Machine Learning for Machines: Data-Driven Performance Tuning at Runtime Using Sparse Coding

Machine Learning for Machines: Data-Driven Performance Tuning at Runtime Using Sparse Coding

A dissertation presented
by

Stephen John Tarsa

to
The School of Engineering and Applied Sciences
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy
in the subject of

Computer Science

Harvard University
Cambridge, Massachusetts
October 2014

Dissertation Advisor Author

Professor H. T. Kung **Stephen John Tarsa**

Machine Learning for Machines: Data-Driven Performance Tuning at Runtime Using Sparse Coding

Abstract

We develop methods for adjusting devices to runtime conditions based on system-state predictions. Our approach statistically models performance data collected by actively probing conditions such as wireless link quality or leveraging infrastructure for device-monitoring such as hardware event counters. We use state predictions to tune devices on-the-fly, and make changes to transmission schedule in wireless devices, voltage and frequency in circuits, and data placement in storage systems. In highly-variable everyday use-cases, large performance gains result not from designing new protocols or system configurations, but from more-judiciously applying those that already exist.

This thesis presents a state-modeling framework based on sparse feature representation. It is applied in diverse application scenarios to data representing:

1. Packet loss over wireless links
2. Circuit performance counters collected during user-driven workloads
3. Access-pattern statistics measured from data-center storage systems

Our framework uses unsupervised clustering to discover latent statistical structure in large datasets. We exploit this stable structure to reduce overfitting in supervised learning models like Support Vector Machine (SVM) classifiers and Classification and Regression Trees (CART) trained on small datasets. As a result, we can capture transient predictive statistics that change with wireless environment, circuit workload, and storage application. Given the magnitude of performance improvements and the potential economic opportunity, we hope that this work becomes the foundation for a broad investigation into on-platform data-driven device optimization, dubbed *Machine Learning for Machines (MLM).*

Contents

List of Figures

List of Tables

Citations to Previously Published Work

Portions of Chapter 2 have appeared in the following:

S. J. Tarsa, T.-H. Lin, and H. T. Kung, “Performance Gains in Conjugate Gradient Computations with Linearly Connected GPU Multiprocessors”, in *USENIX HotPar* 2012.

T.-H. Lin, S. J. Tarsa, and H. T. Kung, “Parallelization Primitives for Dynamic Sparse Computations”, in *USENIX HotPar* 2013.

Portions of Chapter 3 have appeared in the following:

H. T. Kung, C.-K. Lin, T.-H. Lin, S. J. Tarsa, D. Vlah, et al. “A Location Dependent Runs-and-Gaps Model for Predicting TCP Performance Over A UAV Wireless Channel”, in *IEEE MILCOM* 2010.

S. J. Tarsa and H. T. Kung, “Hierarchical Sparse Coding for Wireless Link Prediction in an Airborne Scenario”, in *IEEE MILCOM* 2013.

Chapter 3 includes follow-up work suggested by the dissertation committee that has been submitted for publication. This work includes extensive input from Marcus Comiter.

Portions of Chapter 4 have appeared in the following:

S. J. Tarsa and H. T. Kung, “Workload Prediction for Adaptive Power Scaling Using Deep Learning”, in *IEEE ICICDT* 2014.

Portions of Chapter 5 include input from Mike Mesnier at Intel.

Acknowledgments

I feel inexpressibly lucky to have found a field that I find so fascinating and rewarding at a young age. I am grateful to each of the teachers who have challenged and encouraged me since then, including my parents, Robert Webster, Errin Fulp, and especially H.T. Kung. Working with H.T. may end up being the greatest opportunity I ever get, and it has changed all aspects of my thinking: intellectual, professional, and personal.

To my committee: H.T., Margo, Shlomo, and Yue Lu; to our research collaborators at AFRL and Intel; to the professors and students of the Harvard CS department; and most importantly to C.K. Lin, Tsung-Han Lin, Dario Vlah, Kevin Chen, and the residents of MD 207: thanks for making these such happy and energetic years, while offering a unique, broad, and historically significant perspective on the field I find so interesting.

To my parents, sister, brother, in-laws, nieces, & nephew(s); to my friends and loved ones, from holidays, to hospital rooms, to hockey games in single-digit temperatures: thank-you for your kindness and love. You have put me in deep karmic debt.

The Harvard community is an exciting and colorful place. It has offered me more opportunities in six years than any one person deserves in a lifetime. For symphonies in Havana concert halls, midnight star-gazing among nomads in the Jordanian desert, surfing trips to remote beaches in Taiwan, and summertime adventures up-and-down the west coast of the US, I will be forever nostalgic. Especially to the members of the Harvard-Radcliffe Orchestra and all of the friends I made throughout Cambridge: thanks for making life so much fun.

Dedicated to my parents, with apologies for breaking the family computer so often growing up.

Chapter 1

Introduction

1.1 Motivating Scenario: Wireless Links in Daily Life

Many computer systems struggle to perform efficiently when runtime conditions like network state, workload, and user-demands change continually. This is because it is impractical to optimize a single device for all possible use-cases at design-time. Even with extensive research and planning, fluid runtime variations inevitably create unanticipated scenarios that break static design assumptions. In general, the result is a tradeoff between precisely-tuned systems targeted to a subset of users, or general-use systems that underperform in individual cases.

The network-transport protocol used by mobile phones during the morning commute to Harvard provides one representative example. As shown in Figure 1.1, when a user travels to campus from a nearby suburb, their smartphone must contend with huge variations in link quality over a short period of time. At the outset, conditions in open-space rural environments are affected primarily by line-of-sight occlusions

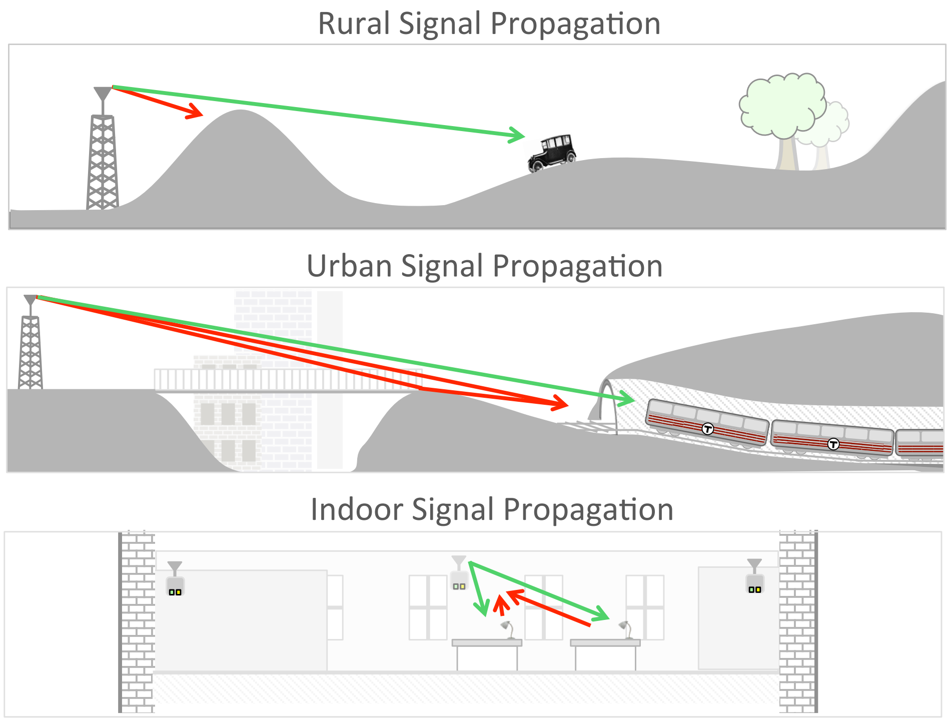


Figure 1.1: Drivers of wireless packet loss change throughout everyday scenarios. In open-space rural environments, occlusions from terrain and radio range cause transmissions to gradually degrade over space. In urban environments, multipath reflections and fast-changing line-of-sight occlusions cause rapid fluctuations in link quality. And in indoor scenarios, densely-placed access points compensate for hard-to-predict dead-zones, but interference from other users affects signal quality.

from terrain, and gradual degradation due to radio range limitations. Once aboard a subway train, that same device must address intermittent packet losses due to fast-changing multipath interference as the train passes through tunnels and over bridges, surrounded by dense ground structure. And finally, upon arrival at the office, signal propagation in indoor 802.11 (WiFi) networks is highly asymmetric, since congestion and multipath interference have a stronger effect on fixed-position access points than isolated mobile nodes.

In this everyday use-case, smartphones rely on TCP for congestion control and reliable data transport when supporting applications like web surfing. Baseline TCP interprets packet loss as a congestion signal, and responds by throttling sending rate to share the link with others. However, given the variety of delay, congestion, and signal propagation characteristics found across networks, this simple assumption is not appropriate in all cases. For example, in asymmetric office-WiFi networks, packets lost traveling downstream to an end-host are more likely to signal network congestion than those lost along flakier wireless links traveling toward an access point.

Since its introduction, researchers have defined more than thirty-five TCP variants tuned for different network scenarios. For example, the Linux default TCP CUBIC targets high-latency high-bandwidth networks. It uses a fast-retransmit option to fix isolated delivery failures, but assumes that lower layers compensate for link loss with automatic-repeat requests (ARQ) and forward-error-correction (FEC). This configuration is sufficient to provide high throughput and fair share in rural and office environments, but the transient link outages of an urban subway track will undermine connection performance. As a result, a smartphone's network throughput will fluctuate wildly during the morning commute.

The variable performance of protocols like TCP in our motivating scenario highlights a system-design limitation: the smartphone's poor network throughput is not due to a broken protocol, but instead arises when static assumptions on link conditions are applied across variable environments. For wireless, rather than inventing a new TCP variant, we demonstrate performance gains in this work by more-carefully applying those that already exist.

Beyond wireless, we show that analogous issues arise in circuit power-management and object-based storage systems. Anecdotally, the ARM A9 processor embedded in warzone communication devices like the Boeing Black is also used in Google Glass prototypes worn by surgeons to film operations [1]. But regardless of chip workload and user needs, a single set of heuristics govern the frequency and voltage scaling rules that drive power dissipation [2]. Likewise, Amazon's datacenters serve both the Christmas shopping rush and the photo rendering computations of Instagram when extra compute cycles are leased to third-parties [3]. But in their current state, commercial datacenters are so over-provisioned for worst-case demand that randomly-assigned jobs run only a minor risk of resource contention in typical conditions [4].

To address the need for more-flexible configuration, we present system architectures that actively measure and model runtime conditions to tune performance. We capitalize on the prevalence of low-level performance counters on off-the-shelf devices, and focus on building high-quality predictive models from their data in uncontrolled real-world measurement conditions. To this end, we adapt tools from the machine learning literature that exploit *sparse feature representations*. For wireless communication, circuit power-management, and object-based storage systems, sparse feature modeling produces useful state predictions when static or expert-derived models struggle to capture runtime variability. As a result, our systems adjust to runtime conditions that change over time, across machines, and among users – a specificity not possible in their static or hand-optimized counterparts.

1.2 Opportunity for On-the-Fly Performance Tuning

For many platforms, popular adoption means that a single design must satisfy vastly different user demands. Heterogeneity is assured, yet current systems are rigid to a surprising degree. However, this is not due to a lack of on-the-fly reconfiguration mechanisms. Over time, manufacturing improvements have reduced computation and memory costs by orders of magnitude, making alternate data pathways and powerful microcontrollers affordable. For example, power-saving reconfiguration mechanisms exist at the circuit-, architecture-, and component-level on devices from AMD, Intel, nVidia, and Cisco. Platforms like the IBM POWER8 CPU control power configuration with the full computational capabilities of an early-2000s CPU. Within larger datacenters, software-driven storage systems like CEPH [5], the Google File System [6], and Lustre [7] manage cheap hardware to provide configuration options for many different workloads. In the meantime, the widespread adoption of virtualization has made low-level software control possible in consumer devices, datacenters, network hardware, storage systems, and soon across System on Chip (SoC) components in designs like the Intel SoFIA.

Among these devices, instrumentation for capturing runtime system state is extensive. Figure 1.2 shows the number of on-board hardware performance counters for popular CPU, GPU, and network interface cards over time. Driven by the need to debug complicated designs, architects have naturally increased the density of counters for events like packet receptions along links, cache hits and misses in a chip's instruction table, and block-level I/O statistics. In one case, we found that an SoC

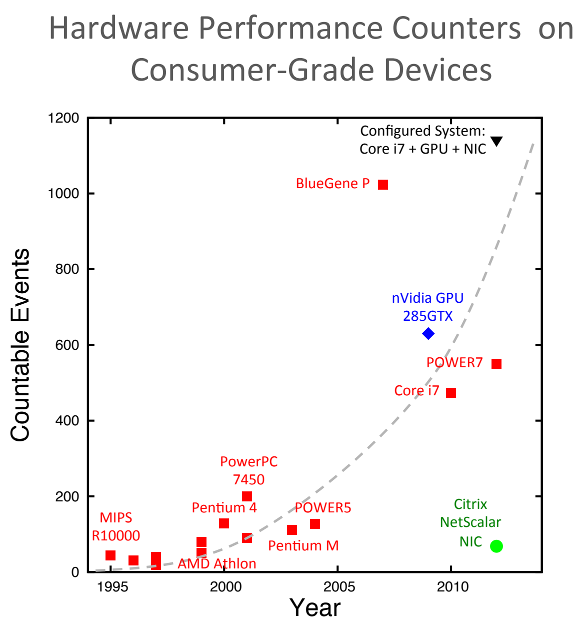


Figure 1.2: The number and diversity of hardware performance counters instrumenting off-the-shelf devices has naturally increased over time. This trend is driven by designers' desire for high-quality debugging information, and has created a rich data source for state-modeling at runtime.

had been instrumented with tens of thousands of hardware performance counters, even though only a few could be accessed in the final design.

Despite devices' low-level versatility and support for fine-grained performance-monitoring, we observe that designers lack a portable, automated approach for modeling system states. Through collaboration with industry, we have seen that brittle strategies for matching device designs to runtime conditions dominate, such as one-off expert analysis or spreadsheet-driven modeling. However, using the rich data generated by performance counters, we show that flexible automated modeling is possible. Advances in machine learning have greatly improved techniques for computing stable statistics from high dimensional datasets acquired in uncontrolled conditions. We will use these tools to capture latent statistical relationships that act as predictive signatures of system state, despite non-Gaussian measurement noise.

This thesis demonstrates automated prediction models for wireless, circuit, and storage devices, as well as performance optimizations that use them. In Chapter 3, we predict temporary wireless link outages to hold TCP packets when successful data delivery is unlikely. In Chapter 4, we predict patterns in architecture-level event counters that arise in circuits during user-driven workloads. By anticipating brief lulls in instruction throughput, chips can scale back voltage and frequency to save power without affecting computation speed. And in Chapter 5, we learn patterns of conflicting I/O access requests in object-based storage systems that colocate data from multiple users. With the resulting performance models, automated placement policies can avoid configurations that cause disproportionately bad performance due to effects like head-of-line blocking.

1.3 Machine Learning for Machines (MLM)

Machine learning techniques implement data-driven models whose parameters are algorithmically estimated from measurements of a target system. This is useful when measurement data captures statistical relationships that reflect underlying system behaviors. Modeling these *hidden* or *latent* relationships is an indirect method of mapping out complicated or unknown processes, for example to make predictions. The field incorporates insights from statistics, signal processing, optimization, artificial intelligence, computational biology, and parallel computing. Application vehicles in the literature draw from computer vision, speech recognition, natural language processing, finance, medicine, and astronomy, among others.

Learning techniques fall into two categories: *unsupervised* and *supervised.* Unsupervised methods discover latent structure from large datasets. Examples include the clustering algorithms of Section 2.2.1.1 that identify patterns formed by similar recurring measurements. By transforming raw data into a representation that reflects this more-meaningful structure, unsupervised learning can implement powerful pre-processing functions like noise removal, or support visualization within a larger workflow such as a Knowledge Discovery in Databases (KDD) process [8].

In contrast, supervised learning tools such as the Support Vector Machine (SVM) classifiers of Section 2.2.2.1 infer unknown information from measurements. For training, they require both measurement vectors and associated examples of a process that is unobservable at acquisition time. They are then trained to reproduce the relationships between these two types of inputs, and generalize them for new measurements. For example, SVMs associate measurements with class *labels*, and compute a separating plane between different classes. Once trained, the SVM can then infer the class label of a new measurement by comparing it to the separator. As described in Section 2.2.2, supervised learning techniques are often limited by the cost of acquiring labels for large datasets.

When paired together, these two approaches complement each other. This is because latent statistical relationships are generally stable over time, and can be used to combat variation in the statistics of limited-sized labeled datasets. For example, the performance-counter data from a circuit reflects patterns created by architecture-level operations. After an SVM has been trained to identify a particular workload based on performance-counter patterns, new measurement variants may arise due to unforeseen errors like missing data. For variations not captured by the SVM's training set, classification results will be uncertain. However, if raw measurements are first matched to templates of common patterns found with unsupervised learning, such distortions will be removed before classification. In this way, unsupervised learning leverages past data to improve the statistical stability of supervised learning techniques.

Machine learning's successes have produced a variety of tools useful for modeling state from performance data. In Chapter 2, we present our framework and describe how to manage tradeoffs of training time, computational complexity, and statistical accuracy to implement real-time state prediction on-device.

1.3.1 Application Constraints for System State Models

When building state models from the performance data of wireless, circuit, and storage devices, we found several common characteristics across domains. We present them here to motivate the tradeoffs discussed in Chapter 2:

MLM(1) **Localized Statistical Stability -** In each domain, our data shows strong temporal effects as exogenous factors like physical environment, program code, and user behavior drive system state. The statistics of predictive patterns are often valid for a short time period, and data acquisition does not yield an unbiased sample of trends experienced over a platform's lifetime. For wireless links, we find that patterns in packet loss data change across rural, urban, and indoor environments. For circuit performance counters, sequences of architecture-level events are driven by the current workload, and change drastically when program code shifts from a cyclic compute-driven video game to I/O-bound web surfing. In the case of storage systems, access patterns change as datacenter customers come and go, or even when an external network is reconfigured to buffer data more efficiently. As a result, we find that good predictions of system state are governed by the statistics of local contexts.

MLM(2) **Constrained Acquisition of Labeled Training Data -** Labeled data acquisition for supervised learning techniques is expensive relative to application lifetimes in all of our scenarios. For wireless, the fine-grained measurements required to associate packet loss with link outages are limited by power, bandwidth, and the financial cost of transmission over commercial networks. For circuits, workload phases lasting just seconds provide only a few observations of instruction throughput lulls. And in storage systems, performance measurements require tens of seconds for devices to be brought to steady state, making it infeasible to observe all possible conflicting access patterns. In all cases, labeled training data is limited, and long collection times dilute net performance.

MLM(3) **Utility of Domain Knowledge to Mitigate Complexity -** When data acquisition and model complexity are limited on-platform, pure data-driven models cannot capture all important statistical trends; as a result, we show that bootstrapping learning techniques with expert knowledge produces major gains in training efficiency and prediction accuracy. For wireless, we compare patterns of packet loss identified by clustering against known network topology to find that queuing delays affect predictive signatures. When we include these effects explicitly in our models, predictions transfer more-effectively among networks to reduce re-training requirements. For storage systems, we compare clustering results to workload labels assigned by a human-in-the-loop to find previously unknown similarities. Combining workload definitions reduces the complexity that must be managed by system administrators or captured by supervised learners. In order to facilitate this bi-directional flow of information between learning tools and experts, we focus on models that can be easily interpreted.

In Section 2.2.3, we use these observations to guide our choice of learning tools. For example, despite their eye-catching results in commercial systems, we avoid convolutional neural networks. This is because large models with many parameters require too much labeled training data to quickly adapt to changing local statistics. We must also consider both accuracy *and* computational complexity, as when we encode measurements in terms of their latent structure. This leads to a platform-specific tradeoff that balances accuracy against computational resources and power consumption. Finally, we find through industry collaboration that machine learning tools with a straightforward interpretation are preferred for capital-intensive projects. Given the cost of chip fabrication or the expenses associated with day-to-day datacenter operation, these concerns reflect a necessary level of conservatism that informs modeling choices in commercial settings.

1.4 Thesis Roadmap and Summary of Applications

Chapter 2 next presents background on mechanisms for on-the-fly device reconfiguration. A review of machine learning tools useful for state modeling then discusses tradeoffs relevant to wireless, circuit, and storage systems. Chapters 3, 4, and 5, employ these tools in the following applications:

Chapter 3: A Protocol- and Environment-Aware Wireless Link Layer - We model packet receptions along wireless links in everyday scenarios. When users travel through harsh urban and indoor environments, links are unstable and temporary outages common. As a result, connections that rely on TCP, link-layer retransmissions, and physical-layer FEC achieve only minimal throughput. We build predictors that anticipate temporary outages by learning signatures of environment effects like entry into subway tunnels, passage through dead zones, or elevator doors closing. Our system, State Informed Link-Layer Queuing (SILQ), holds TCP packets temporarily when delivery is likely to fail. By stabilizing performance in higher layers of the network stack, this method improves throughput and reduces network-performance variation.

Chapter 4: Workload Modeling for Predictive DVFS in Low-Power Circuits - We model hardware performance counters collected from an ARMv7a chip. As user-driven workloads like web surfing create patterns at the architecture-level, we learn signatures that predict lulls in instruction throughput. These predictions support a Dynamic Frequency and Voltage Scaling (DVFS) policy that saves power throughout compute-dominated workloads.

Chapter 5: I/O-Response Modeling for Multi Tenant Storage - We learn patterns in I/O-request statistics when different users' objects are co-located on the same device. A CART model identifies conflicting workloads that produce disproportionately bad performance. This human-readable model is useful for both system-administrators and automated object-placement mechanisms. We use unsupervised learning to discover latent structure corresponding to distinct workloads. This structure reduces CART sensitivity to low-level background processes that interfere with measurement, improving prediction accuracy in datacenter scenarios.

Chapter 2

Background

On-the-fly performance tuning starts with manufacturing. Flexible devices require alternate data pathways, and these must be both technologically *and* economically feasible. To implement reconfiguration policies that control these pathways based on system state information, low-level software control is also necessary. Section 2.1 reviews these capabilities in common devices.

State prediction requires a modeling framework that carefully manages the platform constraints introduced in Section 1.3.1. Section 2.2 surveys machine learning tools that can balance statistical accuracy, training time, and computation complexity, while coping with "real-world" measurement artifacts. These are the basis of the wireless, circuit, and storage state-models deployed in Chapters 3, 4, and 5, respectively.

2.1 Hardware Support for On-the-Fly Performance Tuning

2.1.1 Subsystems Enabling Runtime Adjustment

Reconfigurable systems owe their flexibility to low-cost manufacturing. Technologies such as tri gate transistors and strained silicon reduced chip feature sizes from 130nm to 14nm in only 12 years [9]. This dropped manufacturing costs by an estimated 30% per 10nm [10]. Meanwhile, storage costs fell by 5x for hard disk drives and NAND memory in the four-year period ending in 2012 [11].

