISS_CAUSES_A_WALK
DTLB_STORE_MISSES:WALK_COMPLETED_4K
L2_RQSTS:ALL_DEMAND_MISS
LD_BLOCKS_PARTIAL:ADDRESS_ALIAS
LD_BLOCKS:NO_SR
LD_BLOCKS:STORE_FORWARD
LONGEST_LAT_CACHE:MISS
MEM_LOAD_L3_HIT_RETIRED:XSNP_HIT
MEM_LOAD_L3_HIT_RETIRED:XSNP_MISS
MOVE_ELIMINATION:SIMD_NOT_ELIMINATED
OFFCORE_REQUESTS_OUTSTANDING:DEMAND_DATA_RD_GE_6
OFFCORE_REQUESTS_OUTSTANDING:DEMAND_RFO_CYCLES
RESOURCE_STALLS:RS
TLB_FLUSH:DTLB_THREAD
UOPS_DISPATCHED:PORT_7
UOPS_ISSUED:FLAGS_MERGE

On Kaby Lake, with 18 total events in the union, both the forward and backward searches produce 17 events.

Selected Events

The CFS selected events can be categorized according to the affected hardware components: the in-order front end, out-of-order execution engine, cache, translation look-aside buffer TLB, and memory bus. Understanding the relevance of the selected events requires extensive investigation of the sparsely available literature regarding the specifics of each microarchitecture's implementation and the details of each event's description.

Front End. This class of events describes the behavior of several front end hardware components. This includes the Loop Stream Detector (`LSD`), the Instruction Length Decoder and Instruction Queue (`ILD_STALL:IQ_FULL`), and the Instruction Translation-Lookaside Buffer (`ITLB_MISSES`). Each event may be indicative of low-level patterns in execution which stress the instruction decode pipeline. Branch behavior (`BR_INST_EXEC`, `BR_INST_RETIRED`) is similarly indicative of execution

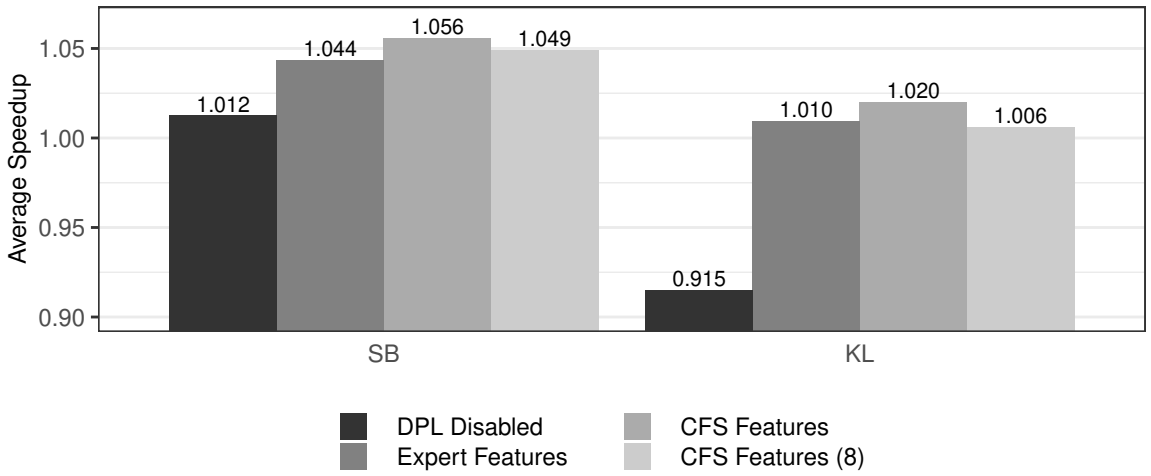


Figure 5.2: Comparison of (geometric) mean policy performance on both the Sandy Bridge and Kaby Lake for the DPL prefetcher.

patterns wherein hardware prefetcher utilization may or may not be constructive. Events such as `UOPS_ISSUED:FLAGS_MERGE`, `MOVE_ELIMINATION`, and `FP_ASSIST` indicate pathological cases which can incur a notable performance penalty.

Execution Engine. Specific µop execution ports are emphasized in the event selections for both systems (`UOPS_DISPATCHED`, `UOPS_DISPATCHED_PORT`). On Sandy Bridge, ports 1 and 5 are responsible for a broad range of instructions, most notably for load effective address (LEA) instructions and branch instructions on Port 5. On Kaby Lake, port 7 is responsible solely for store address commands [67].

Various resources in the execution engine can cause load instructions to become blocked. `LD_BLOCKS` events measure loads which are blocked due to interactions with the store buffer (`STORE_FORWARD`, `DATA_UNKNOWN`) or a lack of available resources to handle a "split" load which crosses a cache line boundary (`NO_SR`). The performance penalty of split loads is also measured by `MISALIGN_MEM_REF`. Related `RESOURCE_STALLS` events measure stalls due to a lack of available load and store buffer resources (`LD_SB`, `SB`), and more generally due to a lack of available reservation station entries (`RS`).

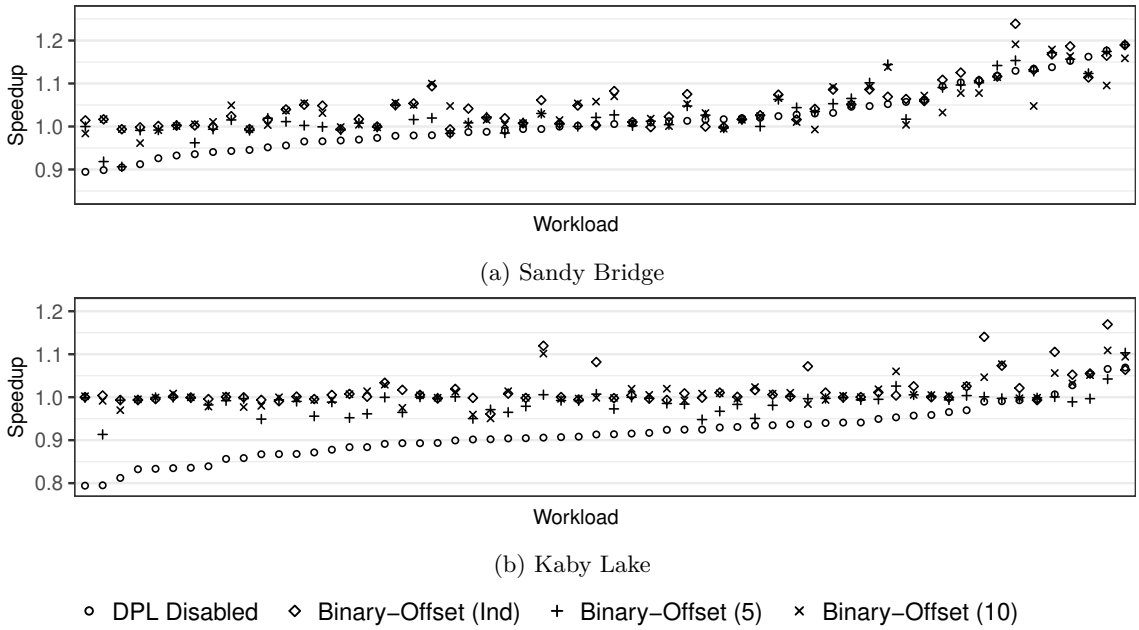
Cache. A broad set of L2 and L3 cache-related events are selected by CFS on both systems, including L2 cache lines filled and evicted (`L2_LINES_IN`, `L2_LINES_OUT`), L2 cache access requests

(L2_RQSTS, L2_STORE_LOCK_RQSTS, L2_TRANS), and L3 cache misses (CYCLE_ACTIVITY:STALLS_L3_MISS, LONGEST_LAT_CACHE:MISS). On Sandy Bridge, some L2 cache events in the selection are specific to cache coherency and the MESI state of the cached data being accessed: L2_LINES_IN:E counts the number of lines allocated in the L2 cache in the exclusive (E) state, and L2_STORE_LOCK_RQSTS:HIT_M counts the number of demand store (read-for-ownership, RFO) requests which hit in the L2 cache and the corresponding cache line was in the modified (M) state.

