UNIVERSITY OF CALIFORNIA SAN DIEGO

Efficient Learning in Heterogeneous Internet of Things Ecosystems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Yeseong Kim

Committee in charge:

Professor Tajana Simunic Rosing, Chair
Professor Chung-Kuan Cheng
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Dean Tullsen

2020

The dissertation of Yeseong Kim is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____

Chair

University of California San Diego

2020

DEDICATION

To my wife, Jeeyoon,

my dauther, Olivia Daon,

and my family, Youngju, Haeja, Arie, and Evie.

EPIGRAPH

O give thanks unto the Lord; for he is good:

because his mercy endureth for ever.

— Psalm 118:1

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to first thank my advisor, Prof. Tajana Rosing for her encouragement, support and guidance during my Ph.D. I am extremely grateful for her understanding in all aspects of my life, both research and life. I would like to also thank my committee members, Prof. Dean Tullsen, Prof. Ryan Kastner, Prof. CK Cheng, and Prof. Farinaz Koushanfar for their feedback and discussions related to this Ph.D. work. A special thanks to Prof. Jihong Kim for his guidance and encouraging me to pursue a Ph.D. I learned many skills that helped me perform well in my research under his guidance.

I would like to thank all my lab colleges in SEELab for their active collaboration, help, and all the good memories. I would also like to give special thanks to Pietro Mercati, Mohsen Imani, and Saransh Gupta. I want to sincerely thank all support from Intel, my previous manager, Michael Kishinevski, and my supervisors, Ankit More, Emily Shriver, and Mahesh Ketkar. My research was made possible by funding from National Science Foundation (NSF) Grant 1527034, 1619261, 1730158, 1826967, 1911095, Intel Corporation, CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and SRC-Global Research Collaboration grant.

Most importantly, I owe so much to my wife, Jeeyoon, my daughter, Olivia Daon, my parents and all my family. I would like to thank them for their endless love and support every day. I could overcome all the difficulties thanks to their understanding, patience, and love.

Chapters 2 contains material from “P4: Phase-Based Power/Performance Prediction of Heterogeneous Systems via Neural Networks”, by Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana S. Rosing, which appears in International Conference on Computer-Aided Design, November 2017. The dissertation author was the primary investigator and author of this paper.

Chapters 3 contains material from “Efficient Human Activity Recognition Using Hyperdimensional Computing”, by Yeseong Kim, Mohsen Imani, and Tajana S. Rosing, which appears in IEEE Conference on Internet of Things, October 2018. The dissertation author was the primary

investigator and author of this paper.

Chapters 4 contains material from “A Framework for Collaborative Learning in Secure High-Dimensional Space”, by Mohsen Imani, Yeseong Kim, Sadegh Riazi, John Merssely, Patrick Liu, Farinaz Koushanfar and Tajana S. Rosing, which appears in IEEE Cloud Computing, July 2019. The dissertation author was the primary investigator and author of this paper.

Chapters 5 contains material from “GenieHD: Efficient DNA Pattern Matching Accelerator Using Hyperdimensional Computing”, by Yeseong Kim, Mohsen Imani, Niema Moshiri and Tajana Rosing, which appears in IEEE/ACM Design Automation and Test in Europe Conference, March 2020. The dissertation author was the primary investigator and author of this paper.

VITA

2011	B. S. in Computer Science and Engineering *Cum Laude*, Seoul National University, Korea
2020	Ph. D. in Computer Science (Computer Engineering), University of California San Diego, US

PUBLICATIONS

Yeseong Kim, Mohsen Imani, Niema Moshiri and Tajana Rosing, "GenieHD: Efficient DNA Pattern Matching Accelerator Using Hyperdimensional Computing", *IEEE/ACM Design Automation and Test in Europe Conference (DATE)*, Mar 2020

Mohsen Imani, Mohammad Samragh, Yeseong Kim, Saransh Gupta, Farinaz Koushanfar, Tajana Rosing, "Deep Learning Acceleration with Neuron-to-Memory Transformation", *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, Feb 2020

Mohsen Imani, Yeseong Kim, Sadegh Riazi, John Merssely, Patrick Liu, Farinaz Koushanfar and Tajana S. Rosing, "A Framework for Collaborative Learning in Secure High-Dimensional Space", *IEEE Cloud Computing (CLOUD)*, Jul 2019 (M. Imani and Y. Kim contributed equally)

Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana S. Rosing, "FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision", *International Symposium on Computer Architecture (ISCA)*, Jun 2019

Joonseop Sim, Saransh Gupta, Mohsen Imani, Yeseong Kim, and Tajana S. Rosing, "UPIM : Unipolar Switching Logic for High Density Processing-in-Memory Applications", *ACM Great lakes symposium on VLSI (GLSVLSI)*, May 2019

Anthony Thomas, Yunhui Guo, Yeseong Kim, Baris Aksanli, Arun Kumar, and Tajana S. Rosing, "Hierarchical and Distributed Machine Learning Inference Beyond the Edge", *IEEE International Conference on Networking, Sensing and Control (ICNSC)*, May 2019

Dongwon Park, Ilgweon Kang, Yeseong Kim, Sicun Gao, Bill Lin, and Chung-Kuan Cheng, "ROAD : Routability Analysis and Diagnosis Framework Based on SAT Techniques", *International Symposium on Physical Design (ISPD)*, Apr 2019

Joonseop Sim, Minsu Kim, Yeseong Kim, Saransh Gupta, Behnam Khaleghi, Tajana Rosing, "MAPIM: Mat Parallelism for High Performance Processing in Non-volatile Memory Architecture", *International Symposium on Quality Electronic Design (ISQED)*, Mar 2019

Yeseong Kim, Ankit More, Emily Shriver, and Tajana S. Rosing, "Application Performance Prediction and Optimization Under Cache Allocation Technology", *IEEE/ACM Design Automation and Test in Europe Conference (DATE)*, Mar 2019

Yeseong Kim, Mohsen Imani, and Tajana S. Rosing, "Image Recognition Accelerator Design Using In-Memory Processing", *IEEE MICRO, IEEE Computer Society*, Jan/Feb 2019

Mohsen Imani, Yeseong Kim, Thomas Worley, Saransh Gupta, and Tajana S. Rosing, "HDCluster: An Accurate Clustering Using Brain-Inspired High-Dimensional Computing", *IEEE/ACM Design Automation and Test in Europe Conference (DATE)*, Mar 2019

Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana S. Rosing, "GRAM: Graph Processing in a ReRAM-based Computational Memory", *IEEE Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan 2019

Yeseong Kim, Mohsen Imani, and Tajana S. Rosing, "Efficient Human Activity Recognition Using Hyperdimensional Computing", *IEEE Conference on Internet of Things (IoT)*, Oct 2018

Joonseop Sim, Mohsen Imani, Woojin Choi, Yeseong Kim, and Tajana S. Rosing, "LUPIS: Latch-up Based Ultra Efficient Processing in-Memory System", *2018 International Symposium on Quality Electronic Design (ISQED)*, March 2018

Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana S. Rosing, "P4: Phase-Based Power/Performance Prediction of Heterogeneous Systems via Neural Networks", *2017 International Conference on Computer-Aided Design (ICCAD)*, November 2017

Yeseong Kim, Mohsen Imani, and Tajana S. Rosing, "ORCHARD: Visual Object Recognition Accelerator Based on Approximate In-Memory Processing", *2017 International Conference on Computer-Aided Design (ICCAD)*, November 2017

Mohsen Imani, Yeseong Kim, and Tajana S. Rosing, "Brain-Inspired Hyperdimensional Computing: An Efficient Classifier for Embedded Devices", *2017 International Conference on Computer-Aided Design (ICCAD)*, November 2017

Mohsen Imani, Yeseong Kim, and Tajana S. Rosing, "NNgine: Ultra-Efficient Nearest Neighbor Accelerator Based on In-Memory Computing", *IEEE International Conference on Rebooting Computing (ICRC)*, November 2017

Joonseop Sim, Mohsen Imani, Yeseong Kim, and Tajana S. Rosing, "Enabling Efficient System Design Using Vertical Nanowire Transistor Current Mode Logic", *25th IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, October 2017

Mohsen Imani, Daniel Peroni, Yeseong Kim, Abbas Rahimi, and Tajana S. Rosing, "Efficient Neural Network Acceleration on GPGPU using Content Addressable Memory", *20th Design Automation and Test in Europe (DATE)*, March 2017

Mohsen Imani, Yeseong Kim, and Tajana S. Rosing, "MPIM: Multi-Purpose In-Memory Processing Using Configurable Resistive Memory", *22nd Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2017

Wanlin Cui, Yeseong Kim, and Tajana S. Rosing, "Cross-Platform Machine Learning Characterization for Task Allocation in IoT Ecosystems", *7th IEEE Annual Computing and Communication Workshop and Conference (CCWC)*, January 2017 (Best Paper)

Mohsen Imani, Yeseong Kim, Abbas Rahimi and Tajana S. Rosing, "Acam: Approximate computing based on adaptive associative memory with online learning", *2016 International Symposium on Low Power Electronics and Design (ISLPED)*, August 2016

Mohsen Imani, Abbas Rahimi, Yeseong Kim and Tajana S. Rosing, "A Low-Power Hybrid Magnetic Cache Architecture Exploiting Narrow-Width Values", *5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, August 2016

Mohsen Imani, Yeseong Kim, Abbas Rahimi and Tajana S. Rosing, "Associative Memory with Online Learning for Approximate Computing", *IEEE/ACM Design Automation Conference (DAC)*, June 2016

Yeseong Kim, Boyeong Jeon, and Jihong Kim, "A Personalized Network Activity-Aware Approach to Reducing Radio Energy Consumption of Smartphones", *IEEE Transaction on Mobile Computing (IEEE TMC)*, March 2016