To realize savings from new technologies, manufacturers pair technical breakthroughs with better economic strategies. This is necessary because manufacturing costs rise as devices get smaller and require greater precision. For example, a 14nm-process fabrication plant planned for 2013 cost Intel more than $5 billion [12]. Photo lithography masks, a key component in the manufacturing process, cost more than $1 million each at 22nm [13]. Yield-driven production strategies recover these expenses, for example by selectively disconnecting components of a chip post-fabrication to sell one die at different price-points [14].

Architects have exploited low-cost hardware to increase runtime flexibility at nearly every level of device design. At the circuit level, Dynamic Frequency and Voltage Scaling (DVFS) reduces the power consumed along circuit pathways at runtime. Provided slower performance does not disrupt computations, this mechanism can adapt a device's power profile to workload. Commercially-available DVFS mechanisms from both AMD and Intel make adjustments at the $10-100\mu$s timescale [15].

Research designs have further reduced adjustment time to nanoseconds [16].

At the architecture level, many processors are equipped with computationally powerful microcontrollers to manage chip operations. For example, the IBM POWER8 CPU repurposes a PowerPC 405 to control gating, frequency, and voltage settings per core [17] [18]. In this case, the processing capabilities of an entire early-2000s CPU manage parallelism against a power-consumption budget. Other devices like the massively parallel nVidia Maxwell GPU similarly trade off chip size for control. In this design, additional scheduling units increase size by 20% in order to selectively power off idle execution units at runtime [19].

Computational capabilities have also migrated into peripheral systems like storage. Self-healing solid-state drives (SSDs) use microcontrollers to detect flash memory failures and manage repairs. Extra computations relocate data to backup chips upon failure, redirect I/O requests temporarily, and prioritize repair operations alongside normal user activity. These optimizations target wear-out failures that inhibit widespread adoption of cheap, high-capacity SSDs [20].

Datacenters have a long history of leveraging low-cost computation and storage hardware. Redundant devices spread-out across networks and geographies provide flexible options for data placement. Computational resources then implement policies that manage network performance and reliability across devices [21] [22]. Datacenters have also given rise to economically-incentivized performance optimizations. The best example is Amazon, which leases datacenter resources to customers using market-driven pricing [23].

2.1.2 Virtualization as a Software Convergence Layer

To realize performance gains in reconfigurable systems, flexible low-level control is necessary. Coinciding with cost reductions in hardware manufacturing, virtualization technology has pushed software deep into many devices. Virtualization gets its name by implementing a lightweight, software-driven, universal interface to heterogeneous hardware resources [24] [25]. By improving interoperability, systems like Xen [26] and VMWare [27] originally simplified long-term management and reduced capital costs in datacenters. Since the initial wave of adoption in the late 1990's, virtualization has spread to consumer devices with Google's Dalvik/Android system [28], network hardware with OpenFlow [29], storage through software-driven object management systems [30], and soon to circuits with the Intel SoFIA.

Virtualization not only simplifies high-level design, it uses low-level software to manage device resources at a fine grain, often with direct hardware support. As a result, it implements a convergence layer ideal for state modeling. For example, software-defined radio systems like GNU Radio [31] or the Joint Tactical Radio System (JTRS) [32] can implement link prediction at the lowest layers of the network stack. Similarly, virtualized circuit platforms like the Intel SoFIA are well-suited to implement the workload-prediction optimizations of Chapter 4. And object-based storage systems that manage datacenters of heterogeneous devices are the target of the performance modeling techniques of Chapter 5.

2.2 Machine Learning Tools for Modeling System State

Section 1.3.1 introduced common constraints that affect state-models in wireless, circuit, and storage systems. These were *MLM*(1) *localized statistical stability,* *MLM*(2) *constrained acquisition of labeled training data,* and *MLM*(3) *utility of domain knowledge to mitigate complexity.* This section introduces machine learning techniques that satisfy these requirements, while addressing variation in measurement data and the restrictions of our target platforms.

2.2.1 Coping with Real-World Measurement Errors Using Sparse Feature Representations

Sparse feature representation is a powerful method for coping with non-Gaussian data variations. These occur in uncontrolled acquisition conditions such as environments containing interference, or situations that preclude careful alignment between measurements and their target. Errors manifest as signal distortions like shifts, deletions, and stretches. These are common in "real-world" data, and sparse feature representation is an important component of systems for natural language processing [33], object-recognition [34] [35], and signal processing [36] [37].

To compute a sparse feature representation, raw measurement data is expressed in terms of a small number of canonical patterns, called *features.* This process latches measurements representing noisy or incomplete signals to stable exemplars, restoring garbled information. Sparsity is enforced by imposing a penalty on the total number of features chosen to represent a measurement vector. This constraint cuts away weakly-expressed information like low-level interference. Unlike approximation methods that purely minimize Euclidean distance, sparse approximation purposefully removes information to remove noise, suppress clutter, and amplify important trends. As a result, statistical models fit to sparse feature representations more-accurately capture trends that can be distorted during data acquisition. Interestingly, evidence suggests that this process occurs in biological systems like the visual cortex [38].

2.2.1.1 Learning Features by Clustering

An encoding algorithm that transforms raw data into a sparse feature representation requires a set of appropriate canonical features. Experts can define such features by hand, as in object-recognition systems like SIFT or biologically-inspired vision systems [39] [40]. However, for data with unknown or unmanageably complex structure, expert-driven feature-design is impractical.

In this case, unsupervised learning techniques can find features from a large representative dataset. Collections of candidate features are often called *dictionaries*. For example, dictionaries in this work are overcomplete matrices whose columns represent canonical patterns found in measurement data. The process of computing a dictionary using unsupervised learning is called *dictionary training*.

The most common generic approach to unsupervised dictionary training is clustering, which groups measurement samples to identify recurring patterns. Since effects like clustered values often result from an underlying unobservable process, cluster definitions are one form of latent or hidden statistical structure. Given their straightforward explanation, clustering algorithms are common in both automated learning systems and data-mining systems that include humans-in-the-loop [8].

2.2.1.1.1 Capturing Latent Structure with K-Means Depicted in Figure 2.1, K-Means is one simple clustering formulation. It assumes that measurement data can be grouped into K clusters, each represented by their mean value, or *centroid* [41]. K-Means is solved greedily by iterating between a cluster-assignment step, and a centroid-update step. The algorithm starts with K as an input parameter, and initializes K candidate centroids. In the assignment step, all measurement vectors are associated to the closest centroid by minimum Euclidean distance. Then, in the update step, each centroid is recomputed by averaging its constituent measurement vectors. This process repeats until a convergence criterion is satisfied.

Though straightforward, K-Means is capable of finding features that support state-of-the-art object-recognition systems [42]. Under this formulation, cluster centroids become dictionary features, and the sparsity constraint $k = 1$ is implied (notationally, we note the difference between K: the number of clusters, and k: the constraint on the number of features used in a sparse representation). K-Means is convenient in many situations because it aligns with a generative Gaussian mixture model.

2.2.1.1.2 Sparse Coding: Linear Combinations of Concurrent Features Sparse coding generalizes K-Means to express measurements with a linear combination of cluster centroids, as shown in the right portion of Figure 2.1. This more-expressive framework is useful when measurements result from multiple underlying processes operating at the same time. For example, circuit performance data will capture concurrently-operating chip components as they increment a common set

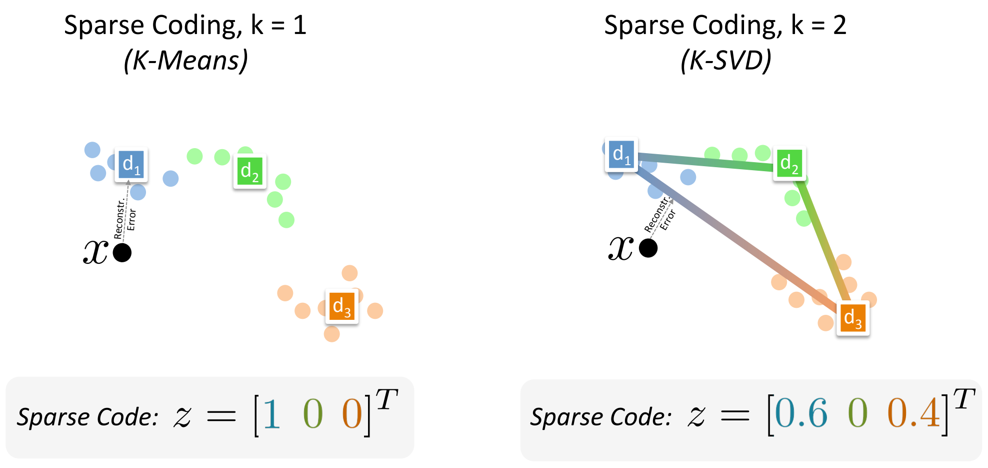


Figure 2.1: Sparse coding is a clustering method that generalizes K-Means. Unlike K-Means, sparse coding allows data to be explained by a linear combination of cluster centroids. This model more-closely fits systems like circuits, in which multiple components increment a common set of counters.

of counters. Clusters found using K-Means will map awkwardly to combinations of circuit components, while clusters found using sparse coding can separate them.

Sparse coding is the subject of intense study, and many alternative formulations exist to exploit tradeoffs like statistical accuracy or computation speed [43]. One simple formulation is based on minimizing approximation error under a strict ℓ-0 constraint on the number of features that may be used:

$$\min \|X - DZ\|_2 \quad \text{s.t. } \|z_i\|_0 \leq k \text{ for } i = 1...t \tag{2.1}$$

with X a matrix containing t n-dimensional measurement vectors, D a feature dictionary with m cluster centroids as its columns, and Z a matrix of t sparse feature vectors with at most k non-zero values. In the prior circuit example, X would contain values captured from n counters sampled at t time instants. Dictionary atoms would loosely represent different underlying circuit operations with unique signatures evident in counter values. Note that when $k = 1$, this formulation is the same as K-Means with the notational difference that K clusters become the m columns of D.

Though ℓ-0 minimization is NP-Hard, greedy approximation algorithms are known. Just like K-Means, K-SVD solves this clustering problem by alternating between cluster-assignment and update steps [44]. In this formulation, sparse feature vectors z_i reflect a weighted combination of several clusters, so K-SVD uses the Matching Pursuit (MP) family of algorithms to perform the assignment step. These are presented in detail in Section 2.2.3.1. Furthermore, since clusters are not a pure average of their constituent vectors, K-SVD uses Singular Value Decomposition (SVD) to update centroids by rank-1 approximation. Though fast, this greedy approach is sensitive to initial conditions.

In contrast, ℓ-1 regularization provides an alternative formulation with a convex objective function (when $m < n$) that can be solved directly. Called the general Least Absolute Shrinkage and Selection Operator (LASSO), this formulation is given by:

$$\min \|X - DZ\|_2 \quad + \ \lambda\|z_i\|_1 \text{ for } i = 1...t \tag{2.2}$$

where λ is a "regularization" parameter that penalizes the sum of absolute values in any sparse feature vector [45].

Though LASSO produces more-stable dictionaries than K-SVD in general, solving it is more-complicated. In practice, we use LASSO to compute dictionaries offline when computation constraints are minimal. Then, once D has been trained, we encode new measurements using easily-parallelizable MP algorithms according to the formulation in (2.1).

2.2.2 Modeling Local Statistics

Clustering and sparse feature representations capture variation-tolerant signatures of system processes – supervised statistical models can then learn the relationships between these signatures to implement state prediction. There are many supervised learning tools, including hypothesis testing, regression, Bayesian inference, Classification and Regression Trees (CART), and Support Vector Machines (SVMs). For all of them, training data must include both a measurement vector and the outcome of the process being modeled, called a *label*. Labeled data is often orders-of-magnitude more costly to acquire than unlabeled data. For example, in image-based object-recognition, labels are the result of human cognition, and acquisition is bottlenecked by the time it takes a person to annotate individual pictures.

When modeling system state, measurement vectors contain samples of performance data and labels reflect a future observed system state. Prediction is implemented by assigning an expected future-state-label to a new input measurement vector. In the simplest case, labels indicate the occurrence of a target event such as a link outage; a binary classifier is then sufficient for prediction. Labeled data is limited in state modeling because every acquisition is a missed optimization opportunity. Furthermore, the relationships between system states change over time as environment, application, and workload changes. Labels are therefore useful only within a limited context, making acquisition difficult to amortize over time.

2.2.2.1 Linear Support Vector Machines

We use SVMs to predict binary events because they are widely considered to be the most flexible and powerful classifier available. In its most basic formulation, an SVM computes a linear separator that maximizes the sum of weighted distances between a separating plane and measurement vectors in different classes. The best separator is a vector whose coefficients have largest magnitude in dimensions that discriminate between classes. Weight vectors therefore provide hints into which measurement dimensions are most useful when telling classes apart.

However, separators can be defined as a kernel function of arbitrary shape. This flexibility enables the SVM formulation to implement probabilistic separators and to generalize techniques like logistic regression. Intuitively, the effect of a non-linear kernel is to first apply a function $f(\dot{})$ to data points x, and then to find a linear separator between transformed points $f(x)$ [46]. When sparse feature representations are computed from raw measurement data and fed to a linear SVM, sparse coding becomes a form of data-dependent kernel [47].

2.2.2.2 Piecewise State Prediction Over Time

Linear approximation is a data-efficient way to capture complicated functions when little information is known about their shape. This is because any n data points can be used to approximate an n dimensional signal by regression. When applied piecewise, independent linear models fitting local portions of a function's domain together capture arbitrary continuous functions. This is especially useful if data acquisition naturally samples points within a small region.

Linear SVMs that perform state prediction over short windows of time implement a similar procedure. In all of our applications, a few prominent predictive effects dominate at any time. For wireless, these are sources of packet loss specific to the current environment; for circuits, they are the function blocks of a workload phase; and in storage, they are the access patterns of a transient workload. The predictive relationships between these dominant processes and system state represent a limited local context of a much more-complicated global prediction model.

Rather than building general high-order statistical models for all possible runtime conditions, it is most effective to train individual models for limited local contexts. This is because data acquisition is highly time dependent – as we will show in later chapters, the statistics of system states driven by environment, program code, and user demands change over time. It is not only prohibitive to collect statistics on the global state function, it is not necessarily useful. For example, predictive trends may change enduringly when a wireless user leaves an environment for good, software code is upgraded, or a storage client changes their workflow.

We therefore implement prediction using linear SVMs to model local statistics. The solution found by a linear SVM can be interpreted as one portion of a piece-wise linear regression of the global state function. For environment-, application-, and workload-specific models, linear predictors require few training samples to converge, enabling our system to easily adapt to new runtime conditions.

2.2.2.3 Modeling Time Series Hierarchically

When predictive signatures of system state stretch over large spans of time, state modeling suffers the curse of dimensionality: each linear increase in scale produces an exponential increase in the number of possible states that may occur. Though clustering and sparse representation collapse variations corresponding to the same latent state at small scales, training data requirements make them impractical when signal dimensions stretch into the thousands or millions. In this case, hierarchical modeling is a tractable approximate solution that is central to deep learning systems for vision, speech, and language processing [48] [49].

As shown in Figure 2.2, a hierarchical model encodes statistical relationships within small independent signal patches at its lowest layer. Relationships *between* these patches are then captured at subsequent higher layers. Practically, this is implemented by cutting a long sequence of measurement data into small pieces, and sparse-coding each independently. Feature vectors corresponding to neighboring patches are then concatenated to increase scale before coding them again with a dictionary that captures feature co-occurrences.

Hierarchy increases training efficiency by assuming that patches within a layer are independent and identically distributed. Full-sized training samples can be subdivided and used to learn a single dictionary for encoding independent patches. This increases the number of patch-sized training samples multiplicatively. Furthermore, reducing signal size to that of a much-smaller patch decreases the number of possible sequences that may occur. As a result, training data requirements drop exponentially. Together, these two effects make it tractable to find latent structure within very large

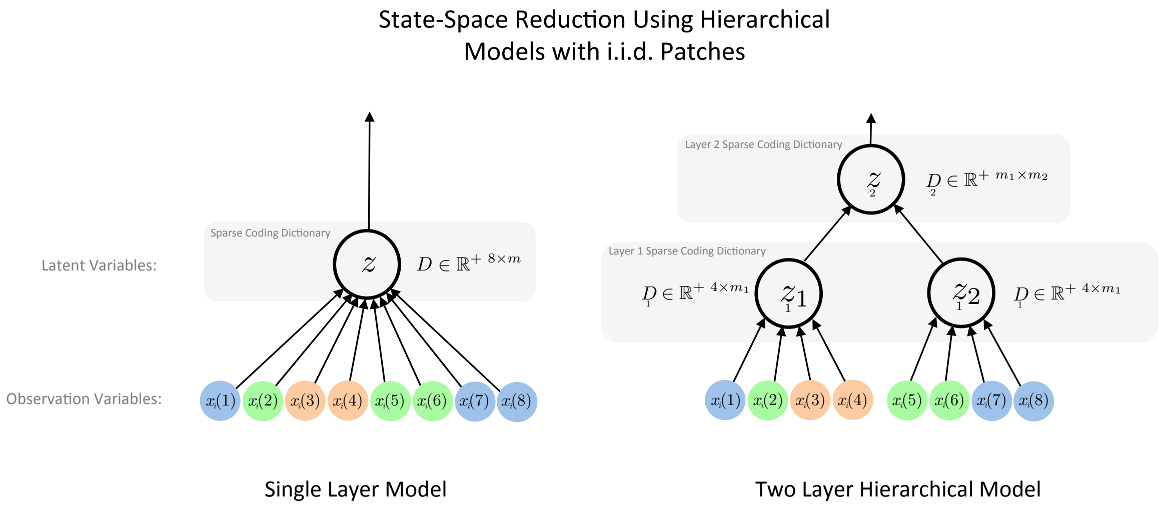

State-Space Reduction Using Hierarchical Models with i.i.d. Patches
Latent Variables:
Observation Variables:
Sparse Coding Dictionary
z
$D \in \mathbb{R}^{+\ 8 \times m}$
$x_i(1)$ $x_i(2)$ $x_i(3)$ $x_i(4)$ $x_i(5)$ $x_i(6)$ $x_i(7)$ $x_i(8)$
Layer 2 Sparse Coding Dictionary
z_2
$D_2 \in \mathbb{R}^{+\ m_1 \times m_2}$
Layer 1 Sparse Coding Dictionary
$D_1 \in \mathbb{R}^{+\ 4 \times m_1}$
$z_1 1$
$z_1 2$
Single Layer Model
Two Layer Hierarchical Model

Figure 2.2: Hierarchy is a general way to ease training data requirements when modeling long sequences. The number of possible patterns that may occur in patches decreases exponentially with patch size, while the number of training samples increases multiplicatively as data is subdivided. Conveniently, hierarchy is an appropriate compositional model for many data modalities, especially in vision, language, and speech applications.

measurement vectors.

In addition to its statistical benefits, hierarchy reflects the natural properties of many datasets, such as vision, speech, and language data [50] [51] [52]. For example, images are often composed of smaller component parts, such as faces that have eyes, noses, and mouths. Language structure fits words into phrases, phrases into sentences, sentences into paragraphs, etc. And, speech data follows a similar structure starting with phonemes, then words, and then phrases. Interestingly, primates' visual cortices model data hierarchically by processing visual inputs in different regions of the organ at increasing degrees of complexity [48].

2.2.3 Computational Complexity and Parallelization

A rich variety of approaches exist for learning models based on sparse feature representations. These include convolutional neural networks, latent-dirichlet allocation, sparse coding, etc. As a result, we can choose a formulation to carefully manage trade-offs of training-data requirements, computational complexity, and parallelization. For example, deep neural networks are popular in vision, speech, and language processing systems. They use the supervised back-propagation algorithm to refine data representations until a model's statistical objective is maximized. While deep learning achieves excellent accuracy on high-dimensional data, commercial implementations utilize tens of thousands of compute cores [34].

In contrast, the modeling framework employed in later chapters uses sparse coding to compute data representations, and linear SVMs to perform classification. While dictionary training occurs offline, SVM training happens either on- or off-platform depending on computation constraints. In all applications, encoding and SVM classification happen on-platform. This configuration gives up the statistical accuracy of large neural network models for the flexibility to spread learning and prediction modules across platforms.

2.2.3.1 Trading Encoding Accuracy for Complexity

For all of our applications, performance gains are sensitive to the cost of computing sparse feature representations. Fortunately, sparse coding's multiple formulations allow room for optimization within our prediction framework. For example, low-overhead wireless link predictions can be calculated in microseconds without dedicated hardware using the Matching Pursuit (MP) algorithm shown in Algorithm 1 [53]. MP iteratively computes sparse-representation coefficients by identifying the largest inner-product correlation between a residual vector and dictionary features. This algorithm consists primarily of matrix-vector multiplications and can be easily parallelized.

Since its introduction, researchers have extended MP to improve approximation accuracy and stability using additional computations. For example, it's closest off-spring is Orthogonal Matching Pursuit (OMP), shown in Algorithm 2 [54]. OMP re-computes its residual vector each iteration by performing a least-squares fit between an estimated set of supporting dictionary features I and the original measurement vector. This step reduces error propagation throughout the algorithm's iterations. When the signal dimension n is small, least-squares computations are expensive relative to overall computation time. However, if n is large or the target platform has a data-parallel hardware accelerator such as a GPGPU, OMP can efficiently improve encoding accuracy. This is the case for the datacenter storage systems of Chapter 5.

2.2.3.2 Accelerating Matching Pursuit on GPGPUs

Using a data-parallel hardware accelerator, OMP can be distributed among processors to reduce encoding time. For example, we have tightly optimized the conjugate gradient method for least-squares on nVidia Fermi GPUs [55]. For this architecture, OMP is bound by the cost of communication between independent parallel multiprocessors during vector-norm computations, as depicted in Figure 2.3. Newer iterations of this hardware like the nVidia Maxwell partially alleviate I/O limitations

Algorithm 1 Matching Pursuit (MP)

Input: $n \times 1$ data vector x, $n \times m$ matrix D, sparsity constraint k

Output: sparse vector z

```
1: $r_0 = x; i = 0; z = \vec{0}^m;$
2: **while** $i < k$ **do**
3:     $q = D' r_i$
4:     $j = \arg\max(q)$
5:     $z_j = q_j$
6:     $r_{i+1} = r_i - q_j D_j$
7:     $i = i + 1$
8: **end while**
```

for small problem sizes by increasing the amount shared memory available to compute units. Researchers have also proposed algorithmic modifications to address this bottleneck [56].

2.2.3.3 Exploiting Dynamic Sparse Dependencies

When OMP is parallelized across many-core platforms, communication bottlenecks may be caused by nodes holding irrelevant operands. For example, after OMP updates its estimated support set I, only a few nodes will participate in the inner-loop least-squares calculation. Other nodes should therefore be prevented from slowing the global computation. Since such optimization opportunities are determined on-the-fly, compile-time optimizers cannot exploit them. Algorithms with this runtime structure are called *dynamic sparse computations* [57] and include MP, OMP, and K-SVD.