In addition to measuring L2 and L3 cache behavior directly, the selected events also count the the retirement of load instructions which based on whether or not that load hit in the respective cache (MEM_LOAD_UOPS_RETIRED, MEM_LOAD_L3_HIT_RETIRED). The later class of events correspond to cross-core snoop requests (XSNP), which are necessary to maintain cache coherency in multi-core systems. Under a hit in the shared last level cache, the cross-core snoop verifies whether the cache line is present (XSNP_HIT) or absent (XSNP_MISS) in the private caches of the other cores.

The relationship between the DPL prefetcher and both L2 and L3 cache is complicated by more recent advancements to the adaptive cache fill policy. Under certain conditions, the prefetcher may forgo populating the prefetched cache line in the L2 cache, and will instead only populate the cache line in the L3 cache. Forgoing data population in the L2 cache when that cache is stressed may help avoid useful cache lines from being prematurely evicted.

Data Translation Look-Aside Buffer (DTLB). High incidents of miss events in the data DTLB, e.g. DTLB_STORE_MISSES and DTLB_LOAD_MISSES, may suggest that the working set of a program is large or unpredictable and thus may suggest that stream prefetching is contraindicated. In addition, the DPL prefetcher restricts each of the 32 available in-flight streams to 4K page boundaries [67]. High incidents of DTLB misses would also suggest that prefetcher effectiveness is limited by this technical constraint, even in the presence of predictable stream accesses. The selected events measure when a miss in the DTLB causes a walk (MISS_CAUSES_A_WALK) and when a walk is completed for a 4K page (WALK_COMPLETED_4K). Under some circumstances, entries of the DTLB may become invalid and require flushing to prevent incorrect address translation. On single-threaded benchmarks, high incidents of TLB_FLUSH:DTLB_THREAD result from far calls which transfer to privileged code, as indicated by the high correlation with the BR_INST_RETIRED:FAR_BRANCH event.



(a) Sandy Bridge

(b) Kaby Lake

Figure 5.3: Workload performance for DPL prefetcher related policies on Sandy Bridge and Kaby Lake experimental environments, relative to the baseline.

Memory Bandwidth. When memory bandwidth usage is saturated, the addition of stream prefetching requests from the DPL prefetcher may be contraindicated. In the absence of events which directly measure memory bandwidth usage, off-core memory requests events (`OFFCORE_REQUESTS`, `OFFCORE_REQUESTS_OUTSTANDING`) can be used as a surrogate measure. The CFS selected events include `DEMAND_DATA_RD_GE_6`, which measures the number of cycles in which at least 6 outstanding demand data read transactions are in-flight (potentially indicating that the memory bandwidth is saturated) and `DEMAND_RFO_CYCLES`, which measures stores which have missed in the inclusive L3 cache and require cross-core invalidation (potentially indicating contention for the last-level cache). The DPL prefetcher will additionally monitor off-core traffic and will throttle the rate of streaming prefetch requests to avoid additional bandwidth usage and prevent additional bandwidth contention [67].

Model Performance

Figure 5.2 details the average workload speedup (Equation 5.2) of DPL prefetcher controllers compared to the static baseline (all prefetchers enabled). On average, CFS event selection outperforms

Table 5.3: CFS Events for Sandy Bridge DCU IP Prefetcher

Mnemonic
ARITH:FPU_DIV_ACTIVE
BR_INST_EXEC:TAKEN_COND
BR_INST_RETIRED:ALL_BRANCHES
BR_INST_RETIRED:CONDITIONAL
BR_MISP_EXEC:NONTAKEN_COND
BR_MISP_EXEC:TAKEN_COND
DTLB_LOAD_MISSES:WALK_DURATION
FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE
IDQ_UOPS_NOT_DELIVERED:CYCLES_LE_3_UOP_DELIV_CORE
ITLB_MISSES:WALK_DURATION
L2_L1D_WB_RQSTS:HIT_M
L2_LINES_IN:ANY
L2_LINES_OUT:DEMAND_DIRTY
L2_RQSTS:RFO_ANY
L2_TRANS:ALL
LD_BLOCKS_PARTIAL:ALL_STA_BLOCK
LD_BLOCKS:DATA_UNKNOWN
MEM_LOAD_UOPS_LLC_HIT_RETIRED:XSNP_HIT
OFFCORE_REQUESTS_OUTSTANDING:DEMAND_RFO

domain-expert event selection on both Sandy Bridge (1.2%) and Kaby Lake (1.0%) and the performance of the eight-event CFS selection is comparable to the performance of the eight-event domain-expert selection on both systems. This performance can be achieved with no knowledge of the underlying microarchitectural design, PMU details, or system specifications.

Individual workload performances for DPL prefetcher controllers, constructed from the domain-expert event selection and both CFS event selections, are given in Figures 5.3a and 5.3b. On both systems, there is ample opportunity for improvement compared to the static prefetcher controllers. In many cases, dynamic DPL control performance matches or exceeds both the static baseline controller (all prefetchers enabled on all cores) and the static DPL Disabled controller (DPL disabled on all cores). Further, for a majority of workloads, the three dynamic controllers result in similar performance (median difference in speedup less than 2.4% on Sandy Bridge and less than 2.0% on Kaby Lake).

Table 5.4: CFS Events for Kaby Lake DCU IP Prefetcher

Mnemonic
ARITH:FPU_DIV_ACTIVE
BR_INST_RETIRED:CONDITIONAL
DTLB_LOAD_MISSES:WALK_COMPLETED
DTLB_LOAD_MISSES:WALK_COMPLETED_1G
EXE_ACTIVITY:2_PORTS_UTIL
IDQ:MITE_UOPS_CYCLES
L2_LINES_IN:ALL
L2_LINES_OUT:USELESS_HWPF
L2_RQSTS:ALL_RFO
OFFCORE_REQUESTS_OUTSTANDING:L3_MISS_DEMAND_DATA_RD
RESOURCE_STALLS:RS
SW_PREFETCH_ACCESS:T0

5.5.2 DCU IP Prefetcher

The events selected by CFS for the DCU IP prefetcher on Sandy Bridge and Kaby Lake are given alphabetically in Tables 5.3 and 5.4. The eight event subsets are again denoted in bold. Similarly to the DPL Prefetcher, there is some overlap in the forward and backward searches. On Sandy Bridge, with 19 total events in the union, the forward and backward searches both produce 18 events. On Kaby Lake, with 12 total events in the union, the forward search produces 4 events and the backward search produces 11 events.