Shruti Patil, Yeseong Kim, Kunal Korgaonkar, Ibrahim Awwal, and Tajana S. Rosing, "Characterization of User's Behavior Variations for Design of Replayable Mobile Workloads", *7th EAI International Conference on Mobile Computing, Applications and Services (MobiCASE)*, November 2015

Yeseong Kim, Francesco Paterna, Sameer Tilak, and Tajana S. Rosing, "Smartphone Analysis and Optimization based on User Activity Recognition", *2015 International Conference on Computer-Aided Design (ICCAD)*, November 2015

Yeseong Kim, Mohsen Imani, Shruti Patil, and Tajana S. Rosing, "CAUSE: Critical Application Usage-Aware Memory System using Non-volatile Memory for Mobile Devices", *2015 International Conference on Computer-Aided Design (ICCAD)*, November 2015

Wook Song, Yeseong Kim, Hakbong Kim, Jehun Lim, and Jihong Kim, "Personalized Optimization for Andriod Smartphones", *ACM Transaction on Embedded Computing Systems (ACM TECS)*, January 2014

Yeseong Kim, Qingqing Zhang, Nosub Sung, and Jihong Kim, "A Mobile Network Emulation Environment for Repeatable Performance Evaluations of Smartphones", *Journal of Korean Institute of Information Scientists and Engineers : Computer Practices (KIISE)*, December 2013

Yeseong Kim, Qingqing Zhang, Nosub Sung, and Jihong Kim, "A Mobile Network Emulation Environment for Repeatable Smartphone Performance Evaluations", *2012 Korea Computer Congress (KCC)*, June 2013 (Best Presentation Paper)

Yeseong Kim, Wook Song, and Jihong Kim, “A Personalized Network Tail Energy Optimization Technique Based on Smartphone Network Usage Behavior”, *Journal of Korean Institute of Information Scientists and Engineers : Computer Systems and Theory (KIISE)*, December 2012

Yeseong Kim, Wook Song, and Jihong Kim, “A Smartphone Network Energy Optimization Technique Using Personalized Network Usage Behavior”, *2012 Korea Computer Congress (KCC)*, June 2012 (Best Paper)

Yeseong Kim, Jongwook Choi, Sungjin Lee, and Jihong Kim, “A Fast File Search Technique Using Direct Access of Metadata Area”, *2011 Korea Computer Congress (KCC)*, June 2011

ABSTRACT OF THE DISSERTATION

Efficient Learning in Heterogeneous Internet of Things Ecosystems

by

Yeseong Kim

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2020

Professor Tajana Simunic Rosing, Chair

The Internet of Things (IoT) is a growing network of heterogeneous devices, combining various sensing and computing nodes at different scales, which creates a large volume of data. Many IoT applications use machine learning (ML) algorithms to analyze the data. The high computational complexity of ML workloads poses significant computational challenges to IoT computing platforms, which tend to be less-powerful and resource-constrained devices. Transmitting such large volumes of data to the cloud also causes various issues such as scalability, security, and privacy. In this dissertation, we propose efficient solutions to perform ML tasks while decreasing power consumption and improving performance. We first leverage the heterogeneous and interconnected nature of the IoT systems, where IoT applications run on many different

architectures (e.g., X86 server or ARM-based edge device) while communicating with each other. We present a cross-platform power and performance prediction technique for intelligent task allocation. The proposed technique estimates the time-variant energy consumption with only 7% error across completely different architectures, enabling the intelligent task allocation that saves the energy consumption of 16.5% for state-of-the-art ML workloads.

We next show how to further advance the learning procedures towards real-time and online processing by distributing such learning tasks onto the hierarchy of IoT devices. Our solution leverages brain-inspired high-dimensional (HD) computing to derive a new class of learning algorithms that can easily run on IoT devices while providing high accuracy comparable to the state-of-the-arts. We present that the HD-based learning algorithms can cover various real-world problems from conventional classification to other cognitive tasks beyond classical MLs such as DNA pattern matching. We demonstrate that the HD-based learning can enable secure, collaborative learning by efficiently distributing a large volume of learning tasks into heterogeneous computing nodes. We have implemented the proposed learning solution on various platforms while offering superior computing efficiency. For example, our solution achieves w 486$\times$ and 7$\times$ performance improvements for each of the training and inference phases on a low-power ARM processor, as compared to state-of-the-art deep learning.

Chapter 1

Introduction

Interconnected devices in the "Internet of Things" (IoT) are generating an ever increasing amount of data. In the year 2020, it is expected that about 1.7 megabytes of new information are being created every second for each human on the planet [1]. IoT applications collect the large amounts of data from various devices and apply machine learning (ML) algorithms to transform that data into actionable knowledge. As a result, running ML algorithms requires significant computational power and storage, resulting in systems that stream most or all the data to the cloud for analysis. However, transmitting all data to the cloud leads to scalability, security and privacy concerns [2, 3]. A promising solution is to distribute the learning tasks onto the IoT hierarchy, however effective learning in the IoT environements is still an open question.

Figure 1.1 shows an illustration of the computing nodes in typical heterogeneous IoT systems, which has various sensors (on *things*) and intermediate computing devices (on *gateways*) beyond traditional servers (on *cloud*). We highlight the main technical challenges in the IoT systems as follows.

- **Heterogeneity of computing platforms:** The emergence of IoT increases the complexity and heterogeneity of computing systems. In the IoT systems, applications, including ML workloads, can run potentially on any device – from the resource-constrained sensory

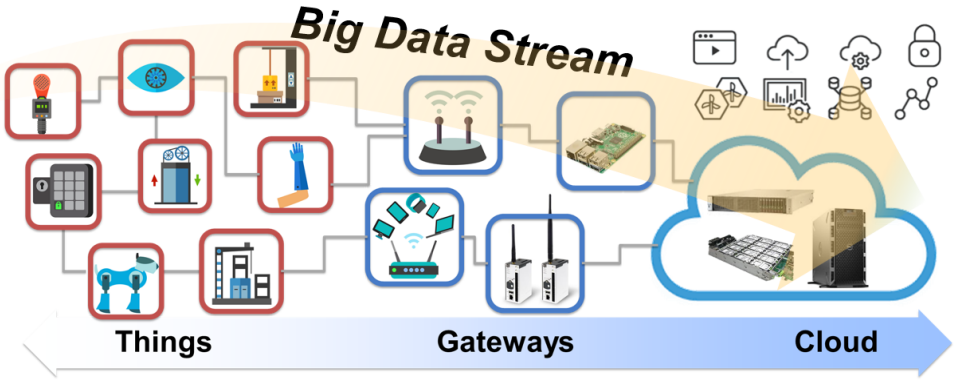


Figure 1.1: Computing Nodes on Heterogeneous IoT Systems

devices to powerful servers, which normally have different architectures. There is already a high degree of heterogeneity even in the datacenter servers, e.g., from legacy to brand-new architectures. The state-of-the-art solutions only estimate average power and performance, not their instantaneous behavior [4]. This makes workload balancing and task allocation difficult.

- **Complexity of ML algorithms:** Training ML models requires a large cluster of application-specific integrated chips (ASIC), e.g., deep learning on Google TPU [5], or is very slow on traditional systems. Distributing the training to devices with heterogeneous data is very difficult due to large communication and computational overhead.

- **Limited computing power and resource:** IoT edge devices often do not have sufficient resources to support heavy workloads of start-of-the-art ML in real-time. For example, many IoT devices use low-power processors such as ARM cores and Intel ATOM architectures, which have limited computing speed and capability. They also often use batteries as their primary energy source. There is a pressing need for alternative learning paradigms which can be run directly on IoT devices for both training and inference, thus enabling real-time and adaptive learning.

- **Variety of learning data:** IoT applications deal with varying types of data generated by

many different sensors. One example is bioinformatics workloads that handle a large size of nucleotide sequences with high runtime and computation costs. We need a holistic approach that efficiently process the variety of learning data [6].

In this dissertation, we propose efficient learning solutions for IoT systems. Our approach covers architecture, application, and IoT hierarchy levels. At the architecture level, we propose a power/performance prediction technique to enable the task allocation across the heterogeneous IoT platforms. The core technology is a cross-machine power/performance prediction technique. The technique extracts key profiles of target applications running on heterogeneous computing architectures and builds prediction models which accurately estimate time-variant power/performance of workloads across different machines and platforms. To ensure the generality of the technique and sufficient coverage of various ML algorithms, the proposed models were developed and verified with diverse industry-standard benchmark applications including SPEC, PARSEC, SPLASH, NERSC, Intel BigDL, and SparkBench. We demonstrate that the proposed technique can enable intelligent task allocation for ML tasks while improving the energy efficiency and costs by 16% over the state-of-the-art distributed computing framework.

Next, at the application level, we design a new class of learning procedures in order to achieve real-time performance with high energy efficiency on IoT devices. We utilize hyper-dimensional (also called as high-dimensional) computing, in short HD computing [7], which is a computing strategy that more closely models the ultimate efficient learning machine: *the human brain*. HD computing mimics several desirable properties of the human brain, including: robustness to noise and hardware failure and single-pass fast learning. These features make HD computing a promising solution for IoT devices with limited storage, battery, and resources. In this dissertation, We can design light-weight algorithms that learn with data by mapping sensor inputs to high-dimensional vectors. We show that the HD-based learning algorithm can solve diverse classification problems in practice with high quality comparable to the state-of-the-art deep learning models.

At the hierarchy level, we enable an HD computing-based collaborative learning framework which efficiently distribute a large volume of learning tasks into heterogeneous computing nodes. Our evaluation show that we can achieve superior computing efficiency with the proposed HD-based learning. For example, we achieve 486× and 7× speedup as compared to the deep learning for training and inference, respectively, when running on a low-power ARM processor. We also show that HD computing can address other challenging tasks beyond classical ML problems. As an example, we focus on a bioinformatics problem of DNA pattern matching.

In the rest of this chapter, we discuss the contributions and related work of this thesis in more details.