We added support for dynamic sparse computations to GraphLab, a many-core parallel programming framework. By introducing two new primitives into the GraphLab

Algorithm 2 Orthogonal Matching Pursuit (OMP)

Input: $n \times 1$ data vector x, $n \times m$ matrix D, sparsity constraint k

Output: sparse vector z

```
1: $r = x; I = \{\emptyset\}; z = \vec{0}^m;$
2: while $|I| < k$ do
3:     $q = D'r$
4:     $I = I \cup \arg\max(q)$
5:     $z_I = D_I^+ x$
6:     $r = x - Dz$
7: end while
```

programming abstraction, OMP's runtime decreased by 4x. The "Selective Push-Pull" primitive temporarily focuses computation on a small number of self-selected nodes. "Statistical Barriers" then prevent a few straggling workers from holding-up computation. This barrier waits for a percentage of workers to check-in at synchronization points before leaving the rest behind – provided no operands are systematically ignored, OMP will naturally correct for missing interim values over subsequent iterations. This implementation was used for the offline learning computations of Chapter 4.

2.2.4 Interpretability for Humans-in-the-Loop

In many scenarios, learning algorithms operate as part of a larger design process that includes humans-in-the-loop. This is the case for circuit workload-prediction, when encoding and prediction systems require implementation in hardware. Here, capital requirements mandate a conservative, well-tested design-process that consists

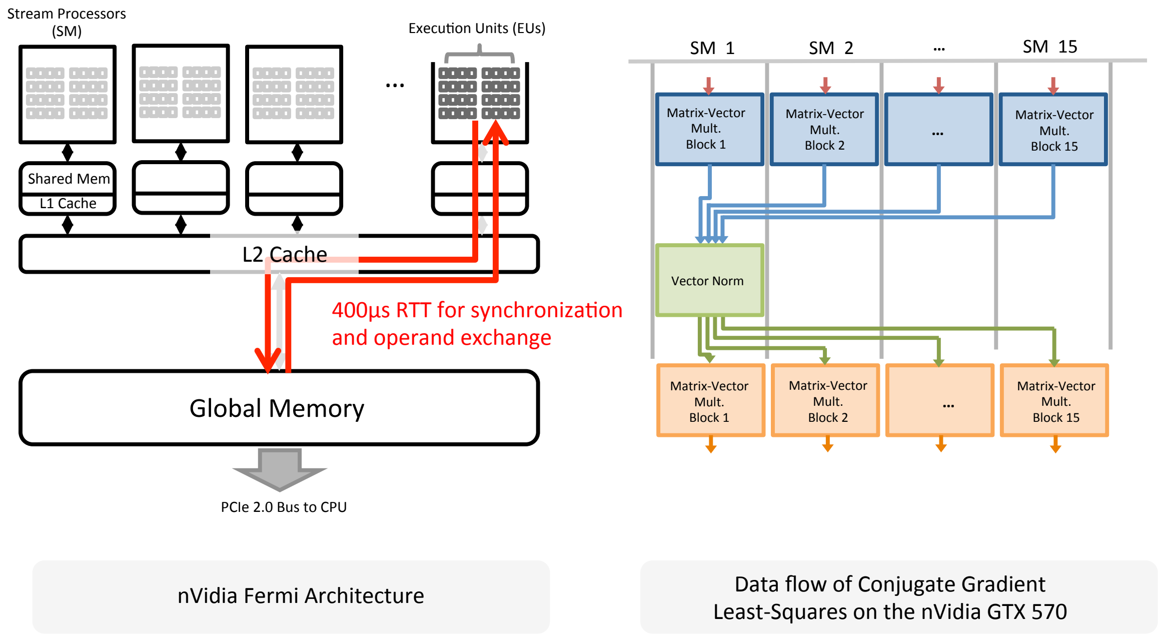

Stream Processors (SM)
Execution Units (EUs)
...
Shared Mem
L1 Cache
L2 Cache
400µs RTT for synchronization and operand exchange
Global Memory
PCIe 2.0 Bus to CPU
nVidia Fermi Architecture
SM 1
SM 2
...
SM 15
Matrix-Vector Mult. Block 1
Matrix-Vector Mult. Block 2
Matrix-Vector Mult. Block 15
Vector Norm
Data flow of Conjugate Gradient Least-Squares on the nVidia GTX 570

Figure 2.3: The conjugate gradient least-squares-estimation algorithm forms the inner loop of OMP. Consisting primarily of data-parallel inner product computations, it can be accelerated using commodity GPUs. However, when shared memory resources are limited, synchronization can bottleneck OMP. Slowdowns are alleviated by better support for on-board synchronization and operand exchange, for example with neighbor-communication or more low-latency memory resources.

of many iterations. This is also the case in storage scenarios, when systems are sold to third-party clients. For these, client-side system administrators must answer for datacenter performance, including learned placement policies.

Some advanced machine learning techniques do not interface well with humans-in-the-loop due to their level-of-complication. For example, SVMs produce accurate statistical classifications that are can be hard to interpret. Separator weights identify features that influence classification, but they do not easily convey the space of all important signatures associated with a target class. Interpretation is even more difficult when many SVMs are combined to implement multi-class prediction.

2.2.4.1 CART and Interpretable CART Models

Classification and Regression Trees (CART) are one alternative supervised learning tool that better-supports humans-in-the-loop. The CART algorithm progressively applies binary splits to labeled training data to greedily improve a target metric like classification error. CART models not only fit non-linear trends, but produce human-readable output when trees are kept to manageable complexity. Unfortunately, they are also prone to overfitting and make poor use of latent data structure. In Chapter 5, we show that CART models trained on sparse feature representations are more-resilient to overfitting. Since clusters are well-understood and easily visualized [58], this procedure does not hurt interpretability.

Recently, researchers proposed an explicit formulation for interpretable CART models that constrains their depth and breadth [59]. Not only are results guaranteed to be low complexity, but the optimization objective is convex. This means that solutions are not prone to overfitting, as with greedy approximations algorithms. The result is a stable, uncomplicated prediction model.

2.2.4.2 Feature-Encoding by Match Filters

Finally, we point out that learning algorithms based on Matching Pursuit have a straightforward explanation in terms of match filters. Each iteration, MP computes inner-product distance between a signal vector and dictionary features. When features are referred to as “filters”, this procedure aligns with match-filtering, a common signal processing technique in communications and electrical engineering. Through collaboration, we found that techniques like SVMs with custom kernels that draw heavily on machine learning terminology are difficult to communicate across disciplines. However, an explanation of sparse encoding based on match-filtering terminology drew on common knowledge. The result was clear communication and a shared intuitive understanding of strengths and weaknesses.

Chapter 3

A Protocol- and Environment-Aware Wireless Link Layer

We now implement a wireless link model that improves throughput in everyday scenarios by predicting temporary link outages. In harsh transmission environments like those found indoors, in urban settings, and in vehicular networks, link variability disrupts network protocols. In the example scenario described in Section 1.1, TCP connections will improperly throttle back upon delivery failures and time out if all packets in a transmission window are lost. For transient outages, this causes low utilization once links are restored. The result is dismal throughput and a frustrating user-experience when devices are slow to react to environment changes.

Our system, State-Informed Link-Layer Queuing (SILQ), actively measures links to predict such outages. SILQ then holds packets at the link layer to preempt TCP when appropriate. This method trades a slight increase in latency for more-predictable transport behavior. As a result, timeouts are avoided and connections react quickly to link fluctuations. We find that overall throughput in scenarios like subway networks increases by up-to 4x, while variation over repeated experiments drops similarly. This chapter presents SILQ in detail, and includes extensive data-collection and field-validation in rural Unmanned Aerial Vehicle (UAV), urban subway, and indoor office environments over 802.11b and 3G cellular networks.

3.1 Wireless Link Prediction in Everyday Environments

Daily, wireless devices face the challenge of providing fast reliable data delivery over flaky links. Link loss arises due to a confluence of factors, including signal strength attenuation, antenna pattern nulls, line-of-sight occlusions, and multipath interference. In general, these effects are determined by complicated environmental properties like geometry, and are exacerbated by node mobility. Consequently, transmission quality is difficult to anticipate in practice.

Strategies for addressing data loss at the lowest layers of the network stack shield applications from unreliable link conditions. At the physical layer, forward-error-correction (FEC) improves symbol resilience to individual bit errors. At the link layer, Automatic Repeat Request (ARQ) quickly detects frame errors to trigger retransmissions [60] [61]. And at the transport layer, TCP options like Fast-Retransmit and F-RTO dampen throughput degradation caused by intermittent packet losses [62].

Inherent to all of these loss-mitigation protocols is an assumption that the link is *actually available* at the 10-100ms timescale. This assumption breaks routinely, for example when users encounter coverage dead-zones. The consequences are unintended protocol behaviors like improper throttling and timeout, which in turn lead to erratic network performance and unreliable applications.

Outage-prediction is one method for stabilizing network protocols. This strategy uses a link model to anticipate future delivery failures, and holds transmissions temporarily until link conditions improve. Link models in the literature tend to be complex and require large amounts of input data. The two most common approaches are ray tracing and statistical modeling. Ray tracing uses detailed physical simulation to compute signal strength and requires comprehensive geometric and propagation information about the environment [63], [64]. In contrast, statistical modeling abstracts away physical effects by capturing relationships between link measurements, like clusters formed by recurring delivery patterns. Examples include applications of Markov chains that reproduce the first- and second-order statistics of 802.11 and GSM networks [65], [66], [67], [68]. For these models, accuracy depends on the number of state-transition probabilities that must be estimated and the amount of available training data.

Hybrid models blend statistical and physical information, such as a mobile node's location [67] [69]. However, in practice we find that location is a poor predictor of individual packet losses for many scenarios. For example, in indoor environments, link state in a particular location may change significantly if a coworker holds open a nearby fire-proof door, temporarily removing a source of signal blockage. In scenarios like subway tunnels, precise location information is simply not available once the train passes underground.

The statistics of wireless link measurements have inherent predictive power because physical factors that affect signal propagation create correlations in transmis-

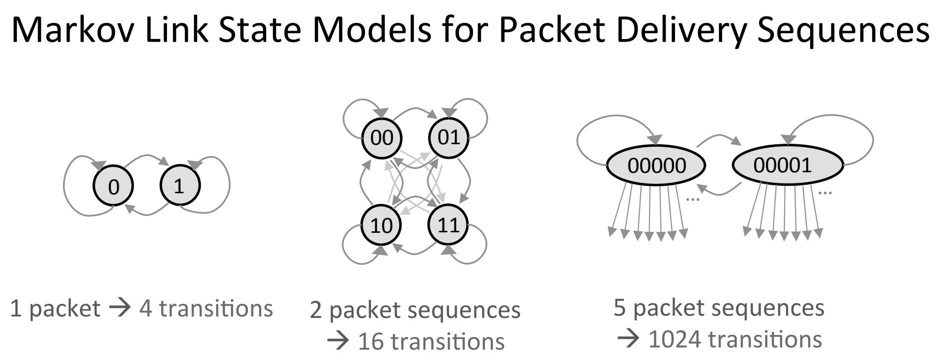

Markov Link State Models for Packet Delivery Sequences
0
1
00
01
10
11
00000
00001
...
1 packet → 4 transitions
2 packet sequences → 16 transitions
5 packet sequences → 1024 transitions

Figure 3.1: Markov models are popular in the research literature for capturing sequences of bit- or frame-transmission errors. The number of state-transitions in these models grows exponentially with the sequence length being modeled. In practice, this creates onerous training-data requirements that limit the ability of general Markov chains to capture predictive effects over time.

sions over time. This effect is often abstractly referred to as "channel memory." For example, when occlusion from the UAV engine in our airborne scenario blocks back-to-back packet transmissions, the failure of several deliveries in-a-row may suggest the outcome of the next.

We therefore implement a data-driven statistical link model, and actively probe links to acquire measurement data. We conduct dedicated data-collection campaigns in diverse environments and use unsupervised learning to find stable link-state signatures. Outage predictors are then implemented by sparse-coding probe data and training a linear SVM for individual environments. During online operations, our system probes outgoing links at a low rate and uses returning reception-reports to make outage predictions. From these, transmitters either forward or hold data packets to avoid delivery failures. This approach allocates a small amount of link bandwidth to state-modeling, but requires no side-channel information like node position.

3.2 Scenarios and Field Experiments

We now describe our rural UAV, indoor office, and urban subway scenarios. In each environment, we measure links to find and compare link states using unsupervised learning. Links are probed by transmitting 66B UDP packets every 1ms. Each packet carries a sequence number that is logged by receivers. Receivers detect failed transmissions from non-consecutive probe arrivals and reconstruct a binary sequence of probe-delivery outcomes, referred to as a link *trace*.

Due to the speed and range of our UAV, we transmit data at 1Mbps modulation for best resilience; we then match this speed in all environments so that results are comparable. In Section 3.4.2 we show that results scale to 10Mbps. For UAV and indoor scenarios, we disable ARQ retransmissions by sending packets in broadcast mode. For the subway scenario, wireless hops implement the 3GPP cellular standard's Hyrbid-ARQ mechanism with 10 retransmissions [70].

3.2.1 Rural UAV Scenario

The UAV flight path is illustrated in Figure 3.2. A low-altitude fixed-wing aircraft flies multiple laps in a dumbbell-shaped pattern over an airfield in upstate New York. Auto-pilot maintains positional consistency between laps, and GPS location variation is on the order of 15m when banking through turns. The flight path spans approximately 1,000m east-to-west. Two wing-mounted 802.11b/g transmitters broadcast UDP probes to ground nodes arranged throughout the property. Ground node locations are organized in three groups, each representing a distinct signal-propagation environment: field, forest, and ground-structure. Measurements are collected primar-

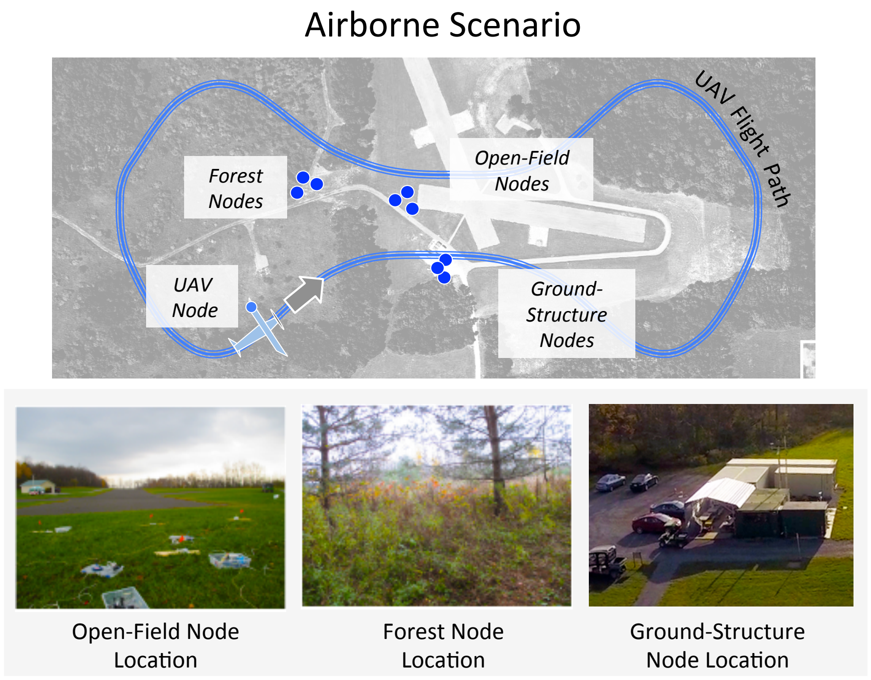


Figure 3.2: The experimental setup for our rural UAV experiment is shown. The UAV test facility covers forest and farmland in upstate New York. Packet losses are driven primarily by radio range effects – deliveries at range edges are intermittent since transmission becomes sensitivity to effects like antenna-pattern nulls and line-of-sight occlusion from the UAV engine block.

ily in October when leaves have fallen and temperatures are low.

Data delivery over these UAV wireless links represents a scenario of both tactical and commercial interest. Alleviating bottlenecks in military networks is a research priority [71], and software defined radios like the JTRS platform can easily be extended to implement link prediction [32]. In the private sector, UAV platforms for Internet relay, package delivery, and bridge, crop, and wildlife inspection are currently in development [72] [73].

3.2.2 Indoor Office Scenario

Our indoor office scenario is shown in Figure 3.3. A user walks throughout an active office environment during normal daytime hours. A handheld 802.11 device communicates with a fixed-position access point (AP) as the user traverses a footbridge to a nearby building, rides an elevator to the basement, and then returns to the AP location. Throughout this tour, packet-loss conditions fluctuate due to radio range and signal attenuation from building structures. For example, links are completely severed when heavy, metal fire doors shut between user and AP. Similarly, the metal elevator car causes significant packet loss, even though signals are powerful enough to reach the basement. Throughout experiments, transmissions are subjected to cross-channel interference from the building's working 802.11 network. This scenario represents a typical indoor environment encountered by most wireless users on a daily basis.

3.2.3 Urban Subway Scenario

Finally, our subway scenario is shown in Figure 3.4. Probe packets are transmitted between a server on the Harvard campus network and a handheld 3G cellular device aboard a subway train. These packets pass through multiple wired Internet hops in addition to the last-hop 3G wireless link. We observe that round-trip time averages 150ms and fluctuates within a manageable distribution. Queuing delays cause probes to arrive in a bursty pattern, however bursts are of roughly consistent length and spacing.

We conduct experiments as a user rides from Cambridge to Boston, MA during daytime hours. Dead zones occur at unknown points in tunnels, creating link outages

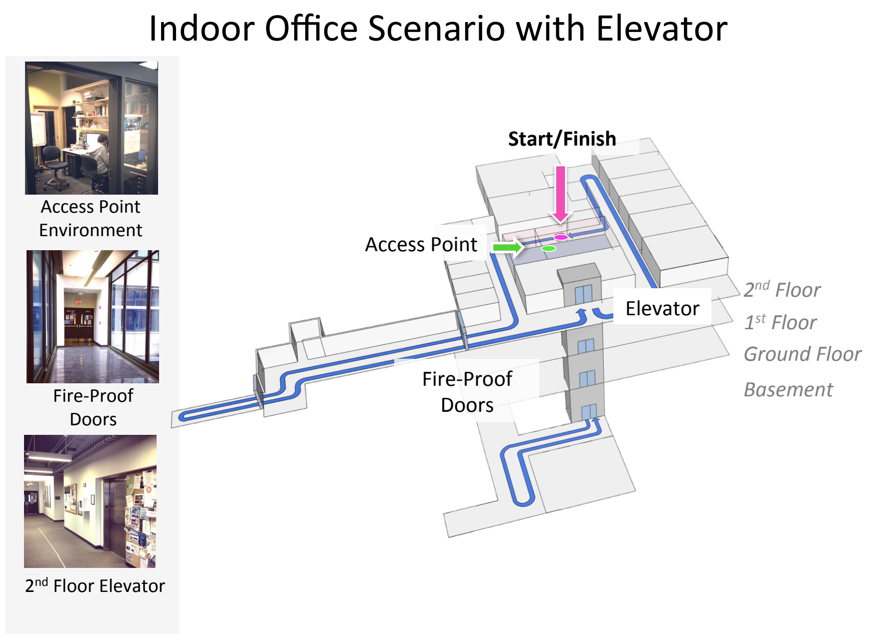


Figure 3.3: Our walking tour throughout an indoor office environment is shown. Fire-proof doors and an elevator are significant drivers of packet loss. Experiments are conducted during normal working hours and data includes cross-channel interference from the building's 802.11 network.

lasting in the 10's of seconds, depending on train speed. Clean line-of-sight transmission is possible between Kendall Square and the Charles/MGH stops when the train passes over a bridge. At this point, we often observe that baseline TCP connections have timed-out due to prior outages, causing links on the bridge to go unused. This environment represents a scenario of particular aggravation to the author.

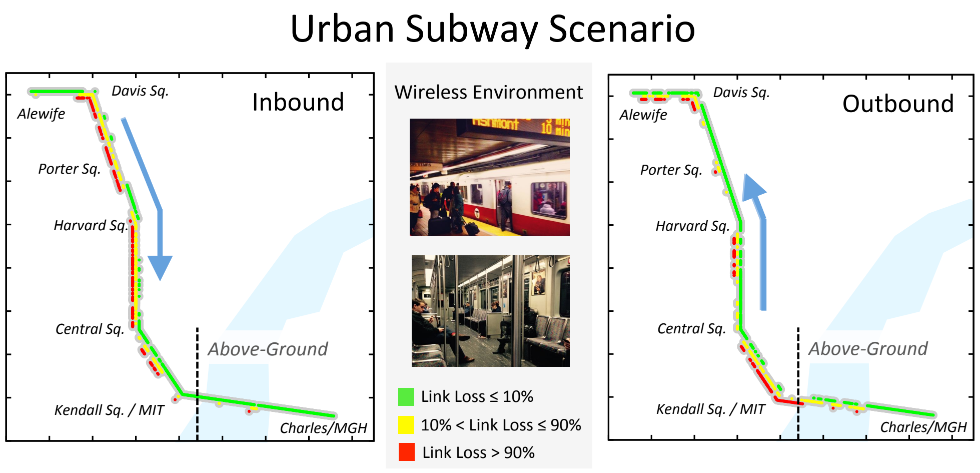


Figure 3.4: Probe reception rates over a commercial 3G cellular network are illustrated along a portion of subway track in Boston, MA. Location information is unknown, and link loss due to dead zones fluctuates with train speed.

3.3 A Sparse-Coding Outage-Prediction Model

3.3.1 TCP Response to Link Outages

Our ultimate goal is to use a link model to implement predictions that improve network performance. Given TCP's ubiquity in modern networks, we use bulk data delivery over a TCP connection as our application vehicle. Other candidate applications include media streaming, reliable transaction processing, or any scenario in which link loss adversely affects higher-layer performance. TCP's behavior over lossy wireless links is illustrated in Figure 3.5.