Selected Events

Many of the events which were present in the DPL event selections are also present in the DCU IP event selections. The events unique to the DCU IP selections are more closely related to front end and execution engine behavior. Notably, there is an increase in branch instruction (`BR_INST_EXEC`, `BR_INST_RETIRED`) and branch misprediction (`BR_MISP_EXEC`) events. Software prefetching, as indicated by the (`SW_PREFETCH_ACCESS`) event can have both a synergistic and antagonistic effect when used together with hardware prefetching [87]. Specific, non-memory related instruction behavior is also identified. This includes cycles in which the divider (integer and floating-point) are active (`ARITH:FPU_DIV_ACTIVE`) and SSE double precision scalar operations (`FP_COMP_OPS_EXE`), which

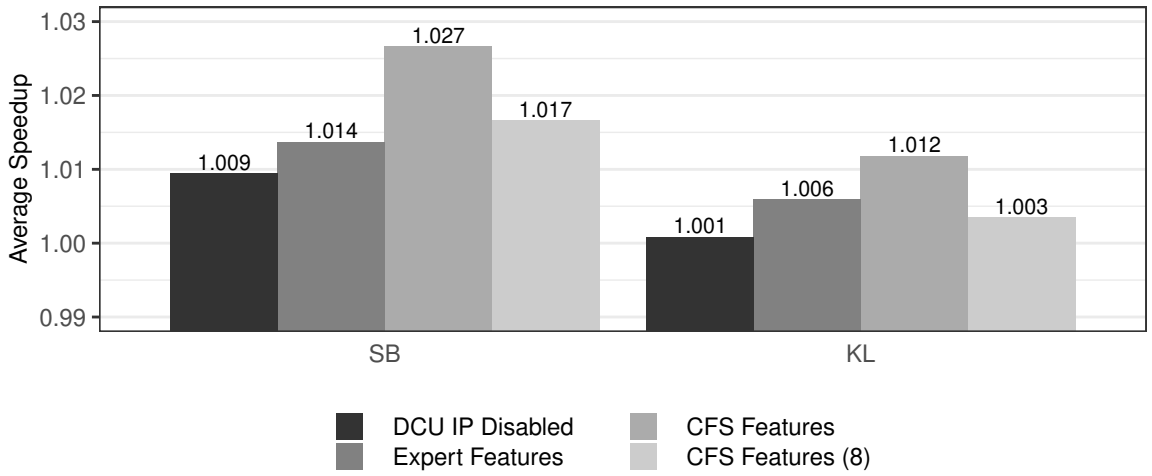


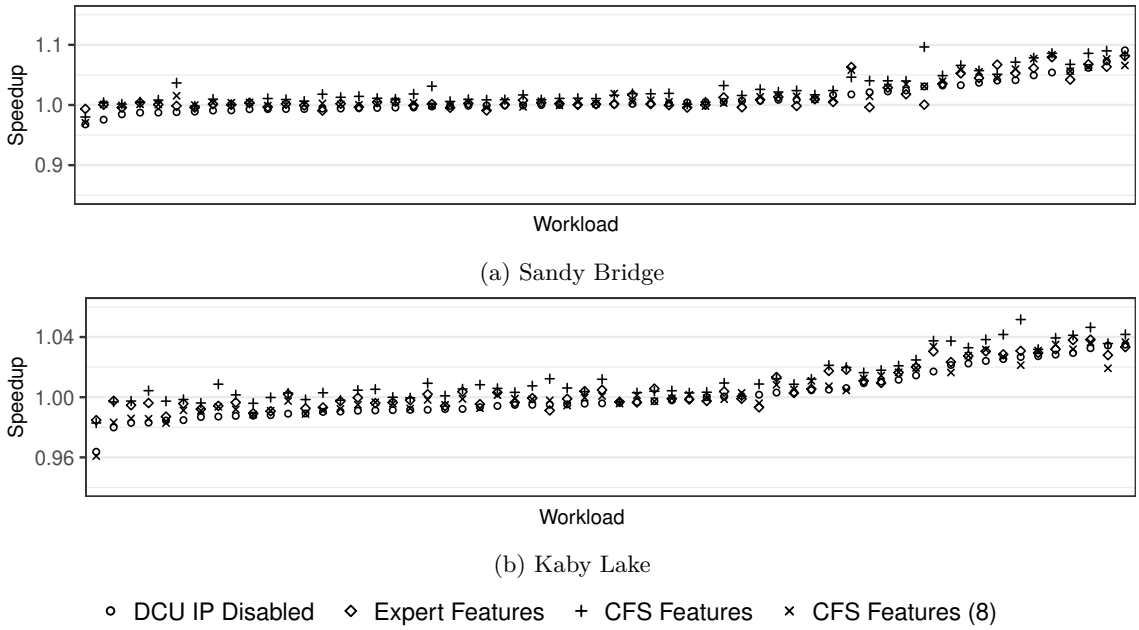
Figure 5.4: Comparison of (geometric) mean policy performance on both the Sandy Bridge and Kaby Lake for the DCU IP prefetcher.

are generally associated with higher cycle latency.

Despite their availability, performance events for measuring L1 data cache behavior are not common in the event selections for this (L1 stride) DCU IP prefetcher. On Kaby Lake, we also see L2 prefetching behavior described by the event selection (`L2_LINES_OUT:USELESS_HWPF`). This is likely due to the interaction of the DCU IP prefetcher with the other three prefetchers. To test this hypothesis, we disabled the three other prefetchers and performed CFS event selection on the DCU IP prefetcher. The result is a increase in the number of L1 cache related events on Sandy Bridge, including L1 data cache behavior (`L1D`), loads blocked due to L1 data cache operation (`L1D_BLOCKS`), and multiple events describing the behavior of write-backs from the L1 data cache to the L2 cache (`L2_L1D_WB_RQSTS`).

Model Performance

Figure 5.4 details the average workload speedup (Equation 5.2) of DCU IP prefetcher controllers compared to the static baseline (all prefetchers enabled). On average, CFS event selection outperforms domain-expert event selection on both Sandy Bridge (1.3%) and Kaby Lake (0.6%) and the performance of the eight-event CFS selection is comparable to the performance of the eight-event



(a) Sandy Bridge

(b) Kaby Lake

Figure 5.5: Workload performance for DCU IP prefetcher related policies on Sandy Bridge and Kaby Lake experimental environments, relative to the baseline.

domain-expert selection on both systems.

Individual workload performances for DCU IP prefetcher controllers, constructed from the domain-expert event selection and both CFS event selections, are given in Figures 5.5a and 5.5b. There is less opportunity on average for performance gain from dynamic prefetcher control compared to the DPL prefetcher. Regardless, dynamic DCU IP control using the domain-expert selection and full CFS selection matches or beats the performance of both the static baseline controller (all prefetchers enabled on all cores) and the static DCU IP Disabled controller (DCU IP disabled on all cores). On Kaby Lake, the controller using the eight-event CFS selection will disable the DCU IP prefetcher in almost all instances, roughly matching the performance of the DPL Disabled controller.