1.1 Cross-Platform Energy Prediction and Task Allocation

The emergence of Internet of Things increases the heterogeneity of computing platforms. Migrating workload between various platforms is one way to improve both energy efficiency and performance. Recent cloud systems not only utilize x86-class processors but also employs a large array of low-power ARM processors [8]. With the emergence of edge computing, task allocation decisions of learning procedures also need to be made across different scales of devices, e.g., high-performance servers vs. mobile devices [9].

Effective task allocation and migration requires accurate estimates of its costs and benefits. To date, these estimates rely on analyzing power and performance relationship to system events by domain experts and computer architects. Most previous research work focused on estimation problems of power and performance only for individual machines [10, 11, 12]. Besides, predicting the costs across different architectures has not been accurate for time-series prediction and requires application source code for instrumentation [4].

To enable the intelligent task allocation, we propose P^4, a new Phase-based Power and Performance Prediction framework which identifies application power and performance

across heterogeneous computing platforms. P^4 utilizes machine learning techniques to automate power and performance modeling procedures for various system components, while identifying key system events such as the best-suitable performance counters. It then extracts machine-independent *application phases* by characterizing computing platforms with a set of benchmarks. Given the application phases, it builds neural network-based models which identify the costs and benefits if an arbitrary program procedure were running on a completely different computing platform without ever having to run it on there. We integrate the models trained in P^4 with state-of-the-art ML tasks running on a distributed computing environments, Apache Spark, to serve diverse optimization goals, e.g., energy demand and energy costs in hierarchical systems.

We evaluate the proposed framework on four commercial heterogeneous platforms, ranging from X86 servers to mobile ARM-based architecture, with 154 industry-standard benchmarks. Our experimental results show that P^4 can predict the power, performance and energy changes with only 6.8%, 5.6%, and 6.1% error, respectively, even for completely different architectures from the ones applications ran on. The model-based framework effectively allocates ML tasks and achieves energy saving of 16.5% and energy cost reduction of 16.8%. This is presented in Chapter 2.

1.2 Efficient Learning Based on HD Computing

A key task of many IoT applications is to understand underlying context and react to the environment based on sensed data. Machine learning is widely used for this. However, they are often overcomplex to run on less-powerful IoT devices although it is appropriately allocated as we discuss in Chapter 2. A key focus of the dissertation is to develop an alternative approach which efficiently supports the learning tasks using brain-inspired HD computing. HD computing is based on a short-term human memory model, Sparse distributed memory, emerged from theoretical neuroscience [13]. It leverages the understanding that the human brain operates

on *high dimensional* representations of data originated from the large size of brain circuits [14]. It thereby models the human memory using points of a high-dimensional space, that is, with *hypervectors*. The hyperspace typically refers to tens of thousand dimensions. HD computing incorporates important functionalities of the human memory model with vector operations which are useful to design a new class of efficient learning solutions. HD computing-based learning is also robust to noises, computationally tractable, and mathematically rigorous in describing human cognition.

We show how the HD computing can be applied to learning problems in IoT systems while improving the accuracy and efficiency. To verify the idea for practical IoT applications, we evaluate the developed learning algorithm using three human activity recognition datasets collected from the sensor data of mobile/embedded devices. We present that the proposed design achieves the speedup of the model training and inference by up to 486x and 7x as compared to the state-of-the-art neural network training [15]. This is presented in Chapter 3.

1.3 Collaborative Learning with HD Computing

As the amount of data generated by the Internet of the Things (IoT) devices keeps increasing, many applications offload computation to the cloud. However, this often entails risks due to security and privacy concerns. Encryption and decryption methods have been proposed and used in practice, e.g., Homomorphic Encryption [16], but they add an already significant computational burden. We utilize the HD-based learning algorithm presented in Chapter 3 to build a collaborative and secure learning solution. In Chapter 4, we present SecureHD, which encodes original data into secure, high-dimensional vectors, while the training is performed with the encoded vectors. Thus, applications can send their data to the cloud with no security concerns, while the cloud can classify the data without additional decryption. We also show how SecureHD can recover the encoded data in a lossless manner.

In our evaluation, our proposed SecureHD framework can perform the encoding and decoding tasks 145.6× and 6.8× faster than a state-of-the-art encryption/decryption library running on the contemporary CPU [17]. In addition, our learning method achieves high accuracy of 95% on average for diverse practical classification tasks including cloud-scale datasets.

1.4 Beyond Classical Learning: DNA Pattern Matching using HD Computing

IoT applications handle varying data types collected from diverse sensors and devices. Although many of them can be collected as the feature vectors which most classical ML algorithms consider, there are many other IoT data which cannot be easily mapped [6]. We examine the potential of HD computing for the wider range of learning tasks by focusing on a key procedure of bioinformatics problem – DNA pattern matching. Acceleration of bioinformatics applications is a key procedure to enable personalized IoT-based healthcare [18] and on-site disease detection [19]. Although previous research proposed various accelerator designs on GPU [20] and FPGA [21], the increasing volume of the DNA data exacerbates the runtime and power consumption of the DNA pattern matching.

We present GenieHD in Chapter 5, which efficiently parallelizes the DNA pattern matching task using the idea of HD computing. We transform inherent sequential processes of the DNA pattern matching to highly-parallelizable computation tasks. The proposed technique first encodes the whole genome sequence and target DNA pattern into high-dimensional vectors. Once encoded, a light-weight operation on the high-dimensional vectors can identify if the target pattern exists in the whole sequence. We also design an accelerator which effectively parallelizes the HD-based DNA pattern matching while significantly reducing the number of memory accesses. The architecture can be implemented on various parallel computing platforms to meet target system requirements, e.g., FPGA or ASIC. We evaluate GenieHD on practical large-size DNA datasets

such as human and Escherichia Coli genomes. Our evaluation shows that GenieHD significantly accelerates the DNA matching procedure, e.g., 44.4× speedup and 54.1× higher energy efficiency as compared to a state-of-the-art FPGA-based design [21].

Chapter 2

Intelligent Cross-Platform Task Characterization and Allocation

2.1 Introduction

With the emergence of the IoT, application tasks can run on many *heterogeneous* computing nodes (e.g., X86 Xeon server vs. ARM-based mobile device) and at varying operating conditions (e.g., CPU frequency and sleep states) [22]. In these computing ecosystems consisting of heterogeneous devices, dynamic management and task mapping of learning applications to meet diverse objectives cannot be done without accurate estimates of the performance/power costs and benefits [23].

Prior work has been conducted to build power and performance models targeting a single machine [24, 11, 12]. A basic assumption of the existing modeling techniques is that the power level is proportional to the workload intensity, such as the amount of computation and memory accesses. Despite much research over the last decades, predicting power consumption across multiple heterogeneous machines still remains a difficult problem since application behavior significantly varies as a function of CPU architecture, platform design, and runtime conditions. For example, given a C program, architecture differences, e.g., x86 (CISC) and ARM (RISC), generate incompatible instructions which require significantly different cycles and hardware usage. In our experiments, power consumption to run the same application also varies more than 100**x** between the two CPU architectures.

In this chapter, we propose a novel characterization framework called P^4 (Phase-based Power and Performance Prediction) for the intelligent task allocation of ML tasks. P^4 identifies (i) machine-independent application behavior at a fine granularity and (ii) cross-platform models that characterize power and performance relationships. We utilize *application phases* to characterize the fine-grained system behavior on different platforms. The application phase is defined as an execution period which homogeneous system usage behavior is observed.

P^4 first identifies the same application phases across different platforms using a set of benchmark applications offline, and trains neural networks models which capture per-phase

power, performance and energy characteristics across machines. The characterization procedures are fully automated with machine learning process. It allows performing the characterization procedures for various computing platforms without significant system-to-system tuning.

We deploy the cross-platform modeling technique in a practical distributed ML computing framework, running on Apache Spark [25]. With all components implemented in a single place, we can optimize energy usage and energy costs of a hierarchy of computing nodes running ML tasks.

To summarize, we make following contributions:

- **P^4 automatically creates power and performance models of various hardware (HW) components including processor, memory, fan, disk, and networking devices.** During the model training, P^4 automatically identifies key system events, e.g., Performance Monitoring Counters (PMC), based on a feature selection method [26]. It eliminates the manual event selection procedure that the earlier work have relied on knowledge of domain experts [10, 24, 11, 12, 27].

- **P^4 recognizes the cross-platform application phases using the key system events as the machine-independent workload profiles.** The recognition tasks is performed in a non-intrusive way, unlike earlier work [28, 4, 29, 30] that depends on either source code or binary instrumentation.

- **With the application phases, we present how to train neural network models that generalize the cross-platform workload characteristics for each phase.** The neural network models accurately predict how much power, performance, and energy that an application would have if it were executed on a platform that is completely different from the one it is currently running on.

- **We integrate the cross-platform models trained by P^4 with Apache Spark framework to enable predictive task allocation and energy optimization of state-of-the-art ML**

procedures. At runtime, the framework performs a model-based decision where to allocate a Spark task for optimized energy usage and costs, even when the task has not been seen during the offline characterization.

We evaluate the proposed technique on four completely different platforms/architectures (e.g., x86 Xeon E5440 vs. x86 Westmere vs. ARM Cortex A15) and scales (e.g., servers vs. mobiles) with 154 industry-standard benchmarks. The experimental results show that P^4 can provide accurate power estimates of HW components and total systems only with less than 7.5% error, while automatically identifying the application phases across computing machines. P^4 also predicts time-variant power, performance, and energy consumption with only 6.8%, 5.6% and 6.1% error across different machines including cross-architecture cases. The predictive task allocation technique integrated with Spark achieves energy saving of 16.5% and energy cost reduction of 16.8%.