Though the Linux network stack can compensate for intermittent delivery failures using Fast Retransmit, F-RTO, ARQ, and FEC, it is configured using static parameter settings such as the number of ARQ retransmissions. This design implicitly assumes that link loss is stable with respect to first-order averages. When loss rates fluctuate, delivery failures will affect higher layers. Measurements from our office environment

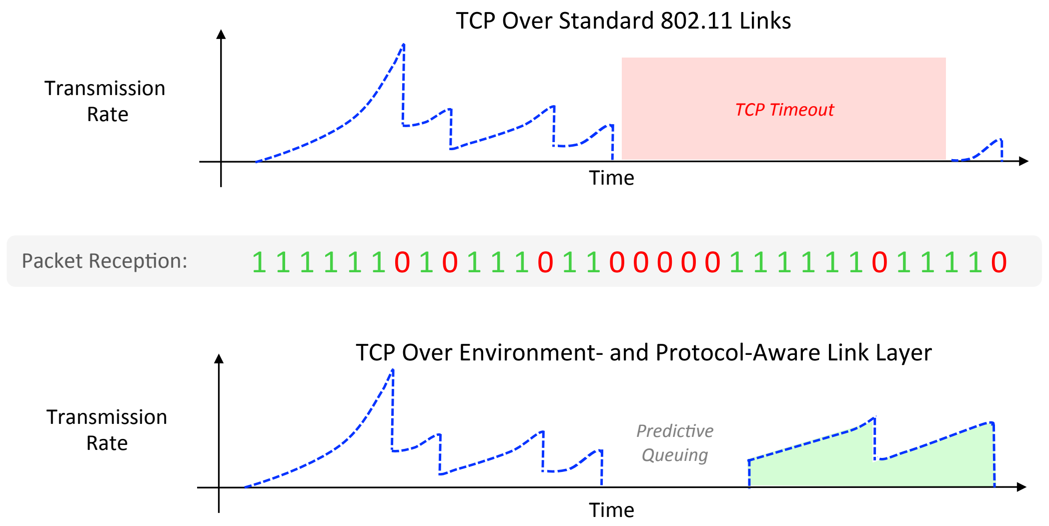


Figure 3.5: When temporary link outages cause all packets in TCP's transmission window to be lost, the connection suspends until TCP timeout. Holding packets temporarily at the link layer prevents this effect by preempting TCP until conditions improve. At that point, the protocol can wake up quickly and resume transmission.

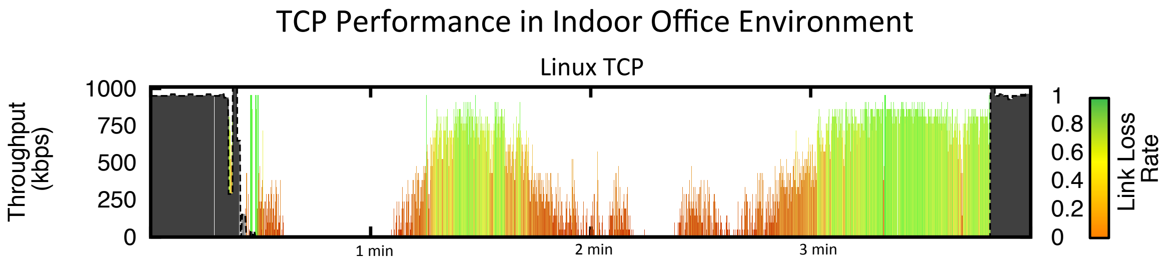


Figure 3.6: A TCP connection's throughput is shown on top of link-loss rates throughout the office walking-tour. TCP times out due to the presence of fire-proof doors and does not resume for several minutes.

show this effect in Figure 3.6. Green and red bars depict loss rates for probe packets, and the shaded black curve overlays the throughput of a TCP connection. In this case, a timeout early in the walking tour prevents large sections of high-quality link conditions from being used. Overall throughput is low, as is network utilization.

3.3.2 Dictionary Training

Our link model implements prediction by classifying link states that precede outages apart from those that do not. Here, we first illustrate the kinds of measurement errors that affect probe data and complicate this procedure. Consider a linear SVM trained to classify raw link-measurements into *noisy channel* or *temporary outage* classes using the following samples:

Noisy Channel Training Data: (intermittent 0's)	*Temporary Outage* Training Data: (long strings of 0's)
$x_1 = [1101010111]$	$x_4 = [0000111111]$
$x_2 = [1010101110]$	$x_5 = [1000011111]$
$x_3 = [1101010101]$	$x_6 = [1111110000]$

Test Observation:

$$x_7 = [1111000011]$$

Intuitively, the test observation belongs to the temporary-outage class, but the SVM will label it otherwise. This is because an SVM produces unexpected results for any input vector that is not a linear combination of the training samples in its class. Common variations such as pattern shifts cause this failure case to occur, and such effects are commonplace when measurements cannot be tightly synchronized to the physical environment. However, if raw data is first latched to a set of candidate features, a linear relationship between training and test samples is more-likely to hold within the SVM. In this example, latching the test sample to its closest training sample by inner-product distance corrects the problem.

To find dictionaries of link-state features, we implement dictionary learning offline using the LASSO algorithm. For the UAV environment, training data is captured from a single 20 minute flight and pooled over nodes in each ground-receiver group. From the subway, training samples are collected over one inbound and one outbound trip between Harvard and Charles/MGH stops. From the indoor office, we train on traces from four laps of the walking tour.

LASSO can be configured to learn *templates* that represent link state with a single feature, or *primitives* that capture link states as a combination of features. Template-learning implements a similar link representation to that of the Markov-based link models in the literature. It is instantiated by setting the dictionary-size parameter m to be large and sparsity-penalty λ to be low, for example: $m \approx 200$ and $\lambda \approx 0.1$. In contrast, primitives are learned by setting the dictionary size to be small and the regularization parameter to be large: $m \approx 20$ and $\lambda \approx 4$. This latter process causes dictionary training to extract only the most prominent pattern from each training sample.

3.3.3 Learned Link Primitives

Here, we show link primitives discovered by training dictionaries for each environment independently. We configure LASSO with dictionary size $m = 20$ with $\lambda = 4$ to learn link state primitives for each type of receiver: UAV field nodes, UAV ground-structure nodes, UAV forest nodes, indoor office nodes, and subway nodes. Figure 3.7 plots frequently-occurring primitives found at each receiver.

Across UAV ground-receiver groups, link primitives are similar. They capture both abrupt and gradual transitions, intermittent packet drops, and temporary out-

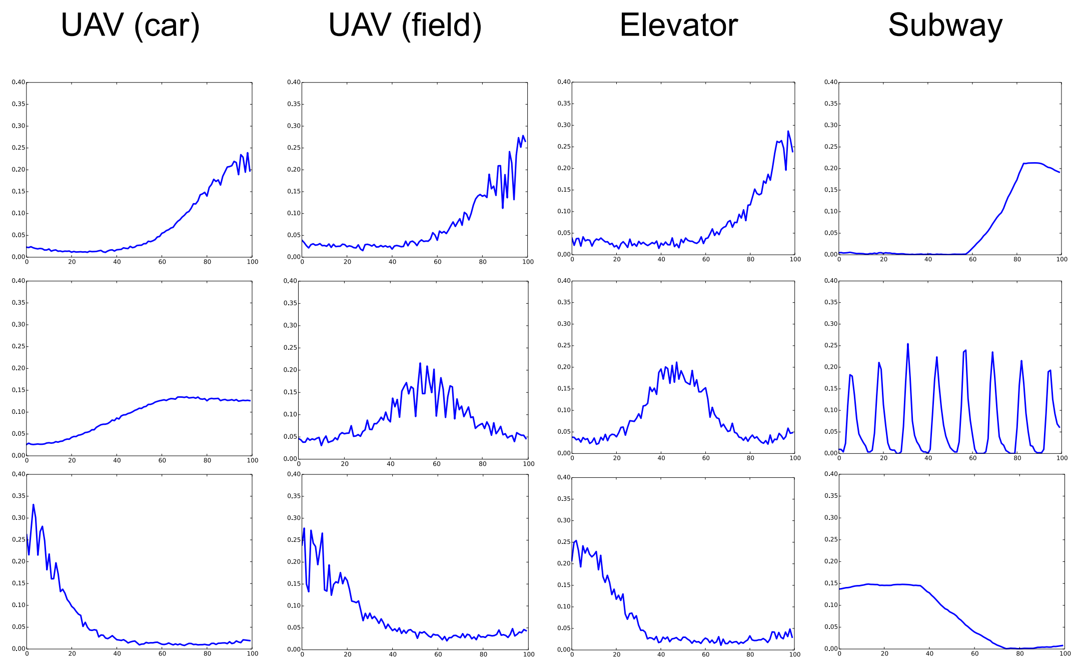


Figure 3.7: Link primitives learned across different environments are similar. Each receiver captures link transitions from good to bad, and *vice versa*. In addition, subway links include bursty patterns caused by network queuing delays. When D is trained using a link-state *primitive* configuration such as $m = 20$ and $\lambda = 4$, networks effects like this are separated from link transitions. *(Figure courtesy of Marcus Comiter)*

ages lasting 10's of milliseconds. When examining the number of measurement samples associated with each dictionary feature, we find that probe measurements from field and forest nodes most-often exhibit long runs of delivery successes in-a-row, as well as smooth, gradual changes in link quality. This corresponds to the fact that these nodes have the best connectivity within our experiment setup, and are far from ground structures that can suddenly occlude line-of-sight. In contrast, receivers in the ground-structure group often exhibit abrupt transitions, since buildings block line-of-sight until the UAV is immediately overhead.

The indoor office and subway environments share many primitives in common with our rural UAV scenario. We see temporary outages as well as abrupt link transitions. In the office, we see a greater degree of noisy variation, possibly due to 802.11 cross-channel interference. In the subway, features appear with regular 10ms-long oscillations in delivery. Based on experiment logs, we find that these correspond to bursty inter-arrivals caused by queuing at hops between transmitter and receiver.

3.3.4 Isolating Network Effects Using Linear Combination

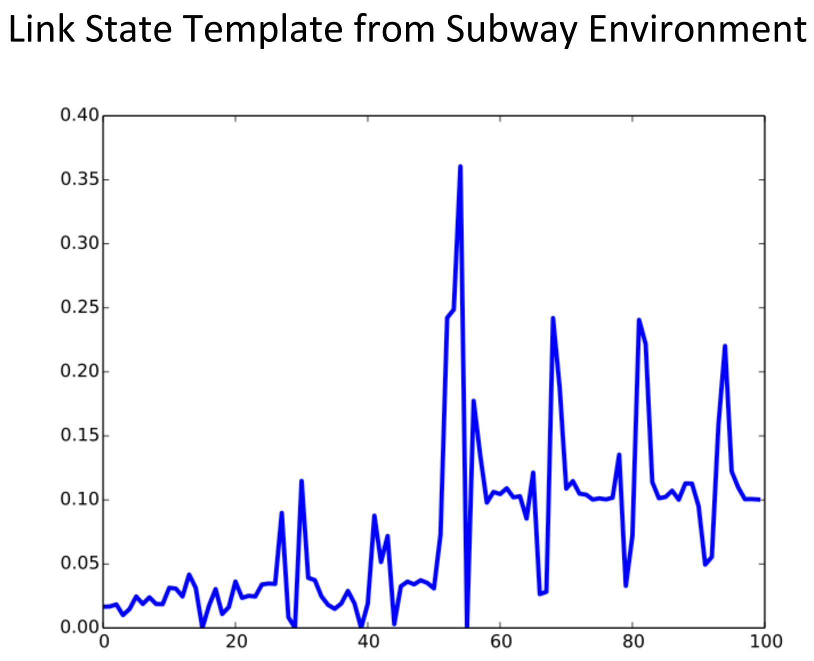


Figure 3.8: A link state template learned from subway data by setting $m = 200$ and $\lambda = 0.1$ is shown. The effects of wireless link-loss and network delays are captured together. *(Figure courtesy of Marcus Comiter)*

The expressive power of a link model based on linear combination is evident in the subway environment. Figure 3.8 plots a typical link state learned from subway data using a *template* model like that of Markov formulations in the literature. Unlike the primitives shown in Figure 3.7, templates superimpose wireless link transitions and queuing delays together in a single feature. As result, we see that link templates are environment-specific, unlike link *primitives*. This shows that, in addition to their portability, primitive-based models can implement powerful pre-processing like clutter suppression – to remove network-queuing effects from these measurements, we need only truncate the corresponding coefficient.

3.3.5 Outage Prediction by Classification

We now describe our outage-prediction model. We train a dictionary of link primitives with $m = 20$ offline using LASSO, pooling training data from all environments together. Our prediction target is defined as any sequence of 192 probe measurements containing any delivery failure. Since TCP can recover quickly from a single packet drop using fast-retransmit, this target captures outages in two back-to-back data-packet transmissions when MTU=1500B, modulation is 1Mbps, and ARQ retries are the Linux default 7.

A linear SVM classifier implements prediction by separating sparse feature vectors that precede gaps from those that do not. SVMs require both measurement vectors and labels corresponding to gap/no-gap classes for training, so we annotate the link traces used in dictionary training to build predictors for those environments. During SVM training, 20% of data is randomly held aside as a test set so that SVMs can be optimized over an intercept parameter. For new environments, SILQ nodes can collect probe data and train SVMs on-platform. All sparse-encoding operations that occur on-platform use Matching Pursuit (MP) for its computational simplicity.

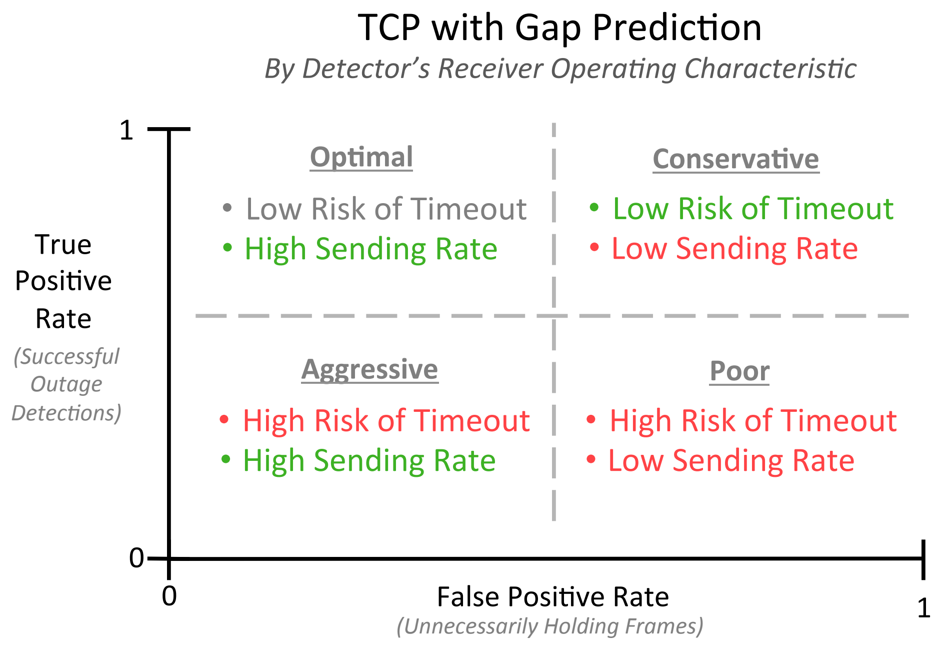


Figure 3.9: A gap detector's sensitivity affects realized TCP throughput. Aggressive detectors lead to high sending rates and a high risk of timeout due to wireless loss, while conservative detectors never time out but tend to under-utilize the link.

3.3.6 Tuning Gap Predictors

We can tune our gap predictor by setting a threshold on the inner product between the SVM separating plane and sparse feature vectors. This effectively pushes the separator toward the outage class, or away from it to set aggressiveness. This sensitivity ultimately affects realized TCP throughput under predictive queuing. Figure 3.9 illustrates this relationship. Aggressive predictors produce few alarms overall, leading to low false positive rates but also low true positives. For TCP, the result is a high sending rate, but increased risk of timeout. In contrast, conservative predictors raise many alarms and produce high true positive rates but also many false positives. TCP timeout rarely happens, but links go underutilized in good conditions.

We explore this relationship using the experiment methodology presented in Sec-

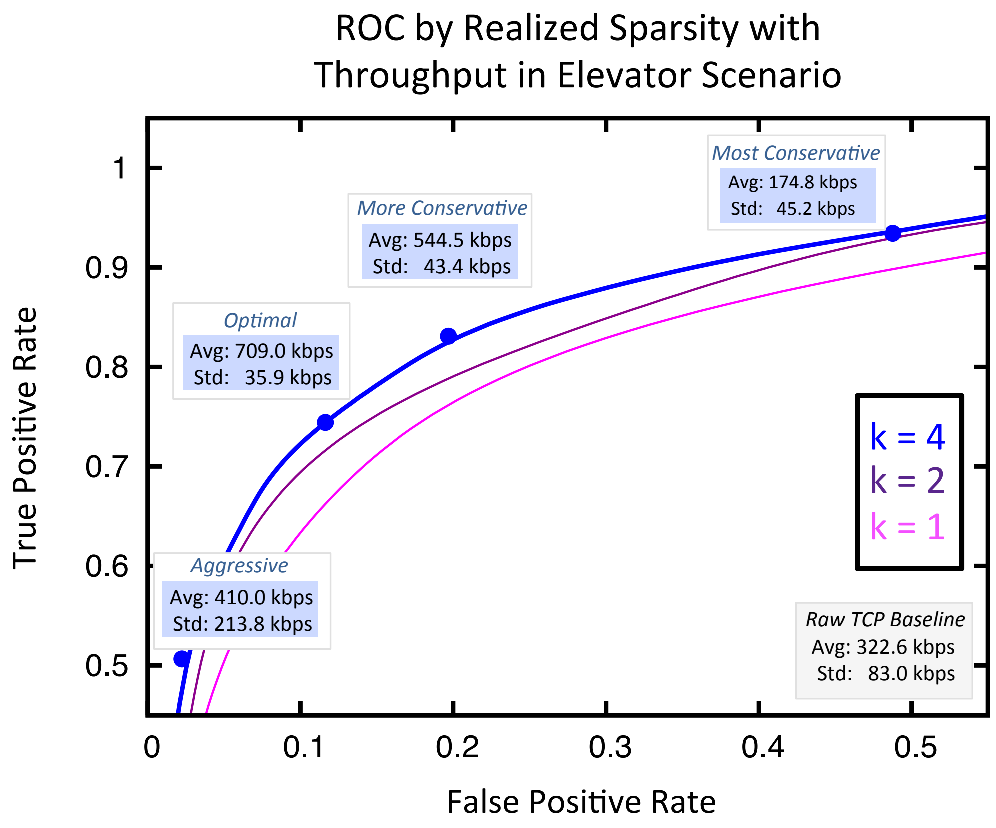


Figure 3.10: ROC profiles for different encoding sparsity k settings are shown. For $k = 4$, the illustrated relationship to TCP throughput is reproduced experimentally.

tion 3.5, empirically balancing true positives and false positives to maximize TCP throughput in the indoor office environment. Figure 3.10 shows that aggressive predictors realize poor throughput on average, with a high standard deviation. This is because timeouts occur frequently, dropping TCP throughput to zero for long periods of time. Since users may sometimes get lucky and experience good throughput, variation is significant. Overly-conservative predictors produce the opposite effect and realize *lower* throughput, but with low variation. Ultimately, these conservative predictors do not send often-enough when the link is usable, paralyzing connections.

Figure 3.10 also shows the Receiver Operating Characteristic (RoC) profile for predictive models with different encoding sparsities k. We see that prediction accuracy is dependent upon finding the right configuration of model parameters – dictionary size m , training regularization penalty λ, and encoding sparsity k. For example, if we set $m = 20$, but $k = 2$, then the detector is strictly *worse* than setting $k = 4$. We therefore configure our link model by searching over many possible parameter settings to maximize a statistical criterion: across scenarios, we choose the configuration with highest gap recall, provided non-gap recall is above a fixed threshold of 0.75 to ensure enough sending opportunities. The best performing model configuration has $m = 20$, $\lambda = 4$, and $k = 4$.

3.3.7 Transferring Features

While some link state primitives are unique to a particular environment, many are shared among scenarios. Intuitively, these represent common characteristics like range effects in large open spaces, or abrupt transitions due to occlusions. In this section, we empirically test the universality of link features across environments. To do this, we learn primitives from training data collected in a single environment, and use them as the basis for a gap predictor trained in a *different* environment. Table 3.1 shows the resulting gap-prediction recall from this experiment. Results verify that primitives are portable, transferring well across different networks like the 3G subway network and the single-hop 802.11 UAV network.

We performed a similar experiment using link-state templates and found that they do not transfer well at all. This confirms the results of Section 3.3.4, and shows that models directly match link sequences are inherently network-specific. In contrast, those that use linear combination to separate network and environment effects are

Table 3.1: Outage Prediction when Transferring Features

		SVM Training and Testing Environment		
		Indoor Office	Urban Subway	UAV
Dictionary	Indoor Office	0.82	0.79	0.83
Training	Urban Subway	0.81	0.73	0.81
Environment	UAV	0.82	0.74	0.81

Gap recall is shown when porting link-state primitives between environments. Unlike template features that capture link states in a single pattern, primitive features used in linear-combinational models generalize across networks and environments.

more-portable.

3.4 SILQ Architecture

This section describes the SILQ architecture and its computational performance on off-the-shelf devices. At runtime, SILQ nodes probe outgoing links by sending UDP packets at a constant rate. Reception reports are bounced back to senders, piggybacked on probes traveling in the opposite direction. Every 100ms, each node computes a link prediction using the prior w measurements. The parameter w trades off measurement granularity for bandwidth, and is investigated in Section 3.5.2. If a link outage is predicted, SILQ holds data packets until a fresh prediction indicates otherwise.