5.6 Related Work

Hardware prefetcher control methods utilize a number of techniques for identifying relevant performance events to measure. Liao et al. [88] and Ebrahimi et al. [45] select subsets of performance

events by hand, choosing events according to domain knowledge, e.g., the events given in Table 4.1. Especially for multi-core systems, where there may be complex interactions between hardware components in response to contention for shared resources, selecting meaningful events with direct, observable connections to the performance metric can be challenging.

In contrast, Rahman et al. [105] select performance events utilizing a distinctness heuristic to determine events whose average counts differ substantially across a set of training workloads. However, this method targets selecting static prefetcher configurations which are suitable for whole-workload execution and may not express informative differences for fine-grained, dynamic prefetcher control.

Other methods eschew performance event monitoring, and instead directly evaluate the target performance measure for each prefetcher configuration. For example, Jiménez et al. [70] directly measure the performance of several prefetcher configurations at periodic intervals to determine which configuration is advantageous to exploit for the current workload. As this method is enumerative in nature, the sampling cost grows with the number of prefetcher configurations of interest.

Correlation-Based Feature Selection has also been used for workload characterization. Yoo et al. [146] use CFS to identify performance events which are relevant to pathological performance cases, e.g., inefficient array, list, and data structure accesses which incur significant cache misses or branch mispredictions. Carefully designed micro-benchmarks were hand-labeled according to the pathological cases which they represent. Consequently, their approach has relatively clean performance data.

5.7 Discussion and Conclusion

In this work, we evaluate feature selection as a means for selecting relevant performance events for hardware memory prefetching control. While our focus was on Correlation-Based Feature Selection, alternative feature selection algorithms may also be applicable. For unstructured features, there are three classes of feature selection algorithms: filters, wrappers, and embedded methods. Filters operate directly on the data, independent of the resulting predictor or model that will be

learned using the chosen features. Many additional filters could be considered including statistical, information-theoretic, Markov Blanket based methods. In contrast, wrappers evaluate feature selections according to the performance of the model generated using those features. As the cost of model evaluation is significant, wrappers are disadvantageous for dynamic hardware prefetcher control. Embedded methods combine feature selection and model construction into a single process, often specific to an algorithm or a class of algorithms. Due to the noise inherent to multiplexing and the OFFSET transformation, finding an embedded method that is noise tolerant would be challenging.

We limit our consideration to binary decisions where a specific hardware prefetcher is either enabled or disabled on a specific core. Events selected by CFS are sensitive both to this decision, as well as many other factors including the the memory characteristics of the system and the configuration of the remaining hardware prefetchers. For example, when the DPL, ACL, and DCU prefetchers are enabled, the cache events selected for the L1 DCU IP primarily feature L2 behavior, including cache behavior related to useless prefetching. When those prefetchers are disabled however, L1 cache behavior is more prominently described. Simultaneous control of all four prefetchers will require event selections which effectively describe interactions between hardware prefetchers and the resulting effects on performance.

In order to generate the dataset on which feature selection is performed, we rely on multiplex sampling for the full set of performance events identified by `libpfm4` ($\approx$ 235 on both Sandy Bridge and Kaby Lake). However, several Intel microarchitectures feature more than a thousand available events [149]. As the number of events of interest grow, the overhead and sampling noise of multiplexing will continue to increase.

Compared to domain-expert selected features, CFS performance event selection is competitive for building effective dynamic hardware prefetching controllers on multi-core systems. On a memory-limited system, we observe a performance improvement of 5.6% compared to the static baseline, and 1.2% compared to a model using domain-expert events, when controlling the DPL prefetcher with CFS selected events. For the DCU IP prefetcher, we observe a performance improvement of 2.7% compared to the static baseline, and 1.3% compared to a model using domain-expert events. Models

constructed using a restricted set of (eight) CFS selected events have a performance comparable to the domain-expert events. In addition to providing comparable performance when trained and evaluated on prefetcher-sensitive workloads, automatic feature selection mitigates the cost of manual analysis required to identify relevant events by hand on each potential microarchitecture of interest. With the addition of feature selection, dynamic hardware prefetcher control can be learned efficiently with less need for domain expertise while providing comparable or better performance.

Chapter 6

Conclusion

This dissertation presents a general framework to modeling runtime control for system configuration and resource allocation problems which are informed by measurable statistics of microarchitecture and workload behavior, such as measurements obtained from the Performance Monitoring Unit (PMU). This work describes the mapping of two motivating applications, paging mode selection and hardware memory prefetcher utilization, to the off-policy contextual bandit, and generates effective runtime control models using random profiling data and Binary-Offset [21]. Finally, a correlation-based feature selection method is evaluated for selecting performance events from the logged random profiling data which are relevant to the the runtime control of hardware memory prefetcher utilization. The selected performance events are examined in detail and provided potential justification by appealing to available documentation.

The models resulting from the presented framework and mapping are competitive in comparison to existing approaches. For paging mode selection, the resulting model provides equivalent performance to the state-of-the-art ASP-SVM [80] method while substantially reducing the computational requirements of profiling to obtain training data, from over 24 hours in the case of ASP-SVM to less than 3 hours. For hardware memory prefetcher utilization, the resulting models are the first to provide dynamic control for hardware memory prefetchers using workload statistics. Existing runtime prefetcher control methods either determined the prefetcher configuration statically, at the beginning of execution, or dynamically, by periodically enumerating the set of configurations and measuring the resulting performance directly.

6.1 Contributions

1. An off-policy contextual bandit model for dynamic runtime control using random profiling data. The model describes a sequence of translations on the random profiling data, including PELT [78] (for phase identification) and Binary-Offset [21], which reduces the problem of dynamic runtime control to weighted classification.

2. An evaluation of two motivating examples of scaffolded difficulty: paging mode selection and hardware memory prefetcher utilization.

3. A performance event selection method, based on Correlation-Based Feature Selection [59], for identifying relevant performance events from random profiling data. Events are identified for hardware memory prefetcher utilization, and are analyzed and justified in the context of available microarchitecture documentation.

6.2 Future Work

There are a number of potential extensions to the contextual bandit framework, motivated by existing techniques in bandit literature, to describe system control with greater fidelity.

The focus of this work was model construction, however, models still required deployment and evaluation in-situ to measure model performance and effectiveness. In comparison, the cost of model evaluation now far outweighs the cost of profiling and model construction. Model evaluation can additionally be addressed from logged data using off-policy contextual bandit methods [86, 44].

When considering hardware memory prefetchers, each prefetcher on each core was modeled as an independent contextual bandit, with a binary action space, and these bandits operated concurrently with no explicit communication. Instead, the bandits communicated implicitly by observing the system-wide performance behavior of the current workload. Expanding the bandit framework to include methods which are amenable to a larger space of actions, including the Offset-Tree [21] generalization of Binary-Offset, would allow for multiple independent configurable elements, such

as multiple prefetchers on each core, to be controlled simultaneously while modeling the combined effect those configurations have on performance as a whole.

Switching between Hardware-Assisted Paging and Shadow Paging incurs a small, yet non-negligible cost as the paging table is reconstructed for the new mode. In the case where a program was transitioning from one phase of execution to another, the cost of this switch would is outweighed by the benefits of switching. However, some program phases exhibit behavior that occurs near the boundary of the learned paging mode selection classifier. During these phases, the system will continue to switch between the two paging modes in reflex to minor, inconsequential changes in performance characteristics. The margin behavior results in a significant accumulation in switching cost penalty without any improvement in performance. Switching cost was modeled by the introduction of a margin around the classifier. Measurements which fell within this margin would not trigger a switch from the current paging mode. The margin size was practitioner-designed, and selected according to manual data analysis. Ideally, the cost of switching would be directly modeled into the bandit formation, taking inspiration from (non-contextual) multi-armed bandits which include such costs [8, 20, 26].