2.2 Related Work

System power modeling: Most power models in literature assume linear relationship between power and system events [10, 27, 31, 32, 33, 12] as also summarized in the following surveys [11, 12]. A number of publications have explored estimates of power consumption when a machine changes C and P power states, rarely change for different power states in a single machine, and developed a linear regression model to estimate power and performance. Similar techniques have been proposed for frequency changes [34] and for cores in heterogeneous multicore systems [35]. Due to increasing complexity and heterogeneity of architectures [36], the cross-platform workload behavior cannot be estimated by only relying on the assumption of the linearity between the PMC events to the power consumption, and thus it requires a more sophisticated approach to get the desired level of accuracy.

Application phase analysis: A promising way to better understand the workload be-

havior across architectures is to exploit fine-grained application phases. Different sections of a program execution show distinct power characteristics [29, 37]. The phase detection technique presented in [38] identified the application status by tracking function calls of Java mobile applications to design a phase-driven power management which applies different CPU frequencies to improve the energy efficiency. Work in [39] inferred the phases from system events of mobile device subcomponents, such as CPU and GPS to detect abnormal power consumption. Another work modeled the phases of HPC applications based on MPI-specific APIs to automatically generate parallel benchmarks [40]. Based on their phase identification algorithm, they developed phase-aware power management mechanism for multicore systems. Work in [41] proposed a thread scheduling technique which finds stable phases based on performance counters and migrates threads when a stable phase is finished to reduce long memory latencies. Work in [28, 4] recently showed that the phase information is useful to predict cross-platform power levels. They proposed a technique which identifies the phases during compile time at the level of basic blocks.

Energy management for distributed systems: The management techniques for distributed systems have been also developed for diverse optimization purposes, e.g., peak power management for data centers [42], resource scheduling based on application classification [43], and management with renewable energy source [44]. Many of these techniques utilizes energy estimates as the key variable of the management techniques, where the estimates are assumed to be available from different sources [45], e.g., CPU speed-based approximation in virtual machines [46], Hadoop/Spark logs [47] and application-specific analysis [48].

In this chapter, unlike the previous work that rely on a priori knowledge of applications such as the previous system traces and offline analysis for program basic blocks, we focus on how to accurately estimate fine-grained power and energy across machines in an automated, non-instrusive way for arbitrary applications. Our technique accomplishes this goal by recognizing cross-platform application phases based on key system events which are available on the existing system architectures. We also show how the fine-grained prediction can be used to improve the

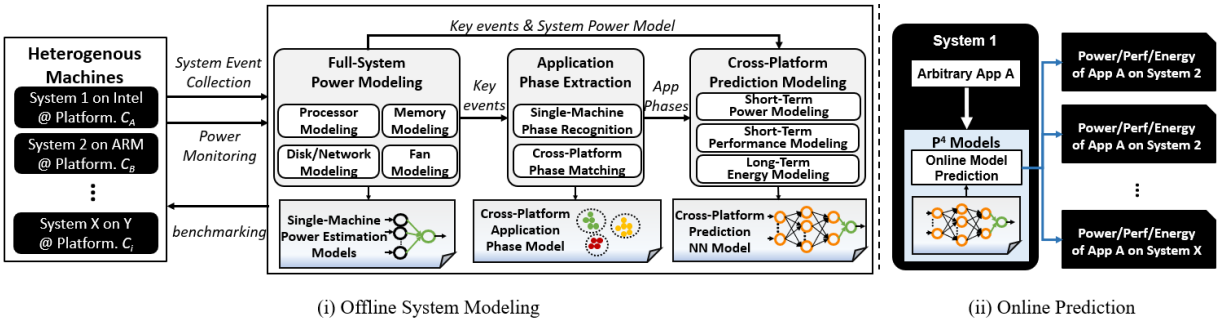


Figure 2.1: An overview of P^4 framework

energy efficiency for a practical distributed computing environment.

2.3 Overview of P^4

Figure 2.1 shows an overview of our proposed P^4 framework. The framework characterizes power/performance tradeoffs of each machine of interest, and develops models to estimate power consumption of the machine and application phases observed in the benchmarks. It then formulates the model for prediction of power consumption and performance across different HW configurations. The offline characterization is done by running a set of benchmarks while collecting PMC data and measuring the power consumption on the multiple HW platforms of interest. Table 2.1 summarizes the models automatically built in P^4 during the offline stage.

The first characterization step is *full-system power modeling* (Section 2.4.1) which takes power measurements of HW components and various system events as inputs. P^4 automatically selects strongly power-related events among all collected events and builds a single-machine power estimation model for each platform. The selected events are used as key parameters for the further learning stages, i.e., *application phase extraction* and *cross-platform prediction modeling*. The application phase extraction module identifies application phases from the selected events by utilizing an automated unsupervised clustering procedure (Section 2.4.2). The goal of the cross-platform prediction is to train power/performance prediction models that are later used online.

Table 2.1: Comprehensive system models built on P^4

Single-Machine Power Model			
Subcomponent	**Key Metrics**	**Machine Learning for Automation**	**Model Notation**
Processor	Performance monitoring counters ($\mathbf{e}_{pmc}$)	Lasso with linear regression estimator	$P_{processor}$
Memory	Memory bandwidth ($\mathbf{e}_{bandwidth}$)	Second-order polynomial regression	P_{mem}
Disk	Number of accessed sectors ($\mathbf{e}_{diskIO}$)	Isotonic regression	P_{disk}
Network	Number of packets ($\mathbf{e}_{networkIO}$)	Isotonic regression	$P_{network}$
Fan (only for x86 servers)	Fan speed ($\mathbf{e}_{RPM}$)	Third-order polynomial regression	P_{fan}
Total System	All subcomponet events	N/A	P_{system}
Cross-Platform Prediction Model			
Types	**Key Metrics**	**Machine Learning for Automation**	**Model Notation**
Phase	All subcomponent events	k-Means++	C_{phase}
Performance	All subcomponent events	Neural networks	NN_{perf}
Power	All subcomponent events	Neural networks	NN_{power}
Energy	All subcomponent events	Neural networks	NN_{energy}

P^4 utilizes the phases as a basic unit to understand cross-platform relationships of application tasks. Since the application phases are identified for a single machine in the first step, we further identify the same phases across platforms through a phase matching procedure. Then, we can relate per-phase event behavior between platforms, and generalize the relationship by training neural network (NN) models (Section 2.5), which are designed to predict cross-platform power and performance behaviors. Combining the neural network models with the power estimation model, we can predict cross-machine power, performance, and energy consumption of arbitrary programs at runtime. In Section 2.6, we show how to integrate the models trained by P^4 with a distributed computing Apache Spark environment for online management.

2.4 Automated System Modeling

2.4.1 Full-System Power Modeling

The proposed framework builds the estimation models for system components such as processor, memory, disk and fans. In this section, we discuss the challenge of the modeling procedure and describe which machine learning technique is suitable to automatically and accurately capture the power consumption for each component.

CPU: Many prior work have used a linear regression model for the CPU power modeling based on the observation that there is a linear relationship between the processor power consumption and a set of performance counters [49, 34]. In the past, system engineers would select the right PMCs for the models by leveraging domain knowledge. However, increasing system heterogeneity makes this manual event selection difficult. For example, modern computing systems have more than a hundred PMC events, but only a few can be collected at the same time due to the limited number of hardware counters and monitoring overhead. Thus, a key challenge is how to select the minimum number of the power-related performance counters among all available.

The P^4 framework fully automates the PMC selection procedure by using Lasso statistical analysis (Least Absolute Shrinkage and Selection Operator) [26]. The Lasso method exploits l_1-regularization to perform feature selection while building a regression model. Let $\mathbf{e}_{pmc}^{app_j} = \langle e_{1,t_i}^{app_j}, e_{2,t_i}^{app_j}, \ldots, e_{k,t_i}^{app_j} \rangle$ be a vector for k PMC events e at a time interval i for an application j, called an *event vector*. Then, the collected data for a computing platform $\boldsymbol{C}_A$ can be represented by a set of event vectors, $\mathbb{D}_{C_A} = \{V_{t_0}^{app_0}, V_{t_1}^{app_0}, \ldots, V_{t_L}^{app_N}\}$. For a general event vector $\mathbf{e}_{pmc} = \langle e_{1,t'}, e_{2,t'}, \ldots, e_{k,t'} \rangle$, a linear power model for $\boldsymbol{C}_A$ is represented by

$$P_{processor}(e_{i,t'}) = \sum_{i=1}^{k} \beta_i e_{i,t'} + \beta_0 \tag{2.1}$$

where β_i is the coefficient correspondent to each event and β_0 is the intercept. The linear regression finds the parameters using least square solutions. Unlike standard linear regression, the coefficients of less power-related events are set to zero by Lasso, and thus we can automatically exclude them and build the power model by only using the selected events. Table 2.2 shows the list of PMC events which are selected by the Lasso method in the P^4 framework. For three tested servers Lasso selected 12 PMCs, while for ARM it selected 11. We evaluate the accuracy of selected performance counters on the *event-based power estimation* in Section 2.8.1.

Table 2.2: Selected PMC events for processor power estimation ($\mathbf{e}_{pmc}$)

		Availablity	
Event	**Description**	**x86**	**ARM**
CLOCK_CYCLES	Clock cycles	•	•
INSTRUCTIONS	Instructions	•	•
RS_UOPS_DISPATCHED	Micro operations dispatched	•	
FP_COMP_OPS_EXE	Floating point operations	•	
BR_INSTS	Branch instructions retired	•	•
BR_MISSS	Branch instructions missed	•	•
L1-DCACHE_LOADS	L1 data cache loaded	•	•
L1-DCACHE_STORES	L1 data cache stored	•	•
LLC_REFERENCES	Last level cache referenced	•	•
LLC_MISSES	Last level cache missed	•	•
BUS_CYCLE	Bus cycles	•	•
RESOURCE_STALLS	Resource stalls	•	
DP_SPEC	Speculative integer operations		•
UNALIGNED_LDST_SPEC	Speculative ld/st operations		•

Memory: Capturing detailed accesses of the memory subsystem involves high monitoring overheads. Instead, we capture high-level memory activities by using an event, BUS_TRANS_MEM, in addition to the events selected for the processor. This event counts all microarchitecture-level activities initiated on the memory bus, thus having high correlations to the memory bandwidth utilization [50]. We evaluated different regression techniques with the event as the input, and found that it has non-linear relationship to the power consumption. P^4 in particular exploits the second-order non-linear regression model for the memory bandwidth utilization, $\mathbf{e}_{bandwidth}$, denoting by $P_{mem}(\mathbf{e}_{bandwidth})$.