3.4.1 Link-Measurement Protocol

SILQ's link measurement protocol is illustrated in Figure 3.11. Each node maintains a bit vector $\texttt{Link}_{in}$ of the last w incoming probes. The most recently-received probe is also stored, denoted $\texttt{Seq}_{\texttt{in}}$. Each arriving probe's sequence number is subtracted

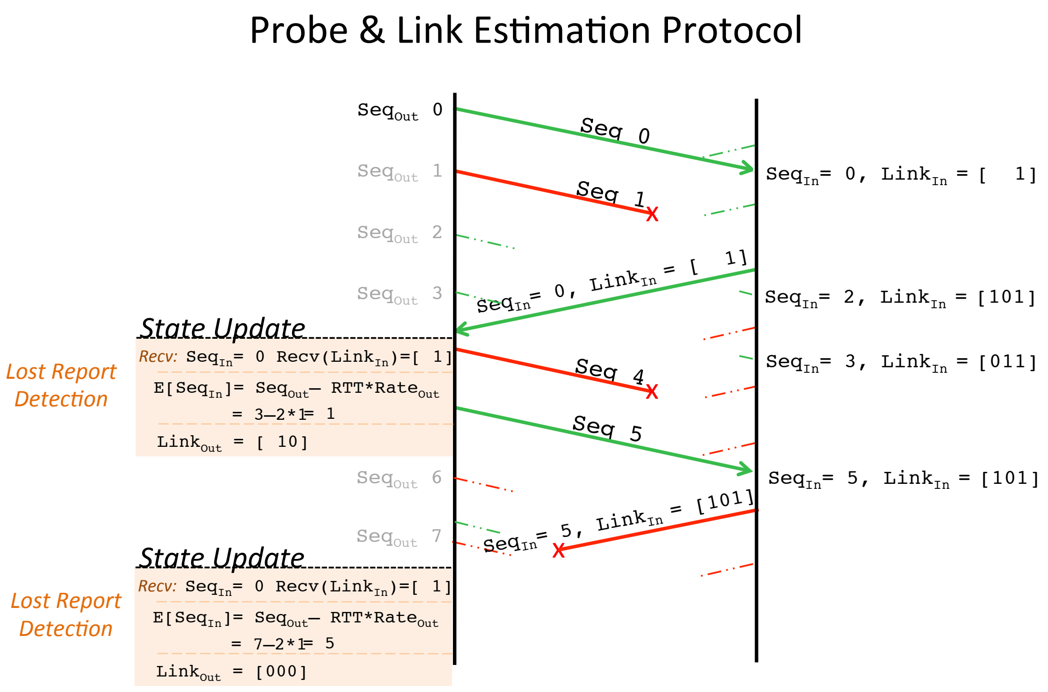

Probe & Link Estimation Protocol
Seq_Out 0
Seq_Out 1
Seq_Out 2
Seq_Out 3
Seq_Out 6
Seq_Out 7
Seq 0
Seq 1
Seq 4
Seq 5
Seq_In = 0, Link_In = [1]
Seq_In = 2, Link_In = [101]
Seq_In = 3, Link_In = [011]
Seq_In = 5, Link_In = [101]
Seq_In = 0, Link_In = [1]
Seq_In = 5, Link_In = [101]
State Update
Lost Report Detection
Recv: Seq_In = 0 Recv(Link_In)=[1]
E[Seq_In] = Seq_Out − RTT*Rate_Out = 3−2*1= 1
Link_Out = [10]
State Update
Lost Report Detection
Recv: Seq_In = 0 Recv(Link_In)=[1]
E[Seq_In] = Seq_Out − RTT*Rate_Out = 7−2*1= 5
Link_Out = [000]

Figure 3.11: SILQ piggybacks incoming reception reports on outgoing probes so that predictions can be made for *outgoing* links. Reception reports consist of a bit vector marking incoming probe deliveries. Nodes detect lost reception reports using a round-trip-time estimate.

from $\texttt{Seq}_{\texttt{in}}$ to detect delivery failures before updating $\texttt{Link}_{in}$ and $\texttt{Seq}_{\texttt{in}}$. These two state variables comprise a reception report; UDP probes therefore require 69B for a 3B bitmap with sequence numbers as 2B `unsigned shorts`.

Nodes also maintain a round-trip-time estimate `RTT` to account for lost reception reports. For example, if only the first of $w = 20$ reports successfully bounces back to the sender, that node must compute the number of reports that *should* have arrived: $\texttt{Seq}_{\texttt{in}} - \texttt{RTT} * \texttt{Rate}_{\texttt{out}} = 19$. This protocol means that bi-directional probing captures asymmetric link effects, except when many reception reports are lost at the end of an interval. Then, we make the conservative assumption that the link is off in both directions. In practice, we find that disruptions due to `RTT` fluctuation resolve after one or two missed 100ms windows, even on a crowded carrier network.

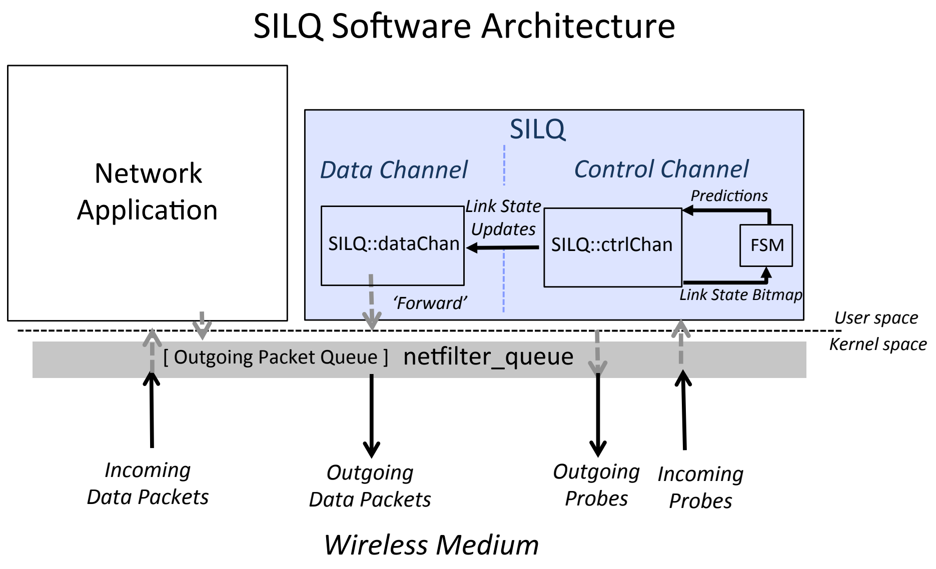


Figure 3.12: The SILQ architecture consists of logically separated data and control channels that isolate link-measurement from data-forwarding. In our testbed, both channels access the same wireless medium using the same interface.

When probes share the wireless medium with data packets, they reduce its effective capacity, creating a tradeoff between measurement fidelity and maximum data rate. For example, sending a probe every 5ms consumes 22% of a 1Mbps link, while probing every 10ms consumes 11%. In harsh environments, we will show that the throughput benefits of high-quality predictions far outweigh their overhead. In contrast, when links are more-stable and easier to predict, additional capacity outweighs lower prediction accuracy. We explore this tradeoff experimentally in Section 3.5.

3.4.2 Software Architecture & Profiling

SILQ's software architecture is shown in Figure 3.12. Each node implements a *data channel* controller and a *control channel* controller. The data channel controller simply holds or forwards packets according to the current link-state prediction. The control channel is responsible for probing and providing up-to-date data to a *finite-state machine (FSM)* link-model module. The logical separation between data and control channels keeps link measurement protocols separate from data delivery protocols, simplifying both. For all results in this chapter, both channels access the same wireless medium using the same interface.

SILQ is currently implemented in user-space, and relies on the `netfilter_queue` Linux module to intercept and administer packets within the kernel. The data channel controller maintains a queue of packet identifiers provided by `netfilter_queue` and issues `NF_ACCEPT` verdicts when it is safe to forward them. SILQ's current implementation scales to 10Mbps. At this rate, our SVM predictor is retrained to identify gaps lasting 20ms. In practice, we find these predictions slightly more aggressive, and tune the SVM to favor gap recall as much as possible.

Within the control channel, FSMs are modular, making it easy to extend SILQ with new link models. For our purposes, we implement *Loss Rate Threshold*, *Heuristic Hold*, `NO_OP` (i.e. always forward), and *Sparse Coding* FSMs. In all but the `NO_OP` FSM, trivial link vectors with all 0's always predict outage, while those with all 1's predict no-outage. For non-sparse-coding FSMs we include a "conservative" parameter that requires back-to-back no-outage predictions before turning a bad link back on. This parameter is needed to stabilize loss-rate and heuristic predictors enough to achieve meaningful throughput. For our sparse coding FSM, we optimize for stable predictions using the previously discussed statistical methods.

The sparse coding FSM uses non-negative Matching Pursuit (MP) to compute sparse feature representations, projecting the results onto an SVM separating plane to implement prediction. MP relies primarily on inner-product computations and has $O(mw)$ complexity, with m the number of dictionary atoms and w number of probes in an interval. Additional non-negativity constraints have been shown to improve MP's performance with respect to SVM classification accuracy [74].

Profiling SILQ on consumer-grade mobile devices, we observe 2% average CPU utilization on a 1.86 GHz Intel Core 2 Duo processor and 6.9% utilization on a 1.0 GHz ARM Cortex A8. Less than 1% of SILQ's runtime is spent computing predictions, with most time spent servicing packets. Using a dictionary of 30 atoms for $w = 20$ probes per interval, SILQ's memory footprint is only 972KB. Both of the devices tested are well below current smartphones in clock rate and number of processing cores, making SILQ widely deployable.

3.5 System Evaluation

In this section, we evaluate SILQ by measuring throughput in the UAV, office, and subway scenarios. First, we train dictionaries and SVMs on our probe traces. Then, we deploy SILQ in each environment, measuring the throughput of a TCP connection as data is transferred to a remote server. To fairly compare throughput over varying links, we compute the *realized* link capacity in post-processing: using SILQ's probe delivery logs, our throughput metric only counts periods with back-to-back successful probe deliveries as valid sending windows, since they reflect good link conditions spanning the transmission time of an MTU data packet.

In all experiments, we use the Linux default TCP CUBIC. We enable the SACK-enhanced F-RTO option for resilience over wireless links, use 9 keep alive probes, and allow 15 tranport-layer retries per packet. Underneath that, our wireless adapters use 7 link-layer ARQ retransmissions for data packets. We disable ARQ for probes.

We compare SILQ against raw TCP and predictors based on a loss-rate-threshold and a simple heuristic that holds data packets if all probes in a previous window are lost. The loss-rate-threshold predictor is similar to physical-layer rate-adaption mechanisms that make a steady-state assumption about delivery probability. The heuristic is the simplest possible method to hold data packets during the long out-of-range regions seen in our UAV and subway tests. We experimentally tune the loss-rate threshold and "conservative" parameters for both methods to achieve maximum throughput.

3.5.1 UAV Throughput in Emulation

Due to logistical barriers related to personnel, budget, and weather, we validate throughput in the UAV scenario using a lightweight UAV link emulator. During emulated flights, SILQ runs on our UAV payload and ground nodes. Probe and data packets are sent over wired interfaces to a central emulation controller that replays probe traces collected in the field. Emulated traces are held out of training datasets used for dictionary and SVM training.

The emulation controller forwards packets to their destination only if a packet reception was recorded in the field trace. We emulate a 1Mbps link and compare 1500B MTU data packets against 12 back-to-back 1ms trace measurements to reflect a 12ms transmission time. An error in any trace measurement causes the larger data packet to be dropped. The emulation controller simulates both hardware ARQ and

CSMA.

Since ground-structure nodes exhibit rapid link transitions that are hard for loss rate and heuristic methods to predict, these methods perform best when tuned to a high conservative parameter and low probe rate. Loss rate predictions realize a 10% throughput boost over raw TCP's 528kbps throughput. Heuristic predictions underperform raw TCP in both throughput and performance variation. Sparse coding performs best overall by using a high probe rate to make accurate predictions, and achieves a 20% throughput boost over baseline TCP.

For field and forest nodes, reception rates are high for a large portion of flight time. Consequently, baseline TCP performs well since it can send data at the highest possible rate of any method. Both heuristic and loss-rate predictions under-perform TCP even when probe rates are reduced to allow maximum data throughput. For loss-rate predictions, poor performance is caused by under-utilization at radio range edges: once link loss rises above the method's threshold, data packets will stop sending even when intermittently-spaced drops could be fixed by ARQ. This behavior occurs because loss rate maps many link states to a single coarse metric – both smooth degradations and abrupt changes are treated similarly.

In this scenario with few unexpected changes in link quality, SILQ performs similarly to TCP in terms of aveage throughput, but with an extremely low variation of only 19kbps across flights. This shows that SILQ predictions reduce variance and maintain throughput when the link is relatively stable. We will next show that SILQ's gains are much larger when links vary wildly in our office and subway environments.

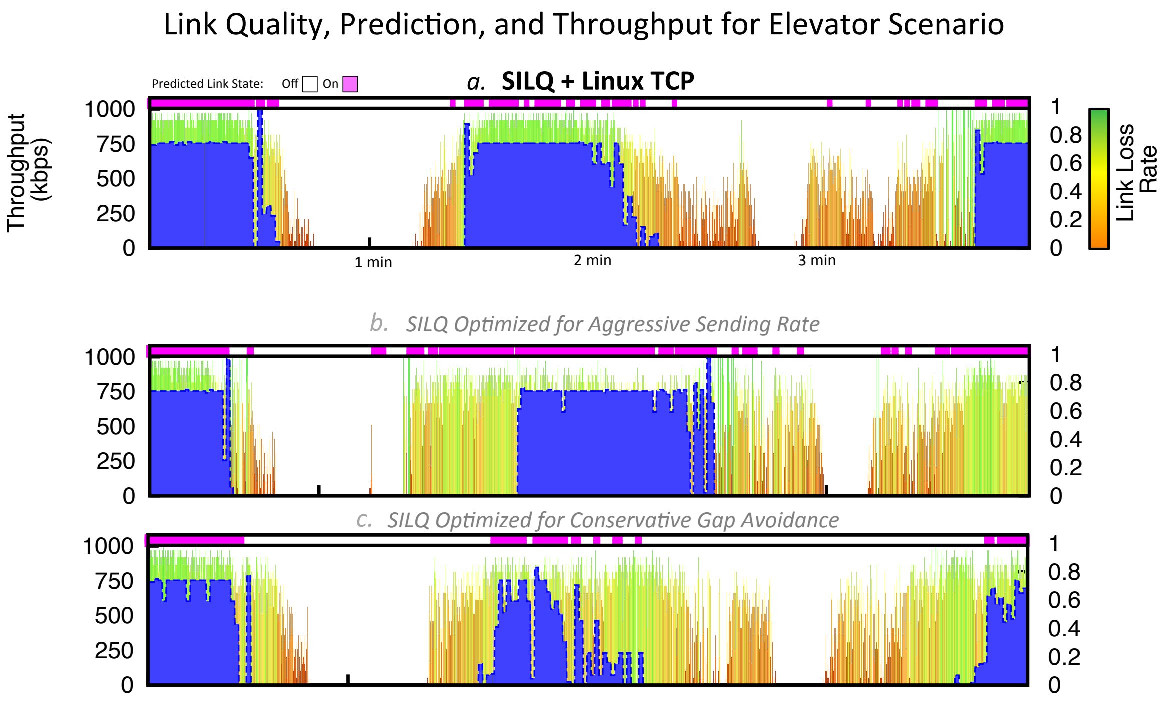


Figure 3.13: Plot (a) shows TCP atop SILQ's predictive queuing mechanism. Timeouts are avoided by holding data packets when the link is off, causing TCP connections to react quickly when the link is restored. Plots (b) and (c) show TCP connections over SILQ when link predictions are optimized to forward data more or less aggressively. Aggressive sending makes use of more transmission opportunities, but carries a higher risk of timeout as shown. Conservative sending can miss large portions of usable link, but will rarely time out.

3.5.2 Indoor Elevator/Office Environment

We run SILQ, raw TCP, heuristic, and loss-rate models in our indoor office environment. Figure 3.13 plots loss rates with faint green and red bars, while TCP throughput is overlaid in blue. SILQ's link-state predictions are denoted by markers at the top of the plot, with a pink marker indicating that the link is predicted to be on and a white marker predicting that the link will be off.

When compared to the performance of the baseline TCP connection shown in Figure 3.6, SILQ produces a throughput gain of 2x on average over 5 runs. This gain is driven by SILQ's rapid response once links are restored after outages. We

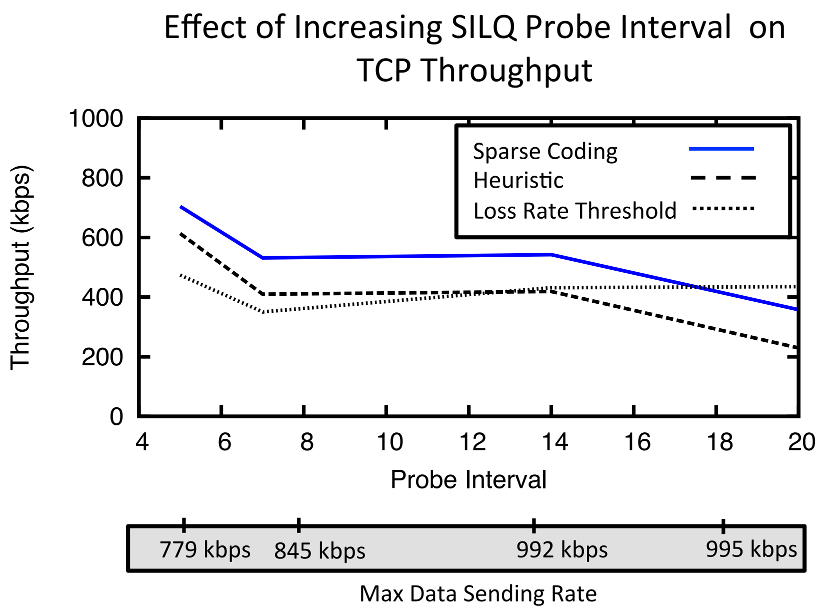


Figure 3.14: Average throughput is compared between prediction models in our indoor office scenario as probe rate is reduced. Sparse coding is resilient to lower probe rates, though the most accurate predictions yield the best TCP throughput.

also show the performance of SILQ under predictors tuned according to the different True Positive/False Positive tradeoffs of Section 3.3.6. The more aggressive predictor shown in Figure 3.13 (b) sends often, but eventually times out. In contrast, the conservative predictor in Figure 3.13 (c) underutilizes the link and misses a large portion of sending opportunities.

In Figure 3.14, we compare SILQ with the heuristic and loss-rate predictors as we reduce the probe rate. In these experiments, we found it difficult to tune either comparison method to perform well over the entire walking tour – neither could cope with both abrupt link transitions as we passed through fire-proof doors *and* the gradual link degradation seen in the elevator using a single configuration setting. However, our sparse-coding predictions degrade gracefully when probe rates are reduced. This makes it easier to find a configuration that works across variable link conditions.

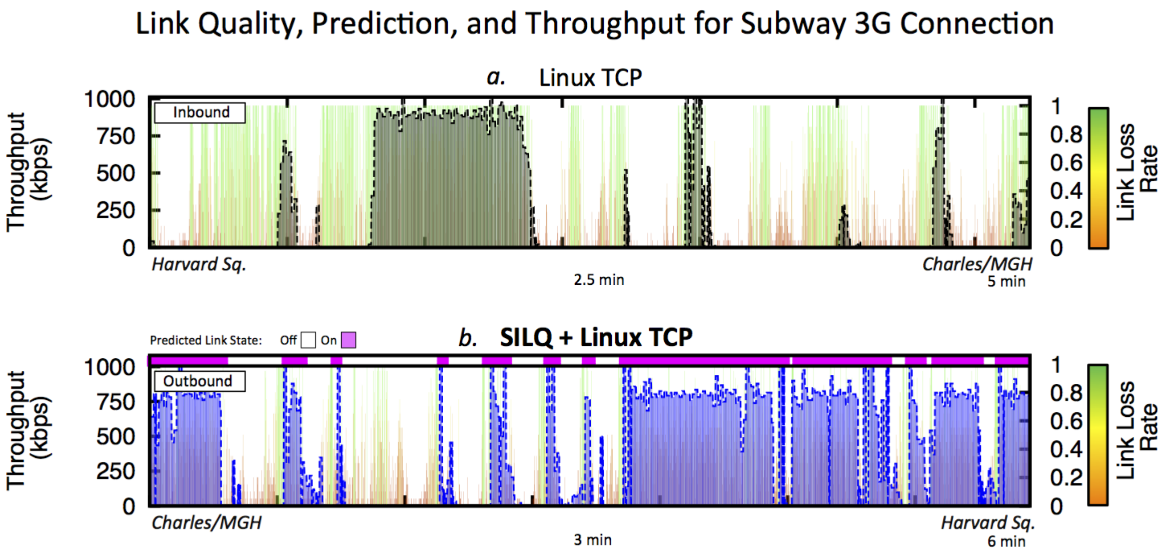


Figure 3.15: A comparison of a raw Linux TCP connection to TCP atop SILQ is shown during inbound and outbound subway rides over a 3G cellular network. Raw TCP times-out quickly, missing later sending opportunities. With SILQ, timeout is avoided and throughput increases by 4x.

Also, since most of SILQ's computation time is spent servicing probes, decreasing the probe rate from once-every-5ms to once-every-10ms correspondingly drops CPU utilization in half.

3.5.3 Urban Subway Environment

Our subway scenario validates SILQ "in the wild." This case presents a completely uncontrolled environment with a great deal of complexity. Mobile nodes experience fast changes in velocity, sudden occlusions from tunnels, spotty connectivity to 3G repeaters, and competeing wireless traffic. This comparison demonstrates the strength of SILQ's adaptable predictions and end-to-end design, since no alterations or configuration of the underlying network were needed for the system to function.

Figure 3.15 compares TCP with and without SILQ over subway rides between the Harvard Square and Charles/MGH stops. Note that Figure 3.15 (a) represents an inbound trip from Harvard to Charles/MGH, while Figure 3.15 (b) represents an outbound trip between the two stops. We see that baseline TCP times out and fails to utilize short, more-transient sending opportunities beyond the Harvard station. In contrast, SILQ enables connections to take advantage of almost every available sending window. The result is a 4x gain in throughput over a five minute train ride.

3.6 Summary and Conclusion

The link models and on-platform SILQ implementation in this chapter demonstrate several important points:

- The precise statistics of outage prediction change throughout everyday use-cases, depending on effects like environment and mobility. For example, rural predictions are driven by range and location, while indoor predictions are based on sudden occlusions and interference.
- Clustering is capable of learning fundamental link states and supports visual comparisons between models of different environments.
- Packet losses in wireless networks with different technologies and environmental conditions have similar latent structure. When a modeling framework like sparse coding can separate network-specific effects from environment-driven wireless effects, link models generalize beyond their training environment.
- Link predictions driven by sparse coding and linear SVM classification can be implemented efficiently on resource constrained devices.

- The performance gains realized by an adaptive link-layer are huge when compared to the baseline static network stack. In common scenarios like offices and subways, throughput improves by up-to 4x, while performance variation drops by the same factor.

Chapter 4

Workload Modeling for Predictive DVFS in Low-Power Circuits

We show how state-models based on sparse feature representation extend to circuits, with the goal of reducing power consumption through predictive dynamic voltage and frequency scaling (DVFS). We predict lulls in instruction throughput during which DVFS can be applied without degrading application performance. Unsupervised clustering finds patterns in hardware performance counters that reflect recurring workload states in applications like user-driven web surfing. When compared to regression models trained on raw counter values, we show that state predictions can be made 3x further in-advance using sparse coding. Since longer-range predictions give more time for chip adjustment, this method better-supports low-cost off-chip voltage regulators [75].

4.1 Reducing Power Consumption with DVFS

DVFS promises to improve operating efficiency by adapting power consumption to workloads at runtime. This flexibility is important since energy consumption varies widely by user and application mix [76] [77].

Total on-chip power consumption is a sum of both *static* and *dynamic power*. Static power refers to baseline dissipation regardless of circuit activity, such as energy lost to the environment by flip-flops. Dynamic power instead refers to energy consumed by the charging and discharging of gates along circuit pathways – this quantity is a function of workload. Dynamic power is approximated by the following relationship:

$$P_{\text{dyn}} = CV_{\text{dd}}^2 f a$$

$P_{\mathbf{dyn}}$	Dynamic power
C	Capacitance
$V_{\mathbf{dd}}$	Supply voltage
f	Frequency
a	Switching-activity rate

Since P_{dyn} is quadratic in supply voltage, DVFS can save power by dialing back V_{dd} when appropriate.

To avoid noticeable performance degradation, DVFS must be applied selectively when chip workloads are light. This is because lower supply voltages lead to a longer signal propagation time through circuit pathways. Clock frequency must therefore be slowed to compensate, resulting in a lower maximum-instruction-throughput when the chip is in its low-power state.