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '06, pages 2–13, 2006.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrancy and Computation: Practice and Experiences*, 22(6):685–701, April 2010.

[3] Shipra Agrawal and Navin Goyal. Thompson sampling for contextual bandits with linear payoffs. In *Proceedings of the 30th International Conference on International Conference on Machine Learning*, ICML '13, pages 1220–1228, 2013.

[4] Shipra Agrawal and Navin Goyal. Further optimal regret bounds for thompson sampling. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, AISTATS '13, pages 99–107, 2013.

[5] Hirotugu Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6):716–723, 1974.

[6] AMD. *AMD64 Architecture Programmer's Manual: Volume 2*, July 2019.

[7] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[8] Manjari Asawa and Demonsthenis Teneketzis. Multi-armed bandits with switching penalties. *IEEE Transactions on Automatic Control*, 41(3):328–348, March 1996.

[9] Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *Proceedings of the 23rd Annual Conference on Learning Theory*, COLT '10, pages 41–53, 2010.

[10] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3:397–422, March 2003.

[11] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, May 2002.

[12] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2003.

[13] Ivan E. Auger and Charles E. Lawrence. Algorithms for the optimal identification of segment neighborhoods. *Bulletin of Mathematical Biology*, 51(1):39–54, January 1989.

[14] Reza Azimi, Michael Stumm, and Robert W. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 101–110, 2005.

[15] Chang S. Bae, John R. Lange, and Peter A. Dinda. Enhancing virtualized application performance through dynamic adaptive paging mode selection. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 255–264, 2011.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[17] Shai Ben-David, Nadav Eiron, and Philip M. Long. On the difficulty of approximately maximizing agreements. *Journal of Computer and System Sciences*, 66(3):496 – 514, 2003.

[18] James Bergstra, Nicolas Pinto, and David Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *2012 Innovative Parallel Computing*, InPar '12, pages 1–9, 2012.

[19] Ramon Bertran, Marc Gonzalez, Xavier Martorell, Nacho Navarro, and Eduard Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 147–158, 2010.

[20] Dimitris Bertsimas and José Niño-Mora. Restless bandits, linear programming relaxations, and a primal-dual index heuristic. *Operations Research*, 48(1):80–90, 2000.

[21] Alina Beygelzimer and John Langford. The offset tree for learning with partial labels. In *Proceedings of 15th International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 129–38, 2009.

[22] Alina Beygelzimer, John Langford, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandit algorithms with supervised learning guarantees. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, AISTATS '09, pages 19–26, 2011.

[23] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '08, pages 26–35, 2008.

[24] Nikhil Bhatia. Performance evaluation of Intel EPT hardware assist. Technical report, VMWare, 2009.

[25] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[26] Monica Brezzi and Tze Leung Lai. Optimal learning and experimentation in bandit problems. *Journal of Economic Dynamics and Control*, 27(1):87–108, 2002.

[27] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, August 2000.

[28] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Foundations and Trends® in Machine Learning*, 5(1):1–122, 2012.

[29] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in multi-armed bandits problems. In *Proceedings of the 20th International Conference on Algorithmic Learning Theory*, ALT '09, pages 23–37, 2009.

[30] Sébastien Bubeck, Rémi Munos, and Gilles Stoltz. Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, 412(19):1832–1852, 2011.

[31] Martin Burtscher, Byoung-Do Kim, Jeff Diamond, John McCalpin, Lars Koesterke, and James Browne. PerfExpert: An easy-to-use performance diagnosis tool for HPC applications. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, 2010.

[32] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *International Symposium on Code Generation and Optimization*, CGO '07, pages 185–197, 2007.

[33] Nicolo Cesa-Bianchi and Gabor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, New York, NY, USA, 2006.

[34] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013.

[35] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 51–61, 1992.

[36] Yong Chen, Huaiyu Zhu, and Xian-He Sun. An adaptive data prefetcher for high-performance processors. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 155–164, 2010.

[37] Wei Chu, Lihong Li, Lev Reyzin, and Robert Schapire. Contextual bandits with linear payoff functions. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, AISTATS '11, pages 208–214, 2011.

[38] Gilberto Contreras and Margaret Martonosi. Power prediction for Intel XScale®processors using performance monitoring unit events. In *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED '05, pages 221–226, 2005.

[39] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, ICPP '93, pages 56–63, 1993.

[40] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *40th IEEE Symposium on Security & Privacy*, S&P '19, 2019.

[41] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88, 2013.

[42] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 353–364, 2011.

[43] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 233–244, 2002.

[44] Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. In *Proceedings of the 28th International Conference on Machine Learning*, ICML '11, pages 1097–1104, 2011.

[45] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 316–326, 2009.

[46] Charles Elkan. Boosting and naive bayesian learning. Technical report, University of California, San Diego, 1997.

[47] Stéphane Eranian. Perfmon2: A flexible performance monitoring interface for linux. In *Ottawa Linux Symposium*, OLS '06, pages 269–288, 2006.

[48] Usama M. Fayyad and Keki B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, IJCAI '93, pages 1022–1027, 1993.

[49] Alexandra Ferreón, Radhika Jagtap, Sascha Bischoff, and Roxana Ruşitoru. Crossing the architectural barrier: Evaluating representative regions of parallel HPC applications. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '17, pages 109–120, 2017.

[50] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[51] Benoît Frénay and Michel Verleysen. Classification in the presence of label noise: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 25(5):845–869, 2014.

[52] Archana S. Ganapathi. *Predicting and Optimizing System Utilization and Performance via Statistical Machine Learning*. PhD thesis, University of California, Berkely, December 2009.

[53] M. Gerndt and M. Ott. Automatic performance analysis with periscope. *Concurrency and Computation: Practice and Experience*, 22(6):736–748, April 2010.

[54] Matthew Gillespie. Best practices for paravirtualization enhancements from Intel virtualization technology: EPT and VT-d. Technical report, Intel, 2009.

[55] John C. Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society, Series B*, 41(2):148–177, 1979.

[56] Mario Gutierrez, Saami Rahman, Dan Tamir, and Apan Qasem. Neural network methods for fast and portable prediction of CPU power consumption. In *Sixth International Green and Sustainable Computing Conference*, IGSC '15, pages 1–4, 2015.

[57] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, March 2003.

[58] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, Bernard King Smith, Julian Wang, Suresh Warrier, and David Wendt. *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8*. IBM Corporation, March 2017.

[59] Mark A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 359–366, 2000.

[60] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34(4):1–17, September 2006.

[61] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Constructing dynamic policies for paging mode selection. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, pages 72:1–72:9, 2018.

[62] Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Machine learning for fine-grained hardware prefetcher control. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 3:1–3:9, 2019.

[63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58:13–30, 1963.

[64] Lajos Horváth. The maximum likelihood method for testing changes in the parameters of normal observations. *The Annals of Statistics*, 21(2):671–680, 1993.