Disk and Network: The relationship between power and IO devices (e.g., networking and disk) has been commonly described by either stateful models [24] or linear regression models [27]. The stateful model can account the power consumption accurately, however it was not an appropriate solution for our automated modeling process, since the *internal* states of the disk (e.g., idle and multiple active states depending on IO traffics) are usually not exposed to the software level. On the other hand, although the linear regression-based method could be performed without those knowledge, we observe that the model is often underfit when multiple states exist.

To automate the modeling procedure without having the knowledge of the internal states, P^4 exploits Isotonic regression [51]. The Isotonic regression automatically generates a piece-wise regression model without specifying breakpoints. We have implemented an IO monitoring tool to collect the amount of traffics from sysfs interface in the operating system with negligible monitoring overhead. The key metrics for the disk and network components are the number of sectors read or written, $\mathbf{e}_{diskIO}$, and the number of transferred packets, $\mathbf{e}_{networkIO}$, respectively. With the collected data, the P^4 framework identifies where the best breakpoint happens indicating the different states according to the amount of the traffic, and in turn creates the piece-wise linear regression model for each state, denoting by $P_{disk}(\mathbf{e}_{diskIO})$ and $P_{network}(\mathbf{e}_{networkIO})$.

Fan: In server systems, the fan subsystems also contribute a significant portion of the power consumption [52]. The power consumption is highly related to the fan speed, usually represented by revolutions per minute (RPM). This event can be monitored from a side-band processor which controls the fan speed. In our environment, we measure the fan speed using Intelligent Platform Management Interface (IPMI) protocol. We adopt a third-order polynomial regression model, $P_{fan}(\mathbf{e}_{RPM})$, which takes the fan RPM as the input.

Once creating all the power models for each subcomponent, we can combine each model to estimate the total power consumption of a system with the measurement of the supply power, P_{system}, as follows:

$$\begin{aligned} P_{system} &= P_{processor}(\mathbf{e}_{pmc}) + P_{mem}(\mathbf{e}_{bandwidth}) + \\ & \quad P_{disk}(\mathbf{e}_{diskIO}) + P_{network}(\mathbf{e}_{networkIO}) + \\ & \quad P_{fan}(\mathbf{e}_{RPM}) + C \end{aligned}$$

where C is the power consumed by the rest of the systems. In our experiment, since P^4 models all

the major subcomponents whose power have substantive power dynamics in the entire systems, C is estimated as a constant value for each machine.

2.4.2 Cross-Platform Application Phase Recognition

In order to associate the total power estimates with actual application workloads running on the system, we build another model which explains how a program interacts with the systems in a high level. The *application phase extraction* module is responsible to identify *application phases*, i.e., clusters of homogeneous system usage behavior. With this model, P^4 regards an application execution as a sequence of multiple application phases.

To better explain the concept of the phases, Figure 2.2 shows power measurements and two representative PMC events for a *Linpack* benchmark [53] running on Intel SR1560SF server. As shown in the figure, the power consumption and PMC events have similar patterns of changes over time, e.g., (A,C) and (B,D). In addition, similar performance characteristic, e.g., the number of instructions for an interval (`INSTRUCTIONS`), is observed for each labeled period. By extracting the application phases, P^4 identifies that the workloads executed from 10 to 50 seconds are the sequence of the four phases.

P^4 automatically relates the similar event behaviors to different application phases with k-means++ clustering algorithm[1] [54]. The phase extraction procedure uses the set of event vectors $\mathbb{D}_{C_l}$, as the input data set of the k-means algorithm. Then, the algorithm assigns a cluster index $\rho_{i,j}$ to each vector $V_{t_i}^{app_j} \in \mathbb{D}_{C_l}$, where $0 \leq \rho_{i,j} < k$. This algorithm requires two parameters, the number of clusters k, and the initial center of each cluster. We determine the best k using kNeedle algorithm [55] while the initial cluster centers using the k-means++ approach [54].

Based on the application phases characterized for each machine, P^4 identifies cross-platform workload characterization. Figure 2.3 shows an example of the power levels and identified phases where each phase is denoted by different colors. In this experiment, a multi-

[1] We also tested other clustering algorithms such as DBSCAN and hierarchical clustering, and chose the k-Means since it identified the phases for most benchmarks sufficiently compared to the other algorithms.

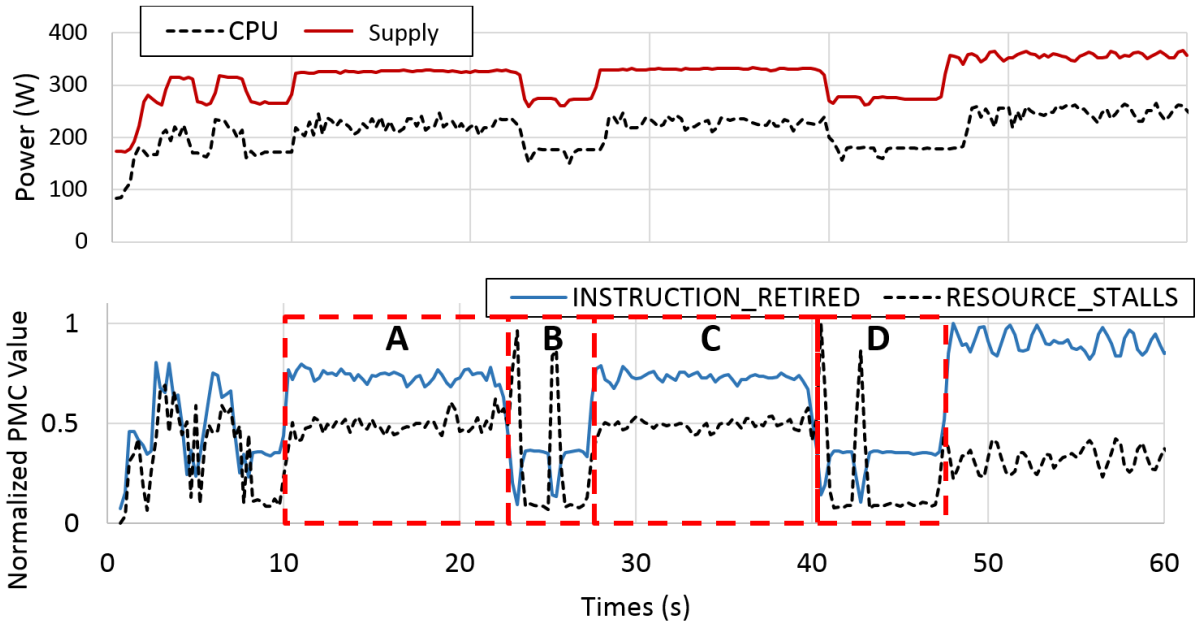


Figure 2.2: Power and performance with PMC events for Linpack benchmark on Intel SR1560SF server at maximum frequency

threaded *bzip2* benchmark was executed on two different servers, Intel SR1560SF and Sun X4270. We observe that, when running the same application, the pattern of the identified phases are very similar across machines. The *bzip2* execution is divided to three periods in a very similar fashion for the two platforms, i.e., A, B, and C for the Intel server and A', B', and C' for the Sun server. It means that the high-level workload characteristics are independent on the running platform – e.g., a compute-intensive phase in a platform is likely to be compute-intensive in another platform as well. Thus, if we can identify which phases are the same across platforms, it is possible to automatically relate the workloads of a *black-box* application without having the source code for instrumentation.

The proposed framework identifies the relationship by associating the application phases extracted on a single machine with *machine-independent* phases on other platforms. This is done with a modified k-means algorithm which tasks the application phases identified while running on a single machine. Figure 2.4 illustrates the cross-platform phase matching procedure. Let $\mathbb{V}_{C_A}^{app_p} = \{\upsilon_{\rho_{1,p}}, \ldots, \upsilon_{\rho_{k,p}}\}$ be a set of cluster centers for the phases obtained for app_p on a