We show that modeling workload patterns using hierarchical sparse coding enables accuracte predictions for brief 500μs lulls in activity, during which DVFS can be applied safely. We target predictions that are long-range to support commercially-available DVFS hardware that requires 100μ's for adjustment.

4.2 Hardware Performance Counters

Hardware performance counters capture architecture-level events like committed instructions, data table hits, misses, flushes, etc. Workload prediction methods using this data most-often fit regressions to raw counter values [78] [79] [80]. However, we will show that regression accuracy degrades at long ranges when future states are not a simple extrapolation of these values. Instead, our method will use sparse representations to capture stable signatures of chip state and classify these to predict workload lulls.

This workload prediction scenario is related to the well-studied problem of phase detection. Phase detection is often motivated by the desire to identify large stable regions of a workload so that configuration-adjustment overheads can be amortized. Detection techniques like working set signatures, basic block vectors, and conditional branch counters apply a threshold to one or more hardware counters to identify deviations relative to long-term variance [81]. While more-sophisticated techniques are necessary for long-range predictions, phase detection *can* serve a complimentary role. For example, large shifts in dynamic range can alert predictors to a change in phase that requires a model update.

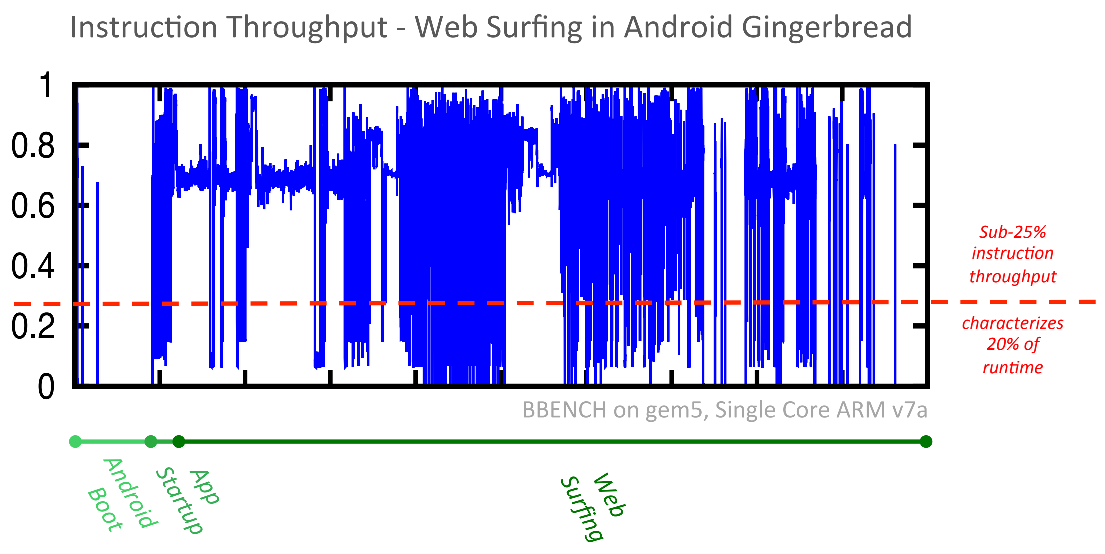


Figure 4.1: The rate of committed instructions per cycle intermittently drops below 25% over the course of the BBENCH workload. In aggregate, these lulls make up nearly 20% of execution time and are targets for predictive DVFS. Lulls occur for two reasons: when waiting on data I/O, and during computations due to architecture-level operations like instruction-cache misses. We use hierarchical sparse coding to find predictive signatures for this latter class of lulls, which make up roughly 6.3% of the surfing phase.

4.3 Experimental Setup and Measurement Data

4.3.1 Device and Benchmark Workloads

We use *gem5* to collect data for modeling. *gem5* is an architecture-level simulator with full-system support, including frame buffer rendering and an interactive shell [82]. Snapshots of counter values are taken every 500μs from a simulated 1.0 GHz ARM v7a chip running the Android operating system.

Our primary workload is the BBENCH benchmark [83] running atop Android Gingerbread. Figure 4.1 plots committed instructions/cycle for three phases of activity: OS boot, web browser startup, and a web surfing phase. During surfing, web sites are loaded from off-chip using Android's built-in browser, and Javascript code simulates user link clicks. We see that instruction throughput drops below 25% during nearly 20% of the surfing phase, making these intervals appropriate for DVFS.

Lulls in instruction throughput occur in two sets of circumstances. First, they arise while waiting on web page I/O. These idle periods can be captured by simple heuristics to realize savings in I/O-dominated workloads. Lulls also occur during computation-dominated periods, due to architecture-level events like instruction-cache misses. This latter class of intermittent breaks in computation requires more-precise state modeling, and makes up 6.3% of BBENCH. These are our prediction targets.

To generalize gains from our approach to other workloads, we also report prediction statistics for the ASIMBENCH/Moby benchmark suite [84]. This set of workloads includes examples of a game, audio and video playback, and document manipulation applications running under Android ICS. Though ASIMBENCH captures additional application behaviors, it lacks the simulated user interactions of BBENCH that cause workload phases to recur over time. We therefore present these limited results with the caveat that user interaction must be incorporated to properly assess power savings.

4.3.2 On-Chip Counters

Our state model learns workload-specific signatures in chip states observed over multiple measurement windows in time, and multiple counters across the chip. We sample all available counters and compute deltas from previous values every 500μs. These are then normalized to lie between 0 and 1, ensuring that no counter dominates learning and coding computations due to larger dynamic range.

Since there are layout and I/O-related costs for sampling counters, we use *gem5* to determine the minimum number of counters needed to support prediction. The selection of counters available on a chip is usually determined by designers, based on desired monitoring and debugging data – for large projects, counters can number in the tens of thousands, though only $10 - 100$ are typically exposed in the final design. We collect data from 120 counters and calculate covariance and correlation matrices: $A = (X - E[X])(X - E[X])'$ and $\Sigma = (A^{\text{diag}})^{-1/2} A (A^{\text{diag}})^{-1/2}$. Here, $X = [x_1, x_2, ...]$ contains 120×1 measurement vectors x_i as its columns, while A^{diag} has each counter's variance along its diagonal and 0's otherwise. By grouping those counters whose correlation exceeds 0.98 and choosing one representative from each group, we can enumerate statistically equivalent configurations.

By this analysis, we identify 34 different groups from the 120 possible counters available on our ARM v7a-chip. We find, for example, that the number of integer register reads is interchangeable with the number of committed integer operations, though these are collected from different locations on the chip. For completeness, we also applied Principal Component Analysis (PCA), a standard technique for dimensionality reduction. Even though PCA found a lower dimensional basis for our data, that representation required linear combinations of all 34 counters and did not allow us to further reduce sampling costs.

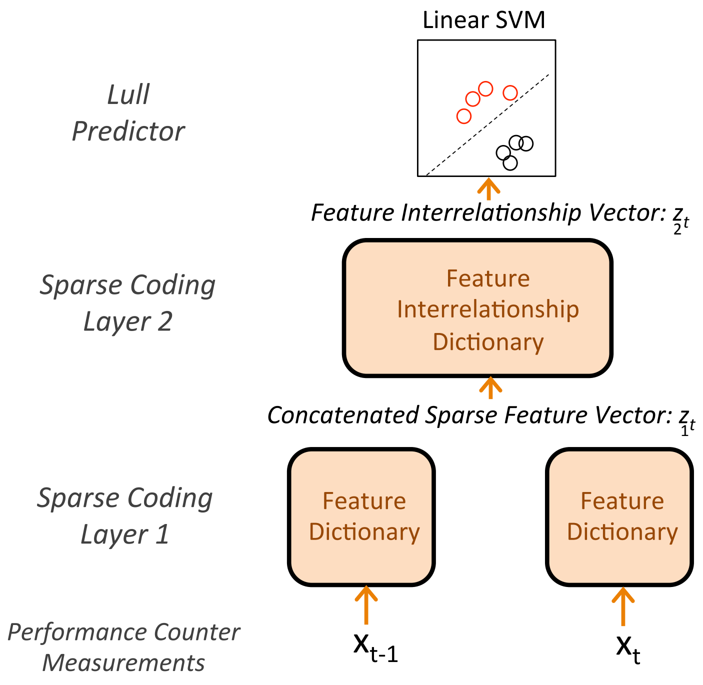

Linear SVM
Lull Predictor
Feature Interrelationship Vector: z_{2t}
Sparse Coding Layer 2
Feature Interrelationship Dictionary
Concatenated Sparse Feature Vector: z_{1t}
Sparse Coding Layer 1
Feature Dictionary
Feature Dictionary
Performance Counter Measurements
x_{t-1}
x_t

Figure 4.2: Our workload model using a two layers of sparse coding to extract signatures over time. A linear SVM then predicts lulls in instruction throughput lasting roughly 500μs.

4.4 A Workload Prediction Model Using Hierarchical Sparse Coding

Our sparse coding model is shown in Figure 4.2. This model captures signatures over time hierarchically, as described in Section 2.2.2.3. Canonical features are extracted from raw counter data at Layer-1, concatenated, and coded again at Layer-2 where feature interrelationships are captured. A linear SVM then assigns a common label to vectors preceding throughput lulls.

4.5 Prediction Accuracy and Power Savings

4.5.1 Accuracy for User-Driven Web Surfing

In this section, we compare prediction performance between hierarchical sparse coding, linear regression, and static heuristics for detecting sub-25% throughput lulls during BBENCH. We define prediction *accuracy* as the portion of all 500 μs windows that yield a correct future-state prediction, lull or not. We also care about false alarm statistics since there is a recovery cost associated with false positives. We therefore report precision and recall for lull-detection. In this context, precision is the number of correctly-predicted lulls over the total number of alarms, while recall is the number of correctly-predicted lulls over the total number of lulls that actually occur during BBENCH.

Sparse coding dictionaries and SVMs are trained using 50% of data from the web surfing phase of BBENCH, and prediction accuracy is calculated on randomly held-out samples by cross-fold validation. We code over $w = 1, 2, 4, 8$ trailing measurement windows of 500μs snapshots to find predictive effects at different time scales. In all cases, we use a two-layer hierarchical model with w independent patches at Layer-1 and a single concatenated signature at Layer-2. We vary dictionary size and sparsity parameters, and report statistics for the best-performing configuration. In Section 4.5.3, we analyze how these configuration settings affect performance.

To compare, we also fit linear regressions of different orders to the instruction throughput metric directly. In Figure 4.3, we show data from an order-8 regression, which performed best. Regressions are computed using a sliding window of measure-

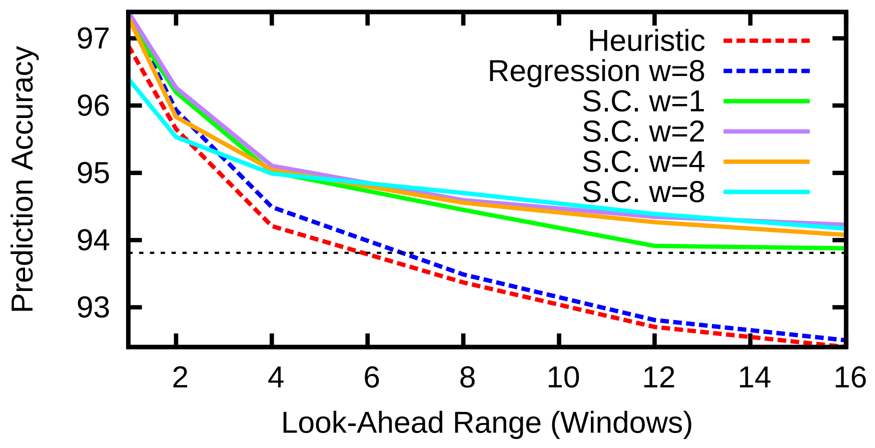


Figure 4.3: Our predictor increases look-ahead range by 3x over regressions fit directly to a single counter and has positive predictive power with up to 8ms heads-up. In contrast, at ranges of 4 measurement windows, raw-data regressions produce more mistaken positive alarms than correct ones. This break-even point is indicated by the dashed line marking the percent of compute-driven lulls in BBENCH.

ment data, and curves are extrapolated and thresholded to implement lull prediction. We also compare to a heuristic that waits until a lull has been observed and assumes that another will follow. This represents the simplest predictor and is a DVFS method that has been tested for commercial GPUs.

Figure 4.3 plots overall prediction accuracy against look-ahead range. First, we see that the static heuristic works only at short ranges, when it can pick up the latter portions of lulls spanning multiple measurement windows. On average, regression only slightly extends prediction range by capturing some dips that can be extrapolated from prior measurements. However, sparse coding consistently has best accuracy for look-ahead ranges of 3 windows or more. Furthermore, we see that regression and heuristics make so many mistakes at long ranges that they are worse than doing nothing. In contrast, sparse coding always has positive predictive power, even at a range of 16 windows. This technique therefore extends look-ahead range by almost 3x. Lull-detection precision and recall are reported for the best-performing models by

Detection Performance
Sparse Coding, w=2

Look-Ahead	Overall Acc.	Prec.	Recall
1	97.3%	0.79	0.76
2	96.2%	0.74	0.60
4	95.1%	0.64	0.50
8	94.6%	0.62	0.33
12	94.3%	0.61	0.21
16	94.2%	0.62	0.16

Figure 4.4: Precision and recall are reported for lull detection as look-ahead range increases. Performance statistics represent the best performing models by overall accuracy. Though recall drops to only 16% at 8ms look-ahead range, overall prediction is still net positive.

overall accuracy in Figure 4.4. We see that our model captures a small set of stable, precise signatures at long look-ahead ranges.

4.5.2 Accuracy Across Workload Types

We next present prediction accuracy for workloads in the ASIMBENCH/Moby benchmark suite including a video game, audio and video playback, and viewers for PDF and Microsoft Office documents. For each workload, we report the percentage of computation time during which instruction throughput is below 25%, and the prediction accuracy using signatures spanning 2 windows. We break statistics out by distinctive workload phases. For example, the *Frozen Bubble* video game has two phases: in the first, application data is loaded and game state initialized; and in the second, the game enters a regular frame-rendering loop. We report performance per phase for regions with at least 8,000 measurements. Benchmark descriptions and prediction accuracy are shown in Table 4.1.

Table 4.1: Prediction Accuracy Across Workload Types

Workload Name	Phase	Description	% Low Instr. Throughput	S.C. Pred. Acc. Look-Ahead=2	S.C. Pred. Acc. Look-Ahead=4
Adobe Reader	All	Display PDF file	11.2%	92.2%	89.9%
Frozen Bubble	*Phase 1*	Initialize and begin game	13.6%	92.0%	88.9%
	Phase 2		64.6%	94.0%	86.0%
k9 Mail	All	Display e-mails	19.3%	89.6%	86.5%
KingSoft Office	*Phase 1*	Open .doc/.xls/.ppt files	14.9%	88.7%	85.5%
	Phase 2		–	–	–
	Phase 3		–	–	–
	Phase 4		70.2%	93.4%	85.4%
MXPlayer	*Phase 1*	Play a video	13.8%	91.0%	88.0%
	Phase 2		–	–	–
	Phase 3		6.6%	98.7%	97.8%
ttpod	*Phase 1*	Play mp3	13.0%	91.1%	88.7%
	Phase 2		11.8%	97.6%	95.1%

Prediction accuracy is reported across workload types. Highly cyclic workloads like video games driven by a regular frame-rendering loop lead to excellent predictions. In contrast, those with little repetition like k9 Mail have few prediction opportunities absent user interactions.

Hierarchical sparse coding performs best when workloads have recurring patterns – among these benchmarks, this includes media applications that are cyclic and driven by regular sampling operations. For Frozen Bubble Phase 2, MXPlayer Phase 3, and ttpod Phase 2, dip prediction is nearly perfect: 94.0%, 98.7%, and 97.6%, respectively. In comparison, an order-8 regression yields sub-60% accuracy for those latter two workloads, indicating that cyclic dips are not extrapolations of prior counter values sampled every $500\mu s$. Though it may be possible to adapt measurement sampling rate to improve regression, sparse-coding signatures need no such adjustment, demonstrating a form of application-driven variation tolerance.

We find that short phases with a high degree of irregular variation lead to few useful long-range signatures. Fitting this description are the k9 Mail benchmark and most applications' Phase 1s, during which the program is initialized. However, for workloads like k9 Mail, user interactions will drive recurrent patterns over the application lifetime, possibly increasing the number of prediction opportunities.

Table 4.2: Prediction Accuracy by Model Configuration

Config.	Layer-1 # Dict Atoms	Layer-2 # Dict Atoms	Acc.	Prec.	Recall
Baseline	100	100	94.4%	58%	38%
Small Layer-1 Dict.	30	–	94.7%	62%	37%
Small Layer-2 Dict.	–	30	94.2%	56%	30%

Shrinking Layer-1 dictionaries improves precision due to more aggressive de-noising within small patches. Large Layer-2 dictionaries are required to capture feature interrelationships, as shown by differences in recall.

Table 4.3: Prediction Accuracy by Layer-1 Training Dataset

Layer-1 Training Workload	Layer-2 Training Workload	Pred Acc.
BBENCH	BBENCH	94.5%
Adobe Reader	BBENCH	93.4%
King Soft (Phase 4)	BBENCH	94.2%
Bootstrapped *(Rand. Sample)* + BBENCH	BBENCH	94.7%

Training Layer-1 dictionaries on data sampled from evenly mixed workloads leads to best performance. This improvement is similar to that found in Chapter 3 across wireless environments, and suggests that Layer-1 features are universal across workloads.

4.5.3 Model Configuration and Prediction Accuracy

In Table 4.2, we examine the effect of changing dictionary size in the different layers of our hierarchical model. When the Layer-1 dictionary is fixed to be small, we see a slight boost in precision due to better de-noising from a more restrictive set of features. In contrast, when we use a small Layer-2 dictionary, recall drops significantly. This indicates that Layer-2 interrelationships are more complicated. Based on the observed utility of a small Layer-1 dictionary, we examine whether those features are universal across workloads. Table 4.3 shows prediction accuracy when the Layer-1 dictionary is trained on data from various workloads, with a 4-window look-ahead and 2-window signatures. We see that, even when the Layer-1 dictionary is trained on out-of-band samples, useful prediction is realized. Similar to the results of Chapter 3, this suggests that latent structure is stable over time and workload and that unsupervised learning captures a universal set of features.

4.5.4 Dynamic Power Savings with Predictive DVFS

The false negative and false positive rates of a predictor impact power savings due to missed opportunities and recovery costs for incorrect scaling decisions. Given these, we model realized dynamic power during instruction throughput lulls when predictive DVFS is applied:

$$\begin{aligned} P_{\text{dyn}} &= Pr(\text{True Pos.}) * (V_{rd})^2 (f_{rd})(a_{rd}) \\ &\quad + Pr(\text{False Neg.}) * (V_o)^2 (f_o)(a_{rd}) \\ &\quad + Pr(\text{False Pos.}) * (V_o)^2 (f_o)(a_o + a_{plt}) \end{aligned} \tag{4.1}$$

This model consists of terms representing power consumption for correct predictions, false negatives, and false positives, respectively. When a lull is correctly predicted, voltage and frequency are reduced from $V_o = 1.0$ to $V_{rd} = 0.25$, and $f_o = 1.0$ to $f_{rd} = 0.25$. For false negatives, voltage and frequency remain at $V_o = 1.0$ and $f_o = 1.0$. For false positives, we assume that higher-than-predicted activity is detected, and that we must execute additional recovery steps.

Dynamic power consumption is proportional to the amount of realized switching activity a [85]. Furthermore, when instruction throughput drops, chips use gating mechanisms to stop electrical activity upstream of unused components. Since these are imperfect and design specific, we use a gating coefficient g to compare different scenarios. Gating efficiencies reported for commercial designs are between $18-37\%$ on the IBM POWER7 [86] and $12-30\%$ for an early PowerPC [87] design, depending

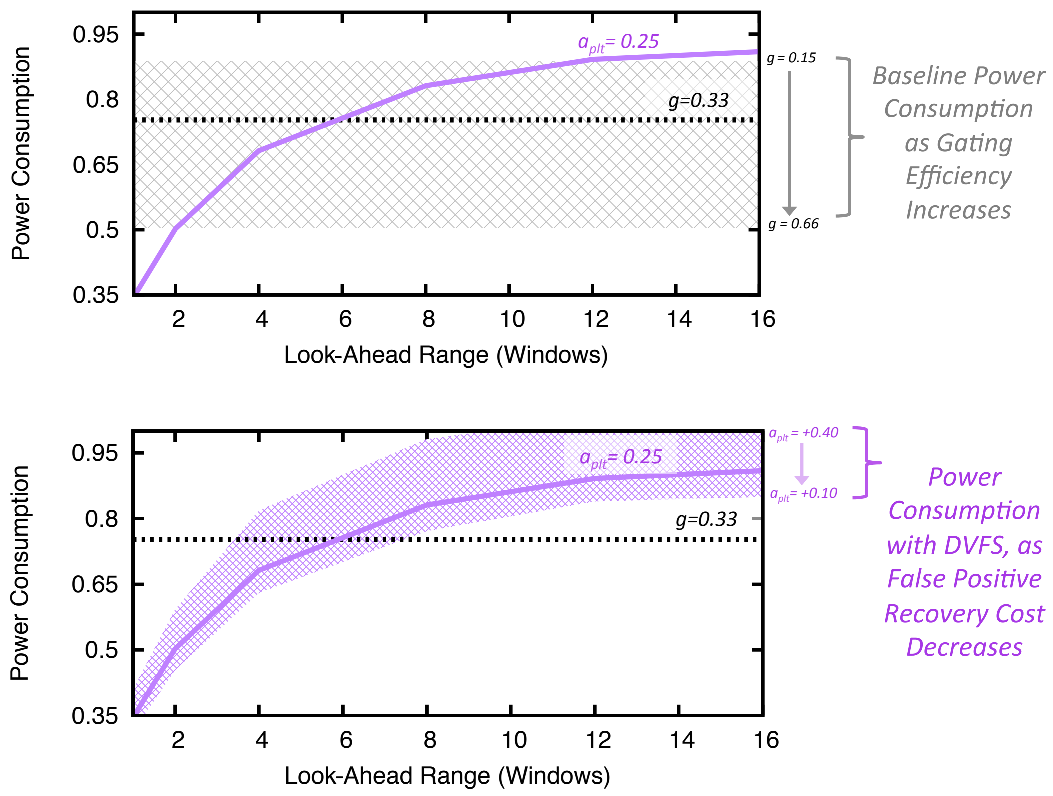


Figure 4.5: Our dynamic power model measures savings during throughput lulls relative to a chip's baseline do-nothing power consumption. At the top, baseline power consumption is parameterized by a chip's gating efficiency – when efficiency is on the order of 15%, on par with commercial processors, predictive DVFS leads to a significant reduction in power consumption. Below, we show the savings of predictive DVFS in light of the cost of false positive recovery, which depends on DVFS hardware specifics. In this figure, a range of power consumption is illustration for DVFS mechanisms with different recovery penalties.

on instruction mix. $a_o = 0.75$ therefore captures ordinary operation, based on the average activity-level during BBENCH. a_{rd} represents reduced activity during dips in instruction throughput, for example $a_{rd} = 1.0 - (0.75) * g$ for a throughput drop of 75%. Finally, a_{plt} represents additional activity needed to compensate for incorrect scaling decisions by rerunning instructions.