[65] Carla Inclán and George C. Tiao. Use of cumulative sums of squares for retrospective detection of changes of variance. *Journal of the American Statistical Association*, 89(427):913–923, 1994.

[66] Intel Corporation. *Intel 64 and IA-32 Architectures Developer's Manual: Volume 3B*, September 2016.

[67] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, April 2018.

[68] Canturk Isci and Margaret Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 93–104, 2003.

[69] Victor Jiménez, Alpher Buyuktosunoglu, Pradip Bose, Francis P. O'Connell, Francisco Cazorla, and Mateo Valero. Increasing multicore system efficiency through intelligent bandwidth shifting. In *21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 39–50, 2015.

[70] Víctor Jiménez, Roberto Gioiosa, Francisco J. Cazorla, Alper Buyuktosunoglu, Pradip Bose, and Francis P. O'Connell. Making data prefetch smarter: Adaptive prefetching on POWER7. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 137–146, 2012.

[71] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 252–263, 1997.

[72] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 135–140, 2001.

[73] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 364–373, 1990.

[74] Sham M. Kakade, Shai Shalev-Shwartz, and Ambuj Tewari. Efficient bandit algorithms for online multiclass prediction. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 440–447, 2008.

[75] Hui Kang and Jennifer L. Wong. To hardware prefetch or not to prefetch?: A virtualized environment study and core binding approach. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 357–368, 2013.

[76] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *Journal of Machine Learning Research*, 17(1): 1–42, January 2016.

[77] Muneeb Khan, Andreas Sandberg, and Erik Hagersten. A case for resource efficient prefetching in multicores. In *43rd International Conference on Parallel Processing*, ICPP '14, pages 101–110, 2014.

[78] Rebecca Killick, Paul Fearnhead, and I.A. Eckley. Optimal detection of changepoints with a linear computational cost. *Journal of the American Statistical Association*, 107:1590–1598, 2012.

[79] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1):273–324, 1997.

[80] Wei Kuang, Laura E. Brown, and Zhenlin Wang. Selective switching mechanism in virtual machines via support vector machines and transfer learning. *Machine Learning*, 101(1):137–161, 2015.

[81] Wei Kuang, Laura E. Brown, and Zhenlin Wang. Modeling cross-architecture co-tenancy performance interference, 2015.

[82] Rick Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux, 2005.

[83] Kanishka Lahiri and Subhash Kunnoth. Fast IPC estimation for performance projections using proxy suites and decision trees. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '17, pages 77–86, 2017.

[84] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing. In *2010 IEEE International Symposium on Parallel Distributed Processing*, IPDPS '10, pages 1–12, 2010.

[85] John Langford and Tong Zhang. The epoch-greedy algorithm for contextual multi-armed bandits. In *Advances in Neural Information Processing Systems 20*, NIPS, pages 817–824, 2007.

[86] John Langford, Alexander Strehl, and Jennifer Wortman. Exploration scavenging. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 528–535, 2008.

[87] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Transactions on Architecture and Code Optimization*, 9(1), March 2012.

[88] Shih-wei Liao, Tzu-Han Hung, Donald Nguyen, Chinyen Chou, Chiaheng Tu, and Hucheng Zhou. Machine learning-based prefetch optimization for data center applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 1–10, 2009.

[89] Tyler Lu, David Pal, and Martin Pal. Contextual multi-armed bandits. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, AISTATS '10, pages 485–492, 2010.

[90] David G. Luenberger and Yinyu Ye. *Linear and Nonlinear Programming*. Springer US, 3 edition, 2017.

[91] Ami Marowka. On performance analysis of a multithreaded application parallelized by different programming models using Intel VTune. In *Parallel Computing Technologies*, pages 317–331, 2011.

[92] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, 2011.

[93] Collin McCurdy, Gabriel Marin, and Jeffrey S. Vetter. Characterizing the impact of prefetching on scientific application performance. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, pages 115–135, 2014.

[94] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Computing Surveys*, 49(2):1–35, August 2016.

[95] Daniel Molka, Robert Schöne, Daniel Hackenberg, and Wolfgang E. Nagel. Detecting memory-boundedness with hardware performance counters. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, pages 27–38, 2017.

[96] Tipp Moseley, Neil Vachharajani, and William Jalby. Hardware performance monitoring for the rest of us: A position and survey. In *Proceedings of the 8th IFIP International Conference on Network and Parallel Computing*, NPC '11, pages 293–312, 2011.

[97] Hitoshi Nagasaka, Naoya Maruyama, Akira Nukada, Toshio Endo, and Satoshi Matsuoka. Statistical power modeling of gpu kernels using performance counters. In *International Conference on Green Computing*, pages 115–122, 2010.

[98] Priya Nagpurkar, Michael Hind, Chandra Krintz, Peter F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *International Symposium on Code Generation and Optimization (CGO'06)*, CGO '06, pages 121–123, 2006.

[99] Tan T. Nguyen and Scott Sanner. Algorithms for direct 0-1 loss optimization in binary classification. In *Proceedings of the 30th International Conference on Machine Learning*, ICML '13, pages 1085–1093, 2013.

[100] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. DICER: Diligent cache partitioning for efficient workload

consolidation. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 15:1–15:10, 2019.

[101] Cristobal Ortega, Miquel Moreto, Marc Casas, Ramon Bertran, Alper Buyuktosunoglu, Alexandre E. Eichenberger, and Pradip Bose. libPRISM: An intelligent adaptation of prefetch and SMT levels. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 28:1–28:10, 2017.

[102] ElMoustapha Ould-Ahmed-Vall, Kshitij A. Doshi, Charles Yount, and James Woodlee. Characterization of SPEC CPU2006 and SPEC OMP2001: Regression models and their transferability. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '08, pages 179–190, 2008.

[103] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, pages 24–33, 1994.

[104] J. R. Quinlan. Bagging, boosting, and c4.s. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, AAAI '96, pages 725–730, 1996.

[105] Saami Rahman, Martin Burtscher, Ziliang Zong, and Apan Qasem. Maximizing hardware prefetch effectiveness with machine learning. In *Proceedings of the 17th International Conference on High Performance Computing and Communications*, pages 383–389, 2015.

[106] Luis Ramos, José Briz, Pablo Ibáñez, and Víctor Viñals-Yufera. Multi-level adaptive prefetching based on performance gradient tracking. *Journal of Instruction-Level Parallelism*, 13:1–14, 2011.

[107] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, August 1952.

[108] James M. Robins, Andrea Rotnitzky, and Lue Ping Zhao. Estimation of regression coefficients when some regressors are not always observed. *Journal of the American Statistical Association*, 89(427):846–866, 1994.

[109] Shuvabrata Saha. A multi-objective autotuning framework for the java virtual machine. Master's thesis, Texas State University, May 2016.

[110] Florian T. Schneider, Mathias Payer, and Thomas R. Gross. Online optimizations driven by hardware performance monitoring. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 373–382, 2007.

[111] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, 1978.

[112] A. J. Scott and M. Knott. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, 30(3):507–512, 1974.

[113] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '04, pages 165–176, 2004.

[114] Timothy Sherwood and Brad Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, University of California San Diego, 1999.

[115] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 42–53, 2000.

[116] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, 2001.

[117] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 45–57, 2002.

[118] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, 2003.

[119] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 336–349, 2003.