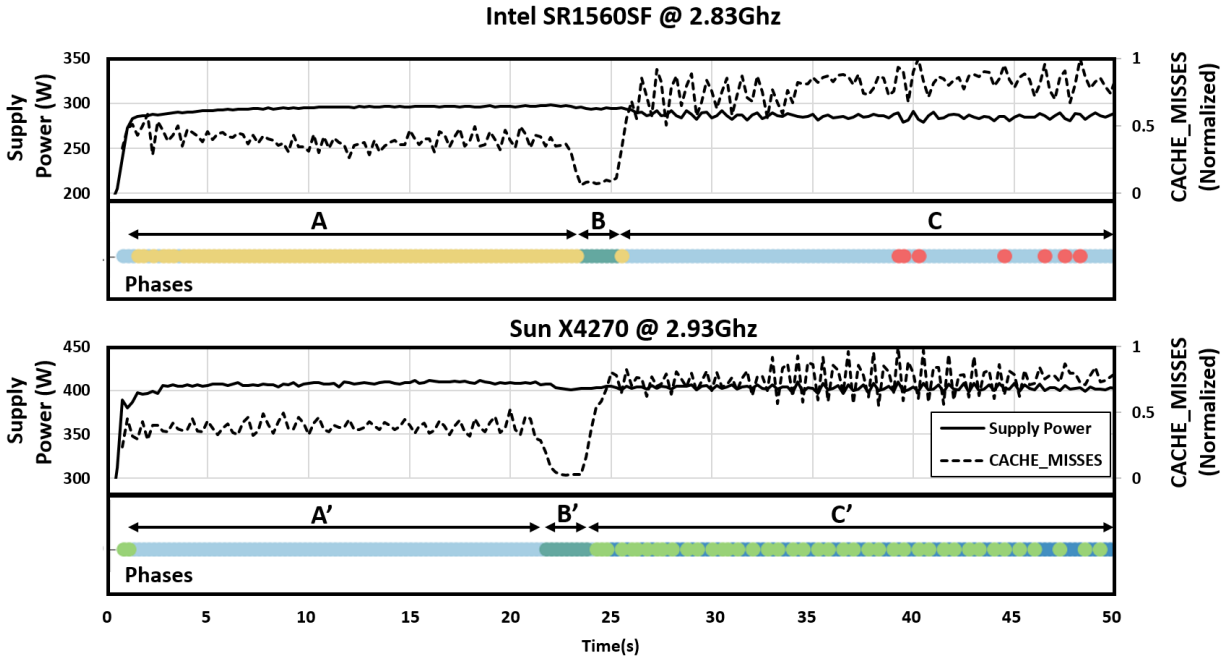


Figure 2.3: Application phases for multi-threaded bzip2 benchmark independently identified for Intel SR1560SF and Sun X4270 servers at the maximum frequency

platform C_A. Also, let $\mathbb{D}_{C_B}^{app_p} (\subset \mathbb{D}_{C_B})$ be the dataset of app_p executed on another platform C_B. To identify the clusters of $\mathbb{D}_{C_B}^{app_p}$ while keeping the identified phase indexes, we apply the k-means algorithm again with the initial cluster centers $\mathbb{V}_{C_A}^{app_p}$. Then, the k cluster centers are moved with the k-means procedure so that each phase is adjusted and fit into the new application dataset. The clusters newly identified for all benchmarks on C_B represent how each cluster on C_A behaves on a different platform C_B.

2.5 Cross-Platform Prediction

We build multi-layer neural network (NN) models to predict the power, performance and energy of applications across machines. Using the power estimation models and the application phase model identified for a single machine, the neural network models learn per-phase PMC event behavior relationship across different machines and settings. Once all models are learned, P^4 can predict the power consumption of different architectures at runtime without actually

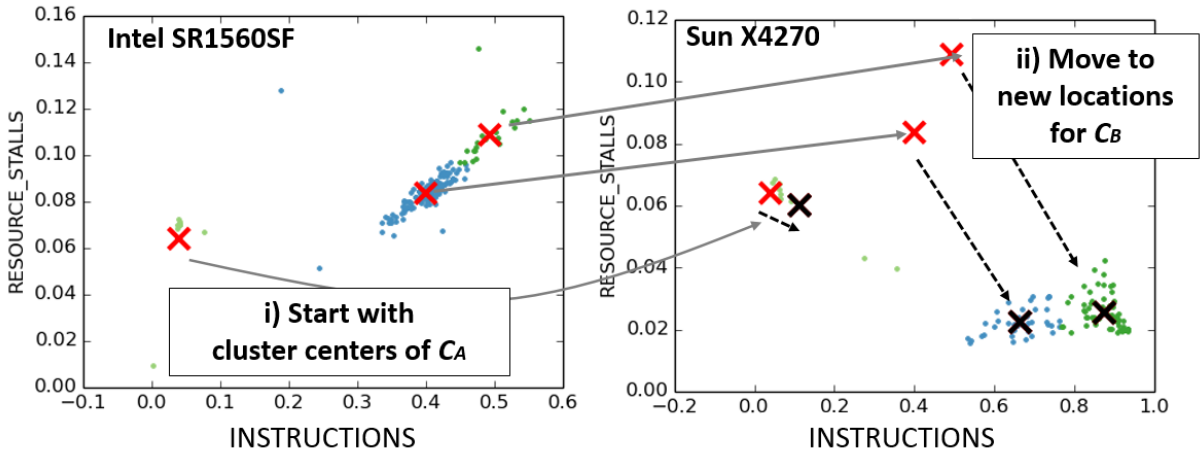


Figure 2.4: Cross-platform phase matching (Intel SR1560SF to SUN X4270, *splash2x.lu_cb*)

running the applications on the architectures. We first describe how to utilize the application phases to capture cross-platform event behaviors in Section 2.5.1. We then show the details of our prediction modeling technique for general applications in Section 2.5.2. Then, we present how the models can perform online predictions in Section 2.5.3.

2.5.1 Phase-Based Training Data Generation

We utilize the application phase as a basic unit to understand event behavior changes across machines and train the cross-platform prediction NN models. Figure 2.5 presents a qualitative comparison of the cross-platform phase behavior for four benchmarks with different colors denoting different clusters (phases). The IntelH (Intel server running at the highest frequency) is used as the reference platform to detect the baseline phases. The phases of the top two benchmarks, *spec.bzip2* and *spec.gcc*, are from single-threaded benchmarks, while the bottom are of multi-threaded ones. The comparison includes different frequencies, i.e. IntelH vs. IntelL (Intel server at the lowest frequency), various platforms, i.e. IntelH vs. SunH, and completely different CPU architectures, i.e. IntelH vs. A15H. The result shows that the phase recognition and matching techniques accurately recognize the phases across different platform

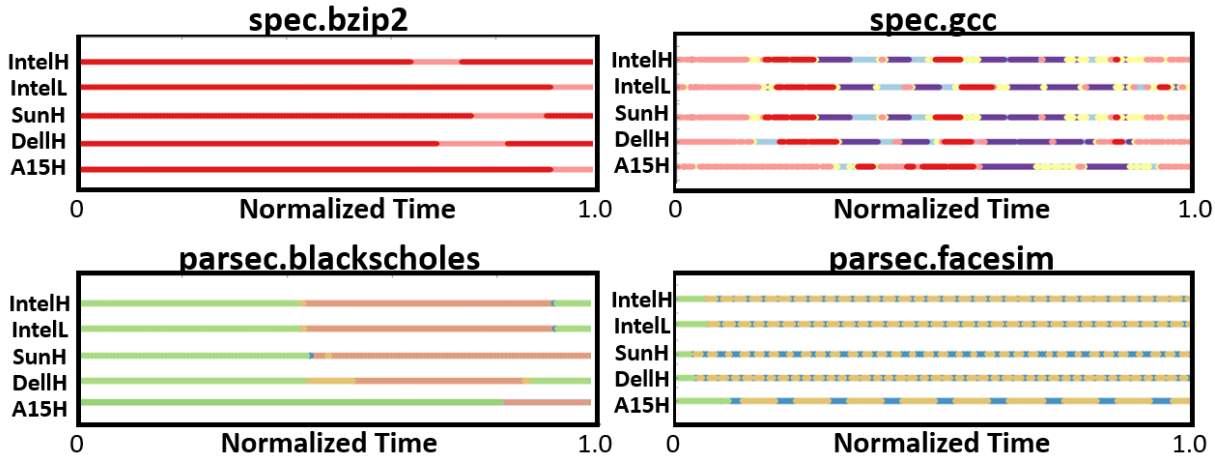
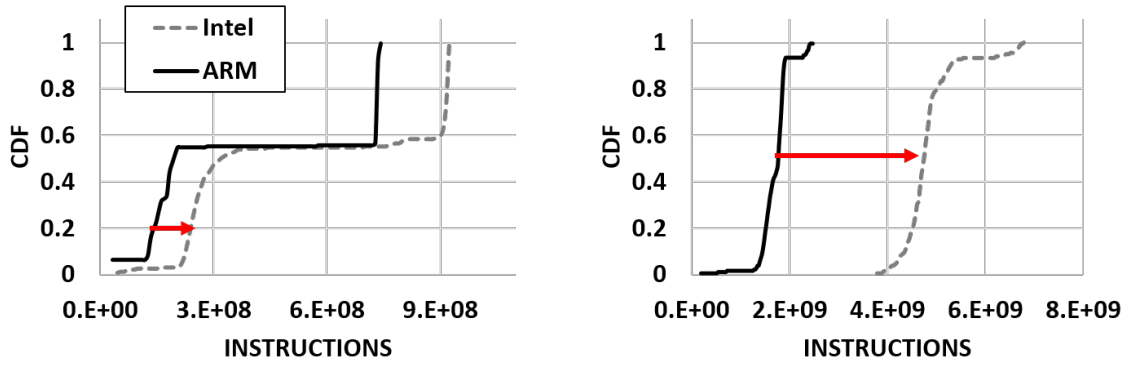


Figure 2.5: Identified phases of four benchmarks running for 60 seconds (IntelH, IntelM and IntelL: Intel server running at highest, medium, and lowest frequency settings. SunH, DellH, and A15H: Sun server, Dell server, and ARM Cortex-15 processor running at highest frequency.)

configurations. For example, the *spec.bzip2* benchmark has a dominant phase, denoted with red color, and an intermediate phase of pink color. The pink color is relatively short on A15H, since the benchmark terminates before completing this phase. Similar findings are also observed for *parsec.blackscholes* in the multi-threaded case. P^4 identifies the cross-platform phases accurately for more complex benchmarks, e.g., *spec.gcc* and *parsec.facesim*.

Since the phases represents the same workloads across machines, we can utilize the phases to capture the event changes between platforms for the prediction model training. Figure 2.6 shows the cumulative distribution function (CDF) graphs of the `INSTRUCTIONS` event for two representative clusters, running on the Intel Xeon E5440 processor at 2.8 GHz frequency and the ARM Cortex A15 at 1.8 GHz frequency. The results show that the per-cluster event distributions have non-linear relationship between different platforms. In addition, although different clusters for each platform could have very different trends, the events in the same cluster behave very similarly across platforms.