Figure 4.5 plots P_{dyn} per lull for our best sparse coding model against look-ahead range. Gating efficiency establishes the do-nothing baseline power consumption during a lull, so we vary $g = 0.15...0.66$ to capture savings for a range of chip designs.

Power savings are also parameterized by the recovery cost a_{plt}, which we vary from +10% to +40% switching activity. When the recovery cost is +25% activity, predictive voltage scaling successfully reduces power consumption with a 4-window heads-up, or 2ms. If DVFS hardware can react fast enough to tolerate a 1ms chip adjustment time, then this savings is a 50% gain over a $g = 0.33$ gating-efficient design without voltage scaling.

4.6 Summary and Conclusion

Circuit-workload modeling and lull prediction for DVFS extends the state-modeling approach of Chapter 3 to a new application domain. In this context, we see that:

- Clustering by sparse coding captures useful predictive signatures of circuit state from hardware performance counters.
- Sequences of circuit states over time produce more-stable predictions than those of a single measurement snapshot. In this case, hierarchy is an efficient means to extend sparse coding to large state-spaces.
- Our state-modeling framework has many flexible configuration options. When both dictionary training *and* SVM training are shifted off-platform, simple inner-product circuitry is all that is needed to make predictions on-platform.

Chapter 5

I/O-Response Modeling for Multi-Tenant Storage

Good storage system performance requires a configuration that is carefully tuned to both higher-level workloads and lower-level device specifics. This fact is evident in a long history of optimization strategies such as queuing algorithms that account for arm repositioning in magnetic disks [88], and datacenter storage systems like CEPH, the Google File System (GFS), and Lustre that alleviate application-specific bottlenecks throughout large distributed platforms [5] [6] [7].

In this chapter, we focus on improving object placement, a critical management task for datacenter storage systems. Specifically, we use sparse coding to reduce overfitting for human-readable CART models that drive placement policies. By discovering latent structure in I/O-access-pattern statistics that differentiate workloads, we improve CART stability when measurements capture unwanted background processes. Furthermore, we show how clusters can be visualized in a format familiar to administrators to reveal new insights about system behavior.

5.1 Object-Based Storage Systems

Object-based storage is a popular paradigm for virtualizing datacenters of heterogeneous commodity devices. As illustrated in Figure 5.1, the object abstraction decouples management tasks like data placement, replication, and load balancing, from block-level storage specifics. Applications interact with a generic interface that accesses data by first obtaining object metadata from a management server, and then using it to request blocks from specific Object-Storage Devices (OSDs). Within the system, management nodes are responsible for global administration, while OSDs use on-board compute resources to oversee individual devices, implementing locally appropriate queuing, caching, etc.

Object-based storage systems have the flexibility to shape overall performance to application workloads. Software management permits runtime resource-scaling, load-balancing, and on-the-fly reconfigurations of distributed devices. For example, the three previously mentioned implementations sculpt system architecture to different access patterns: CEPH uses clever hashing to reduce delays associated with metadata retrieval through decentralization; GFS takes the opposite approach, using a single master node to simplify load balancing and replication management; and Lustre implements granular locks that optimize response time under non-uniform contention patterns.

5.1.1 Object Placement

The assignment of objects to physical OSDs can either improve system performance by load-balancing, or undermine responsiveness when conflicting access patterns create

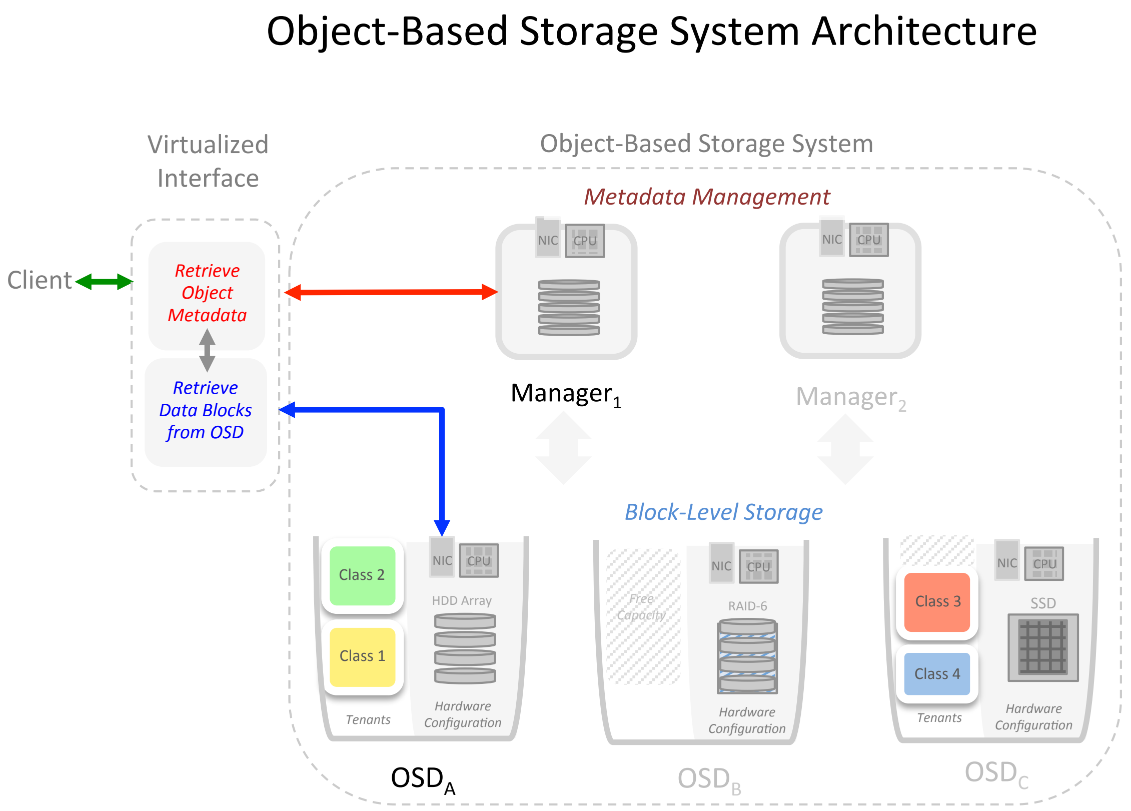


Figure 5.1: Object-based storage decouples block-level devices from metadata managers. Management nodes are responsible for system-wide maintenance like layout and replication. Meanwhile, heterogeneous Object-Storage Devices (OSDs) use computational and network capabilities to make local optimizations for queue management and caching.

bottlenecks at OSDs. Typically, object placement is implemented using rule-based policies set by a system administrator, or according to a predictive performance model that captures interactions between access patterns and device specifics [89] [90]. To simplify placement decisions, objects are often categorized into classes. For example, a manager might group small objects that are accessed randomly to OSDs with large solid state drives (SSDs), and large sequentially-accessed objects to rotating disks [91].

Performance models are most useful when object categorizations are accurate and meaningful. Categories can be determined from application-provided hints [92] or inferred from access pattern measurements. Though hints reflect ground-truth intentions, they are unreliable – programmers ignore even the simplest interfaces, while external designations do not always align with storage-system architecture. In contrast, inferred labels are always available and can be defined with specific optimizations in mind. In addition to standard regressions, CART models are popular in the storage literature. Examples include object classifiers trained on system attributes like filename [93] and class-specific performance predictors [94] [95].

The popularity of CART trees in storage alludes to an important constraint: humans in the loop. Though object placement is an automated process, system administrators are required to answer for downtime, provision for future needs, and actively balance security, economic, and environmental concerns. In our collaborations, black-box or hard-to-interpret learning techniques were consistently rejected in favor of methods that supported human operators.

5.1.2 Multi-Tenant Performance Modeling

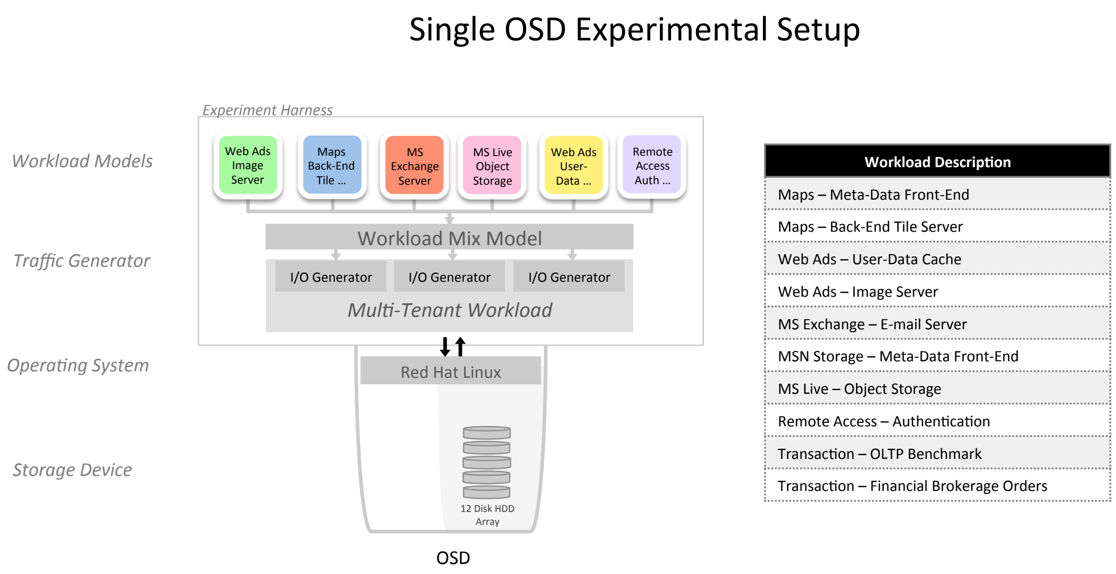
Modeling and prediction methods are well-studied for devices that host objects for a single tenant-workload – however multi-tenant environments like cloud datacenters introduce additional complexity. Consider a workload represented by a distribution of I/O requests W_1. Now suppose that it's host device OSD$_A$ has a capacity of C_A input/output per second (IOPs), and is achieving performance of $IO_A(W_1)$ IOPs. In this scenario, a manager must predict the OSD's future performance if blocks from a new workload W_2 are assigned to that device. A linear performance predictor would check:

$$C_A - IO_A(W_1) > IO_{\text{SLA}}(W_2) \tag{5.1}$$

for a service-level agreement (SLA) guaranteeing $IO_{\text{SLA}}(W_2)$ IOPs to the new workload [96]. Though it is well-known that storage system performance is *not* linear, such approximations are common even in advanced placement algorithms [89].

Non-linear performance prediction in these multi-tenant scenarios is hard for two reasons. First, the sheer number of possible placement combinations makes comprehensive data-driven modeling impossible. Useful models must generalize beyond their training data. Second, access patterns change as clients come and go or hosted applications update their program code. For example, one log provided by a content delivery network showed that changes to data buffers in an external network caused a sudden, sustained rise in access to medium-sized images. Unlike prior modeling research that relied on months or years of data [97] [93], examples like this indicate the importance of transient statistics in datacenter scenarios.

The performance penalty for incorrect predictions is severe. If W_1 represets a web-transaction system with frequent random reads, and W_2 a media application with long sequential reads, head-of-line-blocking will grind performance to a halt. A single long sequential read will cause transaction requests to back up, overloading the request buffer, and potentially causing client-application instability. In practice, problematic configurations are avoided by over-provisioning resources and replicating objects to multiple OSDs [6]. However, one industry report indicates that storage provisioning is inflated by up to 5x, increasing total datacenter costs by as much as 12% [98]. Instead, improved prediction models are a far-more attractive solution to load-balancing.



Workload Description
Maps – Meta-Data Front-End
Maps – Back-End Tile Server
Web Ads – User-Data Cache
Web Ads – Image Server
MS Exchange – E-mail Server
MSN Storage – Meta-Data Front-End
MS Live – Object Storage
Remote Access – Authentication
Transaction – OLTP Benchmark
Transaction – Financial Brokerage Orders

Figure 5.2: Measurement data is generated using `fio` to recreate request traffic reported from a production Microsoft facility. Hosted workloads include everything from a financial transaction server to e-mail servers. Requests are directed at a 12-disk HDD array and steady-state performance is measured after warming the system for 30s.

5.2 Workload Models and Data Collection

We collect training data from a single OSD configured with a 12-disk hard-disk-drive (HDD) array running Red Hat Linux. The `fio` traffic generator recreates request traffic for workloads observed and characterized in a Microsoft production datacenter [99]. Workload descriptions are listed in the right half of Figure 5.2. We probabilistically model parameters for on-disk footprint, block distribution mean/mode, I/O depth, inter-arrival time, and sequential-scan run-length, which are reported for both read and write requests. Parameters are varied by fitting average values to a Gaussian distribution whose variance is a tunable fraction of the mean.

To create a workload instance, we sample parameter values to instantiate `fio`. We fit block-distribution mean and mode parameters to a power-law distribution to capture long-tail effects reported in the literature. As illustrated in the left half of

Figure 5.3: Randomly selected two-tenant configurations exhibit three tranches of IOPs performance when compared to the predictions of an additive linear model. Configurations in green perform close to expectations, indicating that simple bin-packing would be accurate. Configurations shown in orange exhibit slightly sub-linear performance. Finally, configurations in red have disproportionately bad performance, and should be explicitly avoided during object placement.

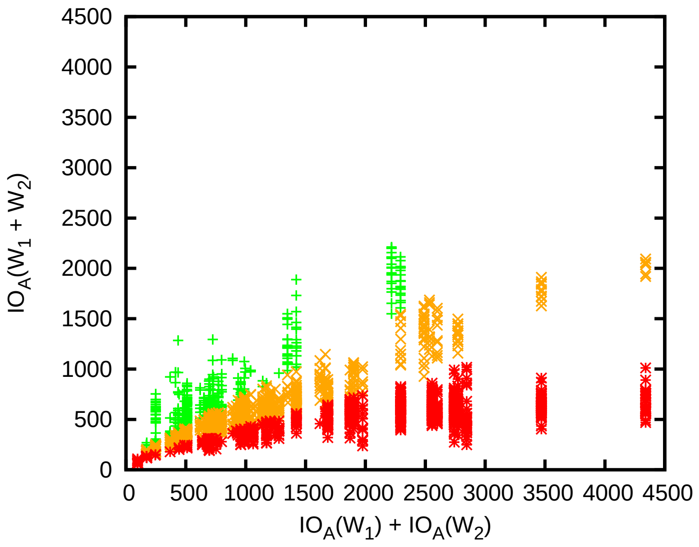


Figure 5.2, each measurement is produced by first selecting tenant workloads and then spawning `fio` instances to generate request traffic. We run `fio` directly on the OSD to factor out network delays, and bring the system to steady state over 30 seconds. We record metrics like IOPs, response latency, bandwidth, etc.

Figure 5.3 shows IOPs measurements for two-tenant configurations. The horizontal axis plots predicted IOPs using a simple additive linear model against measured-performance on the vertical axis. Our experimental data exhibits three distinct tranches based on tenant-workloads: in green are configurations with near-linear performance, in orange are those with slightly sub-linear performance, and in red are those with disproportionately bad IOPs. This last tranche captures high-risk configurations that should be avoided when placing objects.

Table 5.1: Percent of IOPs Predictions Within 40% Accuracy by Representation

	# Background Processes			
CART Input Type	No Interference	2	4	6
Raw Data	77%	63%	57%	55%
OMP with Gold-Dictionary	68%	63%	75%	78%
OMP with Learned Dcitionary	66%	68%	81%	88%

Comparing the proportion of IOPs predictions within 40% of their observed value, we see that CART models trained on raw two-tenant data quickly break down when interfering background processes are introduced. In fact, the low-level contention introduced by interferers stabilizes the storage system, and should ideally make performance *more* predictable. Using OMP to code raw data based on workload templates boosts prediction accuracy significantly across measurement conditions.

5.3 Predicting Storage Performance with CART

5.3.1 Applying CART to Multi-Tenant Scenarios

We now extend CART to model multi-tenant performance. Replicating prior results, we train a model on input vectors x_t that contain aggregate statistics from the device-level counters depicted in Figure 5.5. Measurement values are normalized to lie between 0 and 1, and we use a training set of 400 measurements collected from different randomly-selected tenant configurations. Since CART is a supervised technique, each measurement is associated with a label that represents observed IOPs performance. To mitigate overfitting, we apply bootstrap aggregation ("bagging") and combine the predictions of 5 independently-trained trees. In each, a minimum leaf-count of 5 training samples is enforced. Prediction accuracy is computed on an independent set of 600 measurements of random tenant configurations. Our results are in-line with those previously reported, and achieve 44% prediction error at the 90th percentile cutoff, averaged over all workload models [94].

When we extend this methodology to multi-tenant configurations, performance degrades. Prediction accuracy for two-tenant configurations drops to 52% error at the 90th percentile cutoff. Furthermore, the trained model does not generalize well. For example, when the OSD also includes low-level background processes, prediction quality drops sharply. Background processes are defined as any tenant consuming less than 5% of the OSD's capacity, and are implemented by rate-limited `fio` instances running a randomly-chosen workload. Such processes simulate low-intensity tenants or maintenance tasks that only-slightly impact device performance, but do alter counter values.

Since high-risk-configuration detection is tolerant to some degree of imprecision, Table 5.1 reports the proportion of predictions within 40% error. When background processes interfere with measurements, the portion of useful predictions drops from 77% to 55%. This drop is driven by CART's sensitivity to exact cutoffs during prediction. Ideally, CART's accuracy should *improve* as background processes are added, since low-level contention reduces the likelihood of performance outliers.

5.3.2 Variation-Tolerant CART Using Workload Labels

Since our measurement data is generated according to workload classes reported by researchers, it is easy to define a sparse-coding dictionary of workload templates – single-tenant measurement vectors in training data are simply averaged among workload type. Encoding training samples with this dictionary before fitting a CART model improves prediction significantly. As shown in Figure 5.4, this occurs for two reasons. First, sparse feature representations enable CART to partition data based

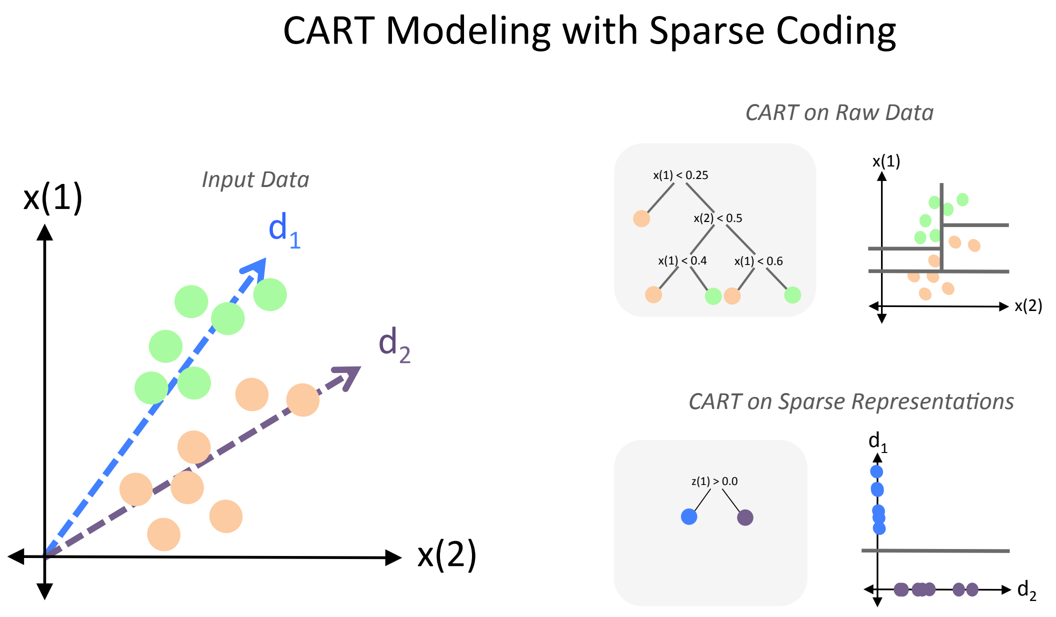


Figure 5.4: CART models greedily partition measurement data along observed dimensions. When trained on sparse feature representations, CART can exploit latent structure to reduce overfitting. The result is improved tolerance under data variations, as well as smaller trees.

on latent structure. This simplifies trees and produces cutoffs that are less sensitive to exact measurement values. Second, as shown throughout previous chapters, imposing sparsity cuts away weakly expressed information such as the interfering effects of background processes.

The strategy of directly-computing a dictionary using ground-truth workload classes is applicable in scenarios with application-provided hints, or when labels can be assigned by-hand to training samples. Features in a "gold" dictionary are defined for each label by: $\underset{g}{d_i} = \frac{1}{|G_i|} \sum_{j \in |G_i|} x_j$, where G_i denotes the set of vectors from workload i, and the underscript g denotes the use of "gold" labels. To compute sparse feature representations, train a CART model, and measure prediction accuracy, we calculate: $\underset{g}{z_t} = \textsc{omp}(x_t, \underset{g}{D}, k = 2)$. The resulting prediction accuracy, reported for the same training and test datasets as Section 5.3.1 is shown in Table 5.1. When

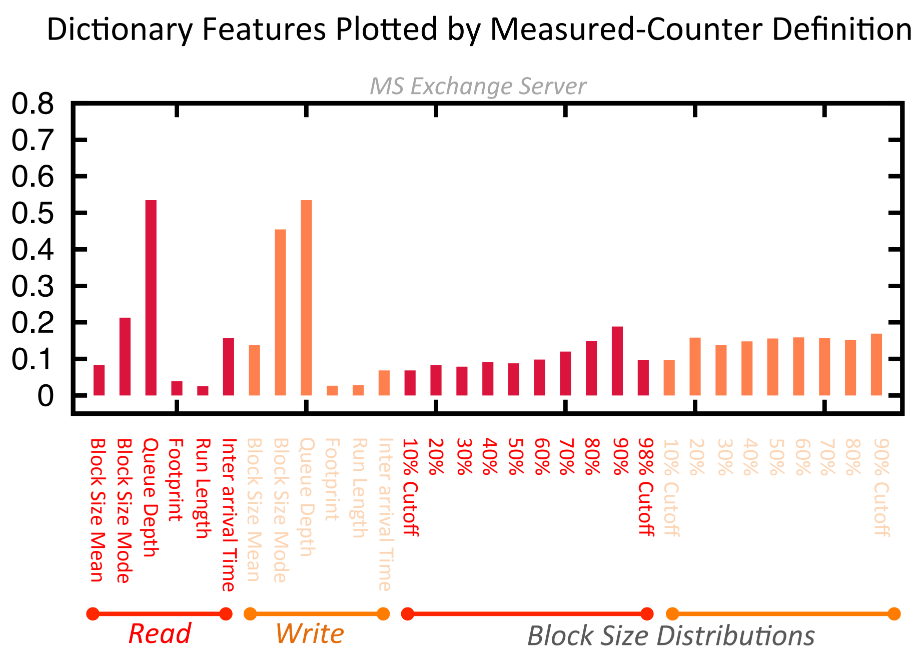


Figure 5.5: Dictionary features can also double as workload profiles. For example, here we visualize the profile of an MS Exchange Server workload and see that both read and write requests are nearly symmetric, though writes have larger block sizes and slower inter arrival time than read requests.