[120] Karan Singh, Major Bhadauria, and Sally A. McKee. Real time power estimation and thread scheduling via performance counters. *SIGARCH Computer Architecture News*, 37(2):46–55, July 2009.

[121] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3):1:1–1:29, 2011.

[122] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. IBM POWER8 multicore server processor. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[123] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 63–74, 2007.

[124] Standard Performance Evaluation Corporation. SPEC CPU 1995. `www.spec.org/cpu95/`, 1995.

[125] Standard Performance Evaluation Corporation. SPEC CPU 2006. `www.spec.org/cpu2006/`, 2006.

[126] Standard Performance Evaluation Corporation. SPEC CPU 2017. `www.spec.org/cpu2017/`, 2017.

[127] Alexander L. Strehl, Chris Mesterharm, Michael L. Littman, and Haym Hirsh. Experience-efficient learning in associative bandit problems. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 889–896, 2006.

[128] Alexander L. Strehl, John Langford, Lihong Li, and Sham M. Kakade. Learning from logged implicit exploration data. In *Proceedings of the 23rd International Conference on Neural Information Processing Systems*, NIPS '10, pages 2217–2225, 2010.

[129] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1998.

[130] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[131] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. DejaVu: Accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 423–436, 2012.

[132] Vish Viswanathan. Disclosure of h/w prefetcher control on some intel processors. Technical report, Intel, 2014.

[133] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[134] Chih-Chun Wang, Sanjeev R. Kulkarni, and H. Vincent Poor. Bandit problems with side observations. *IEEE Transactions on Automatic Control*, 50(3):338–355, 2005.

[135] Chih-Chun Wang, Sanjeev R. Kulkarni, and H. Vincent Poor. Arbitrary side observations in bandit problems. *Advances in Applied Mathematics*, 34(4):903–938, 2005.

[136] Xiaolin Wang, Jiarui Zang, Zhenlin Wang, Yingwei Luo, and Xiaoming Li. Selective hardware/software memory virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 217–226, 2011.

[137] Michael Woodroofe. A one-armed bandit problem with a concomitant variable. *Journal of the American Statistical Association*, 74(368):799–806, 1979.

[138] Carole-Jean Wu and Margaret Martonosi. Characterization and dynamic mitigation of intra-application cache interference. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 2–11, 2011.

[139] Xingfu Wu and Valerie Taylor. Utilizing hardware performance counters to model and optimize the energy and performance of large scale scientific applications on power-aware supercomputers. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW '16, pages 1180–1189, 2016.

[140] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic cache allocation with partial sharing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[141] Yaocheng Xiang, Chencheng Ye, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. EMBA: Efficient memory bandwidth allocation to improve performance on Intel commodity processor. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 16:1–16:12, 2019.

[142] Jun Xiao, Andy D. Pimentel, and Xu Liu. CP-pf: A prefetch aware LLC partitioning approach. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 17:1–17:10, 2019.

[143] Hao Xu, Qingsen Wang, Shuang Song, Lizy Kurian John, and Xu Liu. Can we trust profiling results? understanding and fixing the inaccuracy in modern profilers. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 284–295, 2019.

[144] Min Yang, Linli Xu, Martha White, Dale Schuurmans, and Yao liang Yu. Relaxed clipping: A global training method for robust regression and classification. In *Advances in Neural Information Processing Systems 23*, NIPS, pages 2532–2540, 2010.

[145] Xulei Yang, Qing Song, and Aize Cao. Weighted support vector machine for data classification.

In *Proceedings of the 2005 IEEE International Joint Conference on Neural Networks*, volume 2, pages 859–864, 2005.

[146] Wucherl Yoo, Kevin Larson, Lee Baugh, Sangkyum Kim, and Roy H. Campbell. ADP: Automated diagnosis of performance pathologies using hardware events. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 283–294, 2012.

[147] Jia Yuan Yu and Shie Mannor. Piecewise-stationary bandit problems with side observations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1177–1184, 2009.

[148] Bianca Zadrozny, John Langford, and Naoki Abe. Cost-sensitive learning by cost-proportionate example weighting. In *Proceedings of the Third IEEE International Conference on Data Mining*, ICDM '03, pages 435–442, 2003.

[149] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 14:1–14:9, 2016.

[150] Nancy R. Zhang and David O. Siegmund. A modified bayes information criterion with applications to the analysis of comparative genomic hybridization data. *Biometrics*, 63(1):22–32, 2007.

Appendix A

Copyright Permission

The following documents detail permission from the ACM to reprint the materials herein.

Chapter 3 contains material previously published in the Proceedings of the 47th International Conference on Parallel Processing (ICPP '18) [61]:

> Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Constructing dynamic policies for paging mode selection. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP '18, pages 72:1–72:9, 2018, doi:10.1145/3225058.3225082.

Chapter 4 contains material previously published in the Proceedings of the 48th International Conference on Parallel Processing (ICPP '19) [62]:

> Jason Hiebel, Laura E. Brown, and Zhenlin Wang. Machine learning for fine-grained hardware prefetcher control. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, pages 3:1–3:9, 2019, doi:10.1145/3337821.3337854.

ACM Copyright and Audio/Video Release

Title of the Work: Constructing Dynamic Policies for Paging Mode Selection
Submission ID:pap212
Author/Presenter(s): Jason Hiebel (Michigan Technological University);Laura E. Brown (Michigan Technological University);Zhenlin Wang (Michigan Technological University)
Type of material: Full Paper

Publication and/or Conference Name: 47th International Conference on Parallel Processing Proceedings

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work.