We use the per-phase distribution patterns as the dataset to train the neural network model. Let $V_t^{app_p,C_A}$ be an event vector in a computing platform C_A for a benchmark app_p. In the cluster distribution of each event of C_A, we take a percentile of $e_{i,t}^{app_p,C_A}$ of $V_t^{app_p,C_A}$. Then, we identify



(a) Cluster 1 (b) Cluster 2

Figure 2.6: Cumulative distribution of instructions for two clusters

$\widehat{e_{i,t}}$, which is the event value at the same percentile in the C_B's distribution for the same benchmark. For example, in Figure 2.6, the two arrows respectively describe the estimation of `INSTRUCTIONS` at the 0.2 percentile for Cluster 1 and the 0.5 percentile for Cluster 2. By computing this procedure for all the events, we create an event vector $\widehat{V}_t^{app_p,C_B}$ in C_B where each element of the vector is $\widehat{e_{i,t}}$.

2.5.2 Cross-Platform Prediction Model Training

Once the cross-platform phases are identified, we train three NN models for cross-machine power, performance and energy prediction, respectively.

Power prediction model: The first model is called NN_{power} which infers the key events across platforms to predict the time-variant power levels. The NN_{power} has multiple layers and the output layer which corresponds to the performance counters of the predicted platform. We train the NN_{power} model using the per-phase event distributions across machines discussed in Section 2.5.1, i.e., using $V_t^{app_j,C_A}$ as the input and $\widehat{V}_t^{app_j,C_B}$ as the output. Some events are available only for C_B, e.g., `UNALIGNED_LDST_SPEC` of the ARM processor. In that case, $\widehat{e_{i,t}}$ is computed by averaging the event value of the samples that other common events are selected. Since the output layer of this model produces the estimated events on another platform, we

connect the output layer to the single-machine power prediction regression model of C_B, $P_{system}^{C_B}$. It then estimates the power consumption on the machine C_B with the event prediction results of NN_{power}.

Performance prediction model: The second NN model, called NN_{perf}, has a similar structure to NN_{power}, but identifies how many instructions will be executed on a computing platform C_B using PMC events observed on another platform C_A. Let $T_{C_A}^{app_p}$ and $T_{C_B}^{app_p}$ be the execution time of a benchmark that runs on each platform, C_A and C_B. When the events are sampled at the interval of I_{sample}, P^4 collects $N_{C_A} = \frac{T_{C_A}}{I_{sample}}$ and $N_{C_B} = \frac{T_{C_B}}{I_{sample}}$ respectively for each platform. If the benchmark executes monotonous workloads during their executions, a sample collected on C_A corresponds to the cumulative sum of R samples on C_B, where $R = \frac{N_{C_B}}{N_{C_A}}$. We use this estimation for the samples in each phase, since the identified phase represents such homogeneous workload.

Let us recall $V_t^{app_j,C_A}$ and $\widehat{V}_t^{app_j,C_B}$ which are calculated with the per-phase event distributions. From $\widehat{V}_t^{app_j,C_B}$, we extract the estimated number of instructions on C_B, say $\widehat{Inst}_t^{C_B}$. Then, N_{perf} is trained with $V_t^{app_j,C_A}$ as the input and $R \cdot \widehat{Inst}_t^{C_B}$ as the output, where R is the ratio of the samples for the phase between the two platforms.

Energy prediction model: The two aforementioned models perform detailed sample-by-sample predictions in millisecond-level granularity. To efficiently predict cross-platform energy consumption for a longer time horizon, e.g., in second/minute-level granularity, we train another model, called NN_{energy}. Let $\mathbf{E}_{P_i}^{app_j,C_A}$ be a set of multiple event vectors corresponding a phase P_i for a benchmark app_j. P^4 adds the event values for each metric in $\mathbf{E}_{P_i}^{app_j,C_A}$ to create a cumulative event vector, $V_{P_i}^{app_j,C_A}$, while calculating its total execution time. Once we perform the accumulation procedure for all the benchmark applications and phases, we obtain multiple cumulative event vectors, $\mathbf{V}^{C_A} (\in V_{P_i}^{app_j,C_A})$, with the execution times as the input variables of the training dataset. To train the model with diverse combinations of phase sequences, P^4 also creates another set of training inputs by randomly selecting an event vector from $\mathbf{V}^{C_A}$ and adding

it with $V_{P_i}^{app_j,C_A}$ in order. The output variable of the training dataset is the energy consumed on C_B for the same phase. Then, the NN_{energy} model performs the energy prediction for C_B with a sequence of samples monitored on C_A.

2.5.3 Online Prediction

P^4 utilizes the three NN models online in two fashions: i) time-variant power level prediction and ii) per-task energy prediction.

Time-variant power level prediction: The time-variant power consumption is predicted at runtime using NN_{power}, NN_{perf} and the single-machine power estimation model described in 2.4.1. The predicted event vector, $\widehat{V}_t^{app_j,C_B}$, is used as an input of the regression model since the NN_{perf} predicts how events will behave on a different platform for the same application. The execution time for a sampling period on a different platform, τ_{C_B}, can be predicted by using that on the current platform (τ_{C_A}), NN_{perf} which estimates the number of instructions executed on C_B (I_{C_B}), and an output neuron of NN_{power} which produces the speed in Instructions Per Sampling interval (IPS) on C_B (IPS_{C_B}) with the following equation[2]:

$$\tau_{C_B} = \tau_{C_A} \times \frac{I_{C_B}}{IPS_{C_B}} \tag{2.2}$$

Figure 2.7a illustrates the online prediction procedure as a feed-forward network for a platform pair, C_A and C_B. Once the PMC events are collected as event vectors for an interval τ_{C_A} on C_A, the sampled event vector is input into the two neural networks. The number of instructions of C_B, i.e., I_{C_B}, identified by NN_{perf}, is delivered to the execution time conversion function, π, which computes τ_{C_B} based on Equation 2.2. When P^4 predicts power consumption for different frequencies on the same platform, the NN_{perf} is not activated since the number of instructions required to run the workload is the same regardless of frequency they are run at. Thus, the IPS of

[2]Due to the space limitation, we do not include the detailed proof steps. It can be derived from $\tau_{C_i} \times IPS_{C_i} = I_{C_i}$ and $I_{C_A} = IPS_{C_A}$ in a straight-forward way.

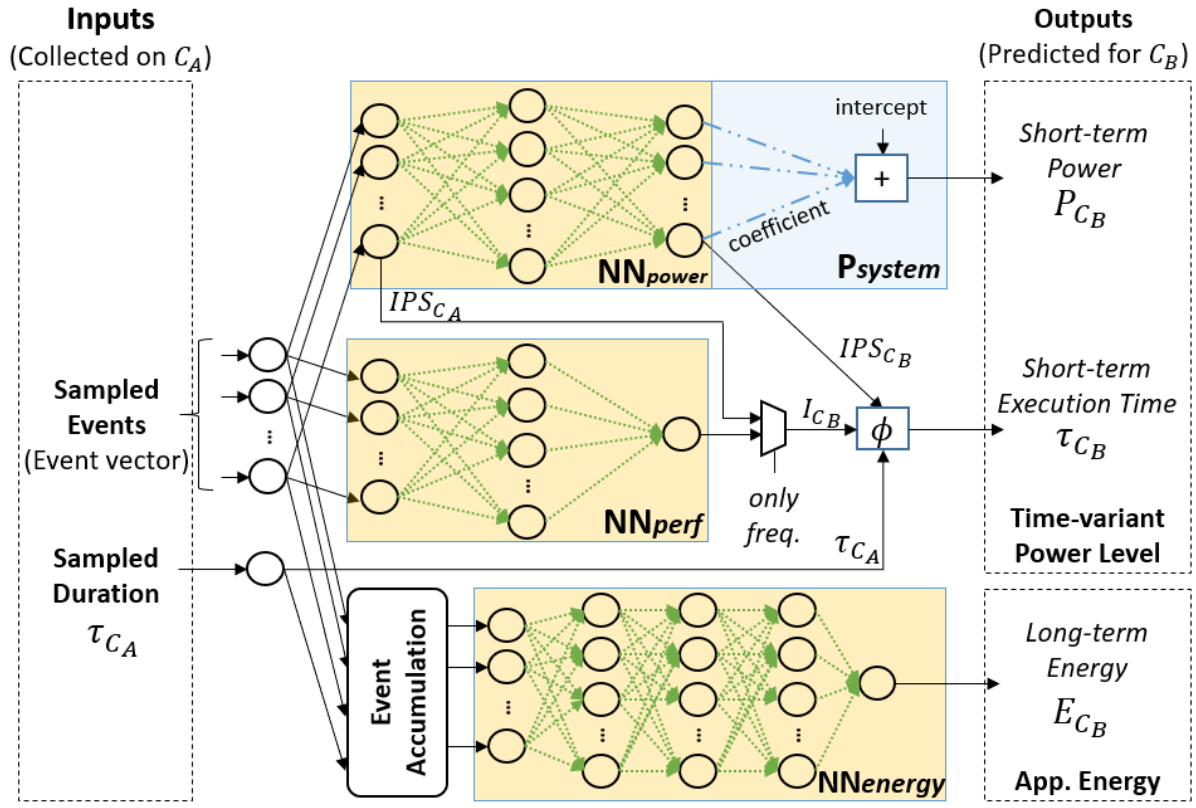


Figure 2.7: Feed-forward neural networks for online prediction

C_A is provided to the execution time conversion function. At the same time, the power is predicted using the regression-based power estimation model where the input of the regression model is given as the output of the NN_{power}.

Per-task energy prediction: The energy prediction model, NN_{energy}, is trained to predict with the accumulated event values, unlike NN_{power} and NN_{perf} built with per-sample basis. For this case, P^4 keep adding the monitored events online, and performs the prediction whenever it is needed. In our experiments, this model requires more layers and neurons for accurately estimates, as it internally combines two different prediction goals as well as the single-machine power prediction model. However, it predicts the energy for multiple collected samples at once, resulting in lower computation overhead than the time-variant power level prediction. In Section 2.7, we discuss the model complexity and runtime overheads of the online prediction in detail.