CART is trained on sparse feature representations, we see that predictions exhibit far-greater stability in the face of background-process interference. Our model generalizes well, and is accurate enough that we see the expected stabilizing effect of low-level contention, with prediction accuracy increasing from 63% to 78%.

5.3.3 Learning Workload Filters with Unsupervised Learning

Though sparse workload representations boost CART resilience to interference, Table 5.1 shows that they hurt performance for interference-free data. First, this is because OMP throws away information in a noise-free environment. Second, by comparing vectors z_{t_g} to their corresponding labels, we see that OMP makes mistakes when workload templates are similar. This highlights a shortcoming of hand-assigned labels:

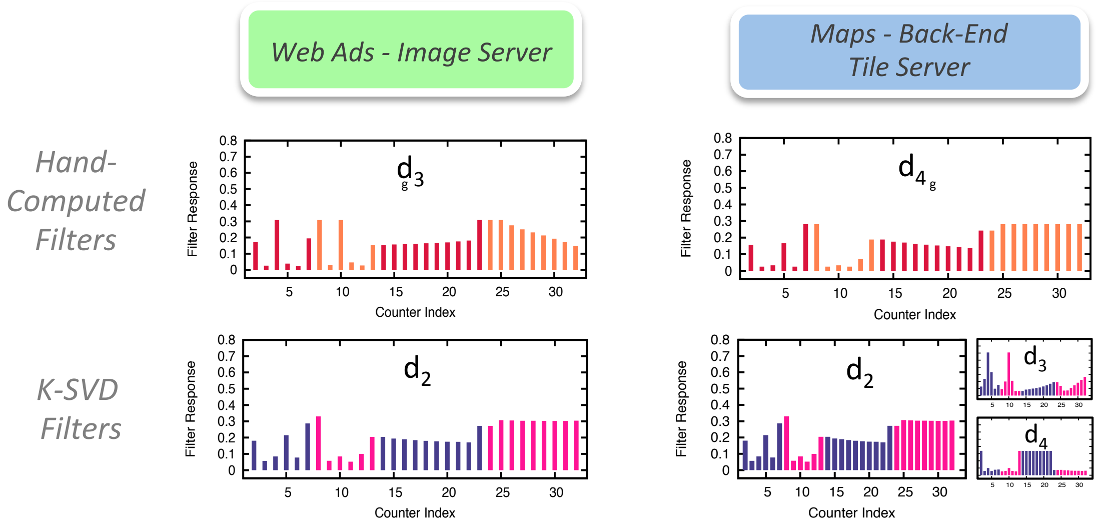


Figure 5.6: Workload templates computed from hand-assigned labels are not guaranteed to be distinctive, as in the case of a Web Ad Image Server and a Maps Back-End Tile Server. The similarity between these two with respect to I/O device counters was discovered during unsupervised learning.

they are not guaranteed to reflect statistical trends.

To address this latter issue, we learn a dictionary using unsupervised clustering and compare its features to the gold-labeled workload templates. Table 5.1 shows that prediction accuracy is highest when CART is trained on learned latent structures, with the portion of useful predictions reaching nearly 90%. Figure 5.6 shows that unsupervised learning finds distinctive workload primitives similar to those captured from wireless link data in Chapter 3. In particular, the similarity of sparse feature representations based on latent structures reveals a previously-unknown similarity between the *Maps - Back-End Tile Server* and the *Ads* workload. Not only does such a relationship improve automated prediction, it provides valuable feedback to system administrators trying to understand how client workloads will affect performance.

5.3.4 High-Risk Configuration Detection

Finally, we characterize detection accuracy when CART is used to identify the conflicting access patterns shown in red in Figure 5.3. Using CART models trained with each of the prior methods, we report accuracy, precision, and recall on this task. Models are trained using sparse representations that capture two-tenant configurations. Test data is then generated by choosing two tenant-workloads, and between zero and six interferer workloads. We make zero- and six-interferer scenarios 50% less likely than two- and four-interferer configurations to reflect the expectation that OSDs will not be interference-free in a tightly provisioned storage system. We set the detection threshold to favor false-positives using an additional +100 IOPs tolerance, since incorrectly flagged configurations can simply be assigned to different OSDs, but false negatives cause bad performance. The results show that CART trained on learned sparse fature representations identifies high-risk configurations with near-80% accuracy, a 7% bump over training CART on raw data.

Table 5.2: Detection Accuracy for Conflicting Access Patterns

CART Input Type	Acc.	Prec.	Rec.
Raw Data	73.1%	0.74	0.73
OMP with Hand-Designed Features	77.3%	0.75	0.73
OMP with K-SVD	**79.5%**	**0.75**	**0.82**

Sparse feature representations significantly improve the detection accuracy of conflicting tenant configurations. When these are based workload templates computed from hand-assigned class labels, accuracy jumps 5%. Refining those labels using unsupervised learning bumps accuracy a further 3%.

5.4 Summary and Conclusion

When applying our state-modeling framework to storage, we demonstrate several new insights:

- Encoding raw measurement data in terms of stable latent structure reduces the sensitivity of CART models to measurement artifacts like interference from low-level background processes.
- Clusters can not-only be directly visualized, they augment interpretable models like CART. When latent structure provides more-separable dimensions for CART, readability improves.
- By discovering latent statistical structure in I/O request statistics, clustering identifies both expected workload-driven patterns, and unexpected similarities between human-assigned workload definitions.
- A state-modeling framework based on sparse coding can exploit additional computational resources like GPGPUs in datacenters. These parallel accelerators support more-stable encoding algorithms like OMP. This tradeoff spends a larger computation budget to improve statistical accuracy.

Chapter 6

Conclusion

6.0.1 MLM: Portable Modeling for Variable Runtime Scenarios

State models are well-studied in wireless, circuit, and storage domains. For example, the earliest wireless burst-error models are over 50 years old [65]. But many of these share a common limitation: they are valid only for the runtime conditions to which they were trained. In this work, we built state models in varying runtime scenarios and exposed large differences in their predictive statistics, as well as useful similarities among latent statistical structures.

One representative example is our UAV wireless link model. Experimentally, the UAV scenario is attractive because links can be repeatably measured and the surrounding environment can be controlled. In the open-space farmland of Stockbridge, NY, the dominant statistical predictors of packet losses are radio range and location. Given time and manpower, it is possible to model this environment accurately. However, UAV link models trained on data like location, range, or exact packet-loss sequences do not port across environments. Quickly, we found that their predictive power was environment-specific and not useful in offices or the subway.

The lesson is that traditional state-modeling techniques, much like the systems we are trying to improve, are not broken, but are simply not flexible enough. When a link model is highly-specialized for a single environment, it will produce uncertain results in new environments. For circuits, workload prediction is similar: designers tune performance models for general-use chips based on benchmark suites like SPEC CPU2000. However, in an early unpublished comparison, we found that the statistics of state prediction are wildly different between these simple, highly-controlled workloads and user-driven applications like web surfing. In storage, performance models tuned in two-tenant conditions performed poorly in less-controlled datacenter scenarios. CART models were sensitive to background processes that perturb request statistics. For each domain, state models from the literature imposed rigid assumptions on target runtime conditions that limit their utility in everyday use-cases.

Machine Learning for Machines (MLM) is therefore *the study of frameworks for modeling performance data that 1) accurately capture state statistics in diverse runtime conditions and 2) meet the computation, power, and data constraints for real-time operation on-platform.* Sparse coding is one such framework. We showed that models based on linear combinations of state primitives were portable across a wide variety of scenarios. Meanwhile, flexible choices of formulation, as well as training, encoding, and prediction algorithms allowed us to trade off accuracy, computational complexity, and interpretability. Still, we recognize that more-effective modeling frameworks may exist, and thus hope that MLM will define an exciting field-of-study to unlock large efficiency and economic gains.

Bibliography

[1] C. Liebert, M. Zayed, J. Tran, J. Lau, and O. Aalami, "Novel Use of Google Glass for Vital Sign Monitoring During Simulated Bedside Procedures," *Abstract - Stanford University School of Medicine* (2014).

[2] S. Eyerman and L. Eeckhout, "Fine-Grained DVFS Using On-Chip Regulators," *ACM Transactions on Architecture and Code Optimization* (2011).

[3] "What Powers Instagram: Hundreds of Instances, Dozens of Technologies," *Instagram Engineering Blog* (2014).

[4] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: Flexible, Scalable Schedulers for Large Compute Clusters," *ACM European Conference on Computer Systems* (2013).

[5] S. a. Weil, S. a. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "CEPH: A Scalable, High-Performance Distributed File System," *USENIX Symposium on Operating Systems Design and Implementation* (2006).

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *ACM SIGOPS Operating Systems Review* (2003).

[7] P. J. Braam *et al.*, "The Lustre Storage Architecture," 2004.

[8] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth, "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Communications of the ACM* (1996).

[9] M. Bohr and K. Mistry, "Intel's Revolutionary 22nm Transistor Technology," *Intel Website* (2011).

[10] R. Fischer, "Intel - How 22nm Yield Changes the Industry," *Seeking Alpha* (2012).

[11] R. E. Fontana, G. M. Decad, and S. Hetzler, "The Impact of Areal Density and Millions of Square Inches (Msi) of Produced Memory on Petabyte Shipments of Tape, Nand Flash, and Hdd Storage Class Memories," *IEEE Symposium on Mass Storage Systems and Technologies* (2013).

[12] P. Darling, "Intel to Invest More Than 5 Billion to Build New Factory in Arizona," *Intel Website* (2011).

[13] S. Borkar, "Design Perspectives on 22nm CMOS and Beyond," *ACM Design Automation Conference* (2009).

[14] I. Wagner and V. Bertacco, "Post-Silicon Verification of Multi-Core Processors," *Post-Silicon and Runtime Verification for Modern Processors, Springer* (2011).

[15] H. Hanson, S. W. Keckler, S. Ghiasi, K. Rajamani, F. Rawson, and J. Rubio, "Thermal Response to DVFS: Analysis with an Intel Pentium M," *ACM International Symposium on Low Power Electronics and Design* (2007).

[16] W. Kim, D. Brooks, and G.-Y. Wei, "A Fully-Integrated 3-Level DC-DC Converter for Nanosecond-Scale DVFS," *IEEE Journal of Solid-State Circuits* (2012).

[17] J. Friedrich, "Keynote: An Introduction to POWER8 Processor," *IEEE International Conference on Integrated Circuit Design and Technology* (2014).

[18] B. Pangrle, "The Good Kind of Regulation," *Semiconductor Engineering* (2014).

[19] M. Harris, "Five Things You Should Know About the nVidia Maxwell GPU Architecture," *nVidia Developer Zone Blog* (2014).

[20] Q. Wu, G. Dong, and T. Zhang, "Exploiting Heat-Accelerated Flash Memory Wear-Out Recovery to Enable Self-Healing SSDs," *USENIX Workshop on Hot Topics in Storage and File Systems* (2011).

[21] L. A. Barroso and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines," *Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers* (2009).

[22] G. Wang and T. E. Ng, "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," *IEEE INFOCOM* (2010).

[23] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing Amazon EC2 Spot Instance Pricing," *ACM Transactions on Economics and Computation* (2013).

[24] R. P. Goldberg, *Architectural Principles for Virtual Computer Systems*, PhD thesis 1973.

[25] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. J. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Systems Journal* (1972).

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM SIGOPS Operating Systems Review* (2003).

[27] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *ACM SIGOPS Operating Systems Review* (2002).

[28] D. Ehringer, "The Dalvik Virtual Machine Architecture," *Techn. report (March 2010)* (2010).

[29] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review* (2008).

[30] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-Based Storage," *IEEE Communications Magazine* (2003).

[31] E. Blossom, "GNU Radio: Tools for Exploring the Radio Frequency Spectrum," *Linux Journal, Belltown Media* (2004).

[32] J. Mitola III, "SDR Architecture Refinement for JTRS," *IEEE Military Communications Conference* (2000).

[33] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *The Journal of Machine Learning Research* (2003).

[34] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, "Deep Learning with COTS HPC Systems," *ACM International Conference on Machine Learning* (2013).

[35] Y. LeCun, K. Kavukcuoglu, and C. Farabet, "Convolutional Networks and Applications in Vision," *IEEE International Symposium on Circuits and Systems* (2010).

[36] M. Elad and M. Aharon, "Image Denoising Via Sparse and Redundant Representations Over Learned Dictionaries," *IEEE Transactions on Image Processing* (2006).

[37] R. G. Baraniuk, "Compressive sensing," *IEEE signal processing magazine* (2007).

[38] N. Pinto, D. D. Cox, and J. J. DiCarlo, "Why Is Real-World Visual Object Recognition Hard?," *Computational Biology, Public Library of Science* (2008).

[39] D. G. Lowe, "Object Recognition from Local Scale-Invariant Features," *IEEE International Conference on Computer vision* (1999).

[40] N. Pinto, Y. Barhomi, D. D. Cox, and J. J. DiCarlo, "Comparing State-Of-The-Art Visual Features on Invariant Object Recognition Tasks," *IEEE Workshop on Applications of Computer Vision* (2011).

[41] J. Hartigan and M. Wong, "Algorithm AS 136: A K-Means Clustering Algorithm," *Journal of the Royal Statistical Society* (1979).

[42] A. Coates, *Demystifying Unsupervised Feature Learning*, PhD thesis Stanford University 2012.

[43] F. Bach, R. Jenatton, J. Mairal, and G. Obozinski, "Optimization with sparsity-inducing penalties," *Foundations and Trends in Machine Learning* (2012).

[44] M. Aharon, M. Elad, and A. M. Bruckstein, "K-SVD and its Non-Negative Variant for Dictionary Design," *International Society for Optics and Photonics, Optics & Photonics* (2005).

[45] R. Tibshirani, "Regression Shrinkage and Selection via the LASSO," *Journal of the Royal Statistical Society.* (1996).

[46] C. J. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery, Springer* (1998).

[47] F. Girosi, "An Equivalence Between Sparse Approximation and Support Vector Machines," *Neural computation* (1998).

[48] M. Riesenhuber and T. Poggio, "Hierarchical Models of Object Recognition in Cortex," *Neuroscience* (1999).

[49] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional Deep Belief Networks for Scalable Unsupervised Learning of Hierarchical Representations," (2009).

[50] M. Steyvers and T. Griffiths, "Probabilistic Topic Models," *Handbook of Latent Semantic Analysis* (2007).

[51] D. Sontag and D. Roy, "Complexity of Inference in Latent Dirichlet Allocation," *Advances in Neural Information Processing Systems* (2011).

[52] N. Morgan, "Deep and Wide: Multiple Layers in Automatic Speech Recognition," *IEEE Transactions on Audio, Speech, and Language Processing* (2012).

[53] S. G. Mallat and Z. Zhang, "Matching Pursuits with Time-Frequency Dictionaries," *IEEE Transactions on Signal Processing* (1993).

[54] Y. C. Pati, R. Rezaiifar, and P. Krishnaprasad, "Orthogonal Matching Pursuit: Recursive Function Approximation with Applications to Wavelet Decomposition," *IEEE Asilomar Conference on Signals, Systems and Computers* (1993).

[55] S. J. Tarsa, T.-H. Lin, and H.-T. Kung, "Performance Gains in Conjugate Gradient Computation with Linearly Connected Gpu Multiprocessors," *USENIX Workshop on Hot Topics in Parallelization* (2012).

[56] P. Ghysels and W. Vanroose, "Hiding Global Synchronization Latency in the Preconditioned Conjugate Gradient algorithm," *Parallel Computing* (2013).

[57] T.-H. Lin, S. J. Tarsa, and H.-T. Kung, "Parallelization Primitives for Dynamic Sparse Computations," *USENIX Workshop on Hot Topics in Parallelism* (2013).

[58] P. Berkhin, "A Survey of Clustering Data Mining Techniques,", 2006.

[59] T. H. McCormick *et al.*, "Bayesian Hierarchical Rule Modeling for Predicting Medical Conditions," *The Annals of Applied Statistics* (2012).

[60] A. Larmo, M. Lindstrom, M. Meyer, G. Pelletier, J. Torsner, and H. Wiemann, "The LTE Link-Layer design," *IEEE Communications Magazine* (2009).

[61] P. Arkachar, "Overview of IEEE 802.11 Standard," (2003).

[62] P. Sarolahti, M. Kojo, and K. Raatikainen, "F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts," *ACM SIGCOMM Computer Communication Review* (2003).

[63] J. B. Andersen, T. S. Rappaport, and S. Yoshida, "Propagation Measurements and Models for Wireless Communications Channels," *IEEE Communications Magazine* (1995).

[64] R. Valenzuela, "A Ray Tracing Approach to Predicting Indoor Wireless Transmission," *IEEE Vehicular Technology Conference* (1993).

[65] E. Gilbert *et al.*, "Capacity of a Burst-Noise Channel," *Bell Systems Technical Journal* (1960).

[66] A. Konrad, B. Y. Zhao, A. D. Joseph, and R. Ludwig, "A Markov-Based Channel Model Algorithm for Wireless Networks," *Wireless Networks, Kluwer Academic Publishers* (2003).

[67] H.-T. Kung, C.-K. Lin, T.-H. Lin, S. J. Tarsa, D. Vlah, D. Hague, M. Muccio, B. Poland, and B. Suter, "A Location-Dependent Runs-And-Gaps Model for Predicting TCP Performance Over a UAV Wireless Channel," *IEEE Military Communications Conference* (2010).

[68] K. Kumar, R. Chandramouli, and K. Subbalakshmi, "On Stochastic Learning in Predictive Wireless ARQ," *Wireless Communications and Mobile Computing, Wiley Online Library* (2008).

[69] B. Ferris, D. Fox, and N. Lawrence, "WiFi-SLAM using Gaussian Process Latent Variable Models," *IEEE International Joint Conference on Artificial Intelligence* (2007).

[70] G. T.-R. W. G. 1, "Proposal of Bit Mapping for Type-III HARQ," *3GPP Meeting No. 18* (2001).

[71] "The Army's Bandwidth Bottleneck," *United States. Congressional Budget Office* (2003).

[72] "Amazon Prime Air - Frequently Asked Questions," *The Amazon Prime Air R&D Team* (2014), Accessed: 2014-09-11.

[73] "Airware - Product Overview," (2014), Accessed: 2014-09-11.

[74] T.-H. Lin and H.-T. Kung, "Robust and Efficient Representation Learning with Nonnegativity Constraints," *International Conference on Machine Learning* (2014).

[75] W. Kim, "Reducing Power Loss, Cost and Complexity of SoC Power Delivery Using Integrated 3-Level Voltage Regulators," *Harvard University* (2013).

[76] K. Rajamani *et al.*, "Application-Aware Power Management," *IEEE International Symposium on Workload Characterization* (2006).

[77] A. Shye, B. Scholbrock, and G. Memik, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," *IEEE/ACM International Symposium on Microacrhitecture* (2009).

[78] G. Contreras and M. Martonosi, "Power Prediction for Intel Xscale® Processors Using Performance Monitoring Unit Events," *IEEE International Symposium on Low Power Electronics and Design* (2005).

[79] K. Singh, M. Bhadauria, and S. McKee, "Real Time Power Estimation and Thread Scheduling Via Performance Counters," *ACM SIGARCH* (2009).

[80] R. Zamani and A. Afsahi, "Adaptive Estimation and Prediction of Power and Performance in High Performance Computing," *Computer Science-Research and Development, Springer* (2010).

[81] A. Dhodapkar and J. Smith, "Comparing Program Phase Detection Techniques," *IEEE/ACM International Symposium on Microarchitecture* (2006).

[82] Binkert and Beckmann, "The Gem5 Simulator," *ACM SIGARCH* (2011).

[83] A. Gutierrez *et al.*, "Full-System Analysis and Characterization of Interactive Smartphone Applications," *IEEE International Symposium on Workload Characterization* (2011).

[84] Y. Huang, Z. Zha, M. Chen, and L. Zhang., "Moby: a Mobile Benchmark Suite for Architectural Simulators," *IEEE International Symposium on Performance Analysis of Systems and Software* (2014).

[85] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a Framework for Architectural-Level Power Analysis and Optimizations," *ACM* (2000).

[86] V. Zyuban *et al.*, "Power Optimization Methodology for the IBM POWER7 Microprocessor," *IBM Journal of Research and Development* (2011).

[87] L. Benini, P. Siegel, and G. De Micheli, "Saving Power By Synthesizing Gated Clocks for Sequential Circuits," *IEEE Design & Test of Computers* (1994).

[88] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," *Winter 1990 USENIX Technical Conference* (1990).

[89] O. Ozmen, K. Salem, J. Schindler, and S. Daniel, "Workload-Aware Storage Layout for Database Systems," *ACM SIGMOD International Conference on Management of Data* (2010).

[90] S. Kraft, G. Casale, D. Krishnamurthy, D. Greer, and P. Kilpatrick, "IO Performance Prediction in Consolidated Virtualized Environments," *ACM SIGSOFT Software Engineering Notes* (2011).

[91] L. L. Ashton *et al.*, "Two Decades of Policy-Based Storage Management for the IBM Mainframe Computer," *IBM Systems Journal* (2003).

[92] M. Mesnier, F. Chen, T. Luo, and J. B. Akers, "Differentiated Storage Services," *ACM Symposium on Operating Systems Principles* (2011).

[93] M. Mesnier, E. Thereska, G. R. Ganger, D. Ellard, and M. Seltzer, "File Classification in Self-* Storage Systems," *International Conference on Autonomic Computing* (2004).

[94] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger, "Storage Device Performance Prediction with CART Models," *IEEE International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems* (2004).

[95] L. Zhang, G. Liu, X. Zhang, S. Jiang, and E. Chen, “Storage Device Performance Prediction with Selective Bagging Classification and Regression Tree,” *Network and Parallel Computing, Springer* (2010).

[96] D. S. Johnson, *Near-Optimal Bin Packing Algorithms*, PhD thesis Massachusetts Institute of Technology 1973.

[97] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, “A Five-Year Study of File-System Metadata,” *ACM Transactions on Storage* (2007).

[98] K. Harty, “Waging War on Overprovisioning,” *Data Center Dynamics* (2013), Accessed: 2014-09-19.

[99] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, “Characterization of Storage Workload Traces from Production Windows Servers,” *IEEE International Symposium on Workload Characterization* (2008).