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\copyrightyear{2018}
\acmYear{2018}
\setcopyright{acmcopyright}
\acmConference[ICPP 2018]{47th International Conference on Parallel
Processing}{August 13--16, 2018}{Eugene, OR, USA}
\acmBooktitle{ICPP 2018: 47th International Conference on Parallel
Processing, August 13--16, 2018, Eugene, OR, USA}
\acmPrice{15.00}
\acmDOI{10.1145/3225058.3225082}
\acmISBN{978-1-4503-6510-9/18/08}
```

ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.
Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\CopyrightYear{2018}
\setcopyright{acmcopyright}
\conferenceinfo{ICPP 2018,}{August 13--16, 2018, Eugene, OR, USA}
\isbn{978-1-4503-6510-9/18/08}\acmPrice{$15.00}
\doi{https://doi.org/10.1145/3225058.3225082}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225082

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images
If your paper includes images that were created for any purpose other than this paper and to which you or your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.
◉ We/I do not have any artistic images.
○ We/I have any artistic images.

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

Funding Agents

1. National Science Foundation award number(s): CSR1422343, CSR1618384

DATE: **05/22/2018** sent to jshiebel@mtu.edu at **10:05:50**

ACM Copyright and Audio/Video Release

Title of the Work: Machine Learning for Fine-Grained Hardware Prefetcher Control
Submission ID:pap223
Author/Presenter(s): Jason Hiebel (Michigan Technological University);Laura E. Brown (Michigan Technological University);Zhenlin Wang (Michigan Technological University)
Type of material: Full Paper

Publication and/or Conference Name: 48th International Conference on Parallel Processing Proceedings

I. Copyright Transfer, Reserved Rights and Permitted Uses

* Your Copyright Transfer is conditional upon you agreeing to the terms set out below.

Copyright to the Work and to any supplemental files integral to the Work which are submitted with it for review and publication such as an extended proof, a PowerPoint outline, or appendices that may exceed a printed page limit, (including without limitation, the right to publish the Work in whole or in part in any and all forms of media, now or hereafter known) is hereby transferred to the ACM (for Government work, to the extent transferable) effective as of the date of this agreement, on the understanding that the Work has been accepted for publication by ACM.

Reserved Rights and Permitted Uses

(a) All rights and permissions the author has not granted to ACM are reserved to the Owner, including all other proprietary rights such as patent or trademark rights.

(b) Furthermore, notwithstanding the exclusive rights the Owner has granted to ACM, Owner shall have the right to do the following:

(i) Reuse any portion of the Work, without fee, in any future works written or edited by the Author, including books, lectures and presentations in any and all media.

(ii) Create a "Major Revision" which is wholly owned by the author

(iii) Post the Accepted Version of the Work on (1) the Author's home page, (2) the Owner's institutional repository, (3) any repository legally mandated by an agency funding the research on which the Work is based, and (4) any non-commercial repository or aggregation that does not duplicate ACM tables of contents, i.e., whose patterns of links do not substantially duplicate an ACM-copyrighted volume or issue. Non-commercial repositories are here understood as repositories owned by non-profit organizations that do not charge a fee for accessing deposited articles and that do not sell advertising or otherwise profit from serving articles.

(iv) Post an "Author-Izer" link enabling free downloads of the Version of Record in the ACM Digital Library on (1) the Author's home page or (2) the Owner's institutional repository;

(v) Prior to commencement of the ACM peer review process, post the version of the Work as submitted to ACM ("Submitted Version" or any earlier versions) to non-peer reviewed servers;

(vi) Make free distributions of the final published Version of Record internally to the Owner's employees, if applicable;

(vii) Make free distributions of the published Version of Record for Classroom and Personal Use;

(viii) Bundle the Work in any of Owner's software distributions; and

(ix) Use any Auxiliary Material independent from the Work. (x) If your paper is withdrawn before it is published in the ACM Digital Library, the rights revert back to the author(s).

When preparing your paper for submission using the ACM TeX templates, the rights and permissions information and the bibliographic strip must appear on the lower left hand portion of the first page.

The new ACM Consolidated TeX template Version 1.3 and above automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.

Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\copyrightyear{2019}
\acmYear{2019}
\setcopyright{acmcopyright}
\acmConference[ICPP 2019]{48th International Conference on Parallel
Processing}{August 5--8, 2019}{Kyoto, Japan}
\acmBooktitle{48th International Conference on Parallel Processing (ICPP 2019),
August 5--8, 2019, Kyoto, Japan}
\acmPrice{15.00}
\acmDOI{10.1145/3337821.3337854}
\acmISBN{978-1-4503-6295-5/19/08}
```

ACM TeX template .cls version 2.8, automatically creates and positions these text blocks for you based on the code snippet which is system-generated based on your rights management choice and this particular conference.
Please copy and paste the following code snippet into your TeX file between \begin{document} and \maketitle, either after or before CCS codes.

```
\CopyrightYear{2019}
\setcopyright{acmcopyright}
\conferenceinfo{ICPP 2019,}{August 5--8, 2019, Kyoto, Japan}
\isbn{978-1-4503-6295-5/19/08}\acmPrice{$15.00}
\doi{https://doi.org/10.1145/3337821.3337854}
```

If you are using the ACM Microsoft Word template, or still using an older version of the ACM TeX template, or the current versions of the ACM SIGCHI, SIGGRAPH, or SIGPLAN TeX templates, you must copy and paste the following text block into your document as per the instructions provided with the templates you are using:

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00

https://doi.org/10.1145/3337821.3337854

NOTE: Make sure to include your article's DOI as part of the bibstrip data; DOIs will be registered and become active shortly after publication in the ACM Digital Library. Once you have your camera ready copy ready, please send your source files and PDF to your event contact for processing.

☑ A. Assent to Assignment. I hereby represent and warrant that I am the sole owner (or authorized agent of the copyright owner(s)), with the exception of third party materials detailed in section III below. I have obtained permission for any third-party material included in the Work.

☐ B. Declaration for Government Work. I am an employee of the National Government of my country and my Government claims rights to this work, or it is not copyrightable (Government work is classified as Public Domain in U.S. only)

Are any of the co-authors, employees or contractors of a National Government? ○ Yes ◉ No

---

II. Permission For Conference Recording and Distribution

* Your Audio/Video Release is conditional upon you agreeing to the terms set out below.

I hereby grant permission for ACM to include my name, likeness, presentation and comments in any and all forms, for the Conference and/or Publication.

I further grant permission for ACM to record and/or transcribe and reproduce my presentation as part of the ACM Digital Library, and to distribute the same for sale in complete or partial form as part of an ACM product on CD-ROM, DVD, webcast, USB device, streaming video or any other media format now or hereafter known.

I understand that my presentation will not be sold separately as a stand-alone product without my direct consent. Accordingly, I give ACM the right to use my image, voice, pronouncements, likeness, and my name, and any biographical material submitted by me, in connection with the Conference and/or Publication, whether used in excerpts or in full, for distribution described above and for any associated advertising or exhibition.

Do you agree to the above Audio/Video Release? ◉ Yes ○ No

III. Auxiliary Material

Do you have any Auxiliary Materials? ○ Yes ◉ No

IV. Third Party Materials

In the event that any materials used in my presentation or Auxiliary Materials contain the work of third-party individuals or organizations (including copyrighted music or movie excerpts or anything not owned by me), I understand that it is my responsibility to secure any necessary permissions and/or licenses for print and/or digital publication, and cite or attach them below.

◉ We/I have not used third-party material.
○ We/I have used third-party materials and have necessary permissions.

V. Artistic Images

If your paper includes images that were created for any purpose other than this paper and to which you or

your employer claim copyright, you must complete Part V and be sure to include a notice of copyright with each such image in the paper.

(●) We/I do not have any artistic images.
( ) We/I have any artistic images.

VI. Representations, Warranties and Covenants

The undersigned hereby represents, warrants and covenants as follows:

(a) Owner is the sole owner or authorized agent of Owner(s) of the Work;

(b) The undersigned is authorized to enter into this Agreement and grant the rights included in this license to ACM;

(c) The Work is original and does not infringe the rights of any third party; all permissions for use of third-party materials consistent in scope and duration with the rights granted to ACM have been obtained, copies of such permissions have been provided to ACM, and the Work as submitted to ACM clearly and accurately indicates the credit to the proprietors of any such third-party materials (including any applicable copyright notice), or will be revised to indicate such credit;

(d) The Work has not been published except for informal postings on non-peer reviewed servers, and Owner covenants to use best efforts to place ACM DOI pointers on any such prior postings;

(e) The Auxiliary Materials, if any, contain no malicious code, virus, trojan horse or other software routines or hardware components designed to permit unauthorized access or to disable, erase or otherwise harm any computer systems or software; and

(f) The Artistic Images, if any, are clearly and accurately noted as such (including any applicable copyright notice) in the Submitted Version.

☑ I agree to the Representations, Warranties and Covenants

Funding Agents

1. National Science Foundation award number(s): CSR1618384,CSR1422342

DATE: **06/05/2019** sent to jshiebel@mtu.edu at **08:06:44**