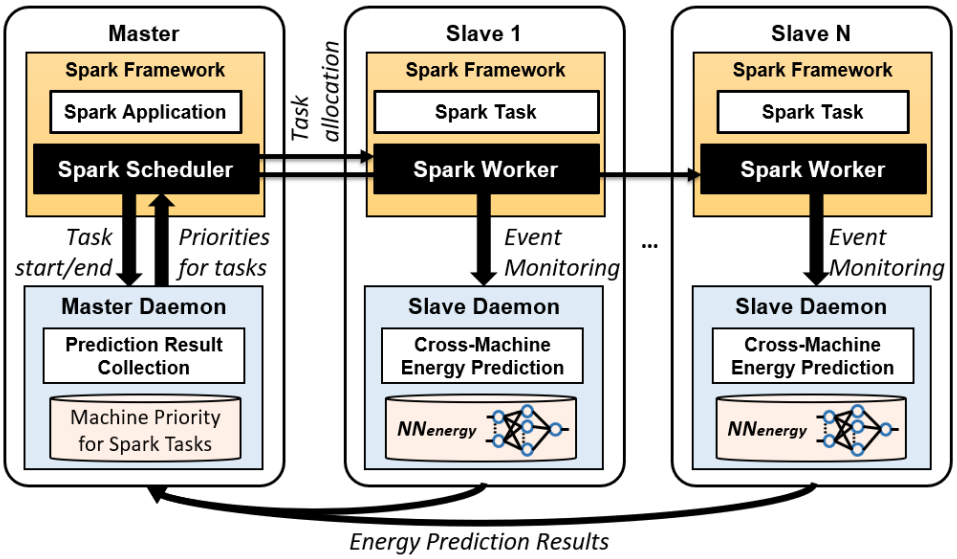


Figure 2.8: Overview of model-driven management on Spark environment

2.6 Cross-Platform Management for ML tasks

The proposed P^4 framework can be utilized for diverse cross-platform analysis and management problems. In this section, we present such a predictive management framework which automatically allocates tasks over hierarchical systems on a distributed computing environment, Apache Spark.

2.6.1 Cross-Platform Management Framework

Figure 2.8 shows the overview of the P^4-based task management framework integrated with Spark which distributes tasks of state-of-the-art learning algorithms. We modified the Spark scheduler to track the task life cycle and change scheduling decisions as a function of our modeling framework. The Spark scheduler communicates with a daemon, called *master daemon*, which decides the best task allocation based on the energy predictions. The prediction results across machines are offered by *slave daemon* which runs on Spark slave nodes while collecting the key events at runtime.

Modified Spark scheduler: An application running on the Spark framework initiates parallelizable functions. The Spark framework is responsible to perform each function in the distributed fashion. To this end, a directed acyclic graph (DAG) scheduler interprets the function into multiple stages which have to be sequentially executed, and execute the *Spark tasks* for each stage on the slave nodes. The default Spark scheduling policy is a randomized round-robin to provide a fair task distribution across the slave machines in terms of the number of tasks.

In our design, we modify two parts of the Spark Scheduler. First, when a task is allocated to a node, we track the start/end of each task with other required information (e.g., allocated node). The monitored information are send to the master daemon running on the same machine. Second, during the task allocation decision, the scheduler asks to the master daemon if the prediction results are available for the allocated task. When the prediction has been made for the target task, instead of using the randomized policy, we assign the task into the optimal machine, e.g., the slave which is expected to consume the lowest energy.

Master daemon: The master daemon relays the task life cycle sent by the modified scheduler to the slave nodes. Once the task is finished, it collects the cross-platform energy prediction results from the slave nodes, and then computes the *priority* of the slave machines for the task based on a *cost function*, e.g., energy consumption or energy price. When the scheduler requests the prediction information for the task, it looks up the computed priorities to select the most efficient machine among all the slave which have not been assigned with any other tasks.

Slave daemon: The slave daemon is responsible to provide cross-machine prediction results to the master daemon. The three prediction models, NN_{power}, NN_{perf}, and NN_{energy} can be used for the management; in this work, we use the NN_{energy} model to predict the entire energy consumption for each task. It monitors the key events of the running tasks in the background, and calculates the accumulated event vector using the task life cycle which is provided by the master daemon. It then performs the energy prediction using the NN_{energy} model with the event vector. The energy prediction results are sent to the master daemon so that it can update the priorities for

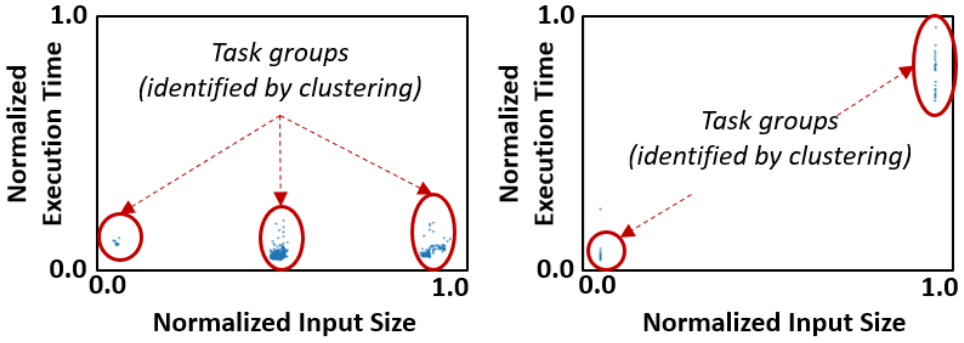
the next task run as a closed-loop control.

2.6.2 Application Task Extraction

Since traditional benchmark suites such as SPEC2006 [56] and PARSEC [57] run the same tasks for every run, we can easily obtain the event traces for multiple machines by executing them on each platform of interest. In contrast, the ML applications running on distributed systems such as Apache Spark [25] are automatically parallelized across machines, and as the result, each machine has different amounts of workloads to run.

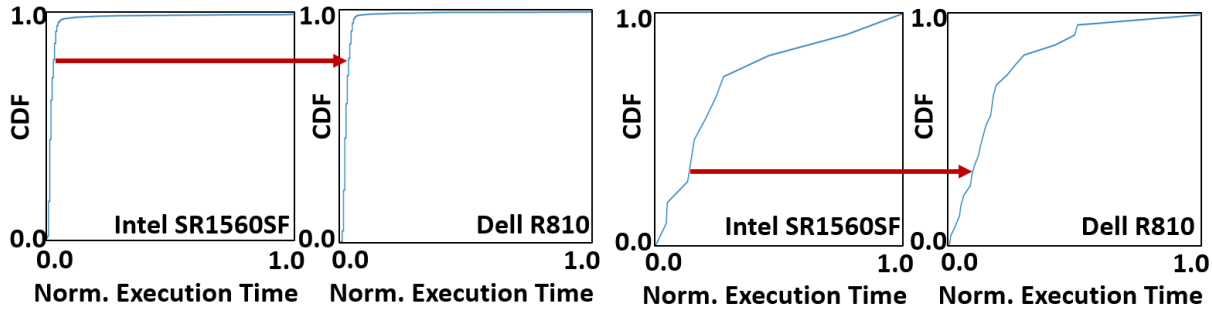
To perform the prediction for the paralleled task in Spark, in short *Spark task*, we monitor the task distribution traces to extract when/where each task is started and finished. Since the Spark task is the minimal execution unit initiated by a user-defined function call, we can regard them as a single workload like an application of the traditional benchmark suite. An remained issue is that the workload behavior can be affected by the input data of the distributed tasks. Figure 2.9 shows the execution time according to the input size for two Spark benchmarks. The plots show that, although the execution times may vary for different input sizes, the tasks which have the similar input size exhibit very similar behaviors.

P^4 identifies the same tasks across machines based on this finding. We first group all the tasks into multiple *task groups* by using DBScan clustering algorithm [58] with the input sizes, and then calculate the distribution of the execution time for the task groups. Figure 2.10 shows the execution time distributions for task groups of two benchmarks as examples. The results show that each task group has an unique pattern in their distributions, even though the task groups may have different characteristics each other. We exploit the percentile-based method discussed in Section 2.5 to extract the most similar workload pairs for each task group. The workload pairs are in turn regarded as the cross-machine application runs in the prediction procedure.



(a) Strongly Connected (b) Triangle Count

Figure 2.9: Task group identification for two Spark applications



(a) Auto encoder (b) Pregel Orientation

Figure 2.10: Cumulative distribution of execution times for two task groups

2.6.3 Task Allocation Case Study

The proposed management framework eventually decides the task allocation with the cost function. In this section, we describe two general cost functions in the hierarchical setting, i) total energy use and ii) cluster-level energy cost. It is beyond the scope of this dissertation, but it is worth noting that the cost function can be combined with other well-known control knobs existing on the individual machine, e.g., dynamic voltage and frequency scaling (DVFS).

Energy Use Optimization: In this case, we prioritize the worker machine whose predicted energy is less than others. Since a set of the same task is usually distributed to different slaves, the master daemon may have multiple prediction results for each machine and task. Thus, we calculate the cost function as the average energy predicted. Then, we add it with

the communication energy cost which is estimated using the $P_{network}$ model with the network bandwidth and data size of the task. It is because the cross-machine prediction results only account the energy use of the slaves without considering the cost on the master side to distribute the data. Eventually, the scheduler allocates the task on the machine which consumes the least energy for both computation and communication in total. This policy is effective when the scheduler has multiple choices of the slave machines in allocating a task.

Energy Cost Reduction: Modern data centers may be deployed with multiple clusters located at different places [59]. Each cluster typically have long-term power contracts, and are charged market prices when exceeding the contract. The overages can be five times more expensive than the contracted price [60]. In addition, the clusters may have asymmetric energy pricing due to the contracts and energy source [61]. For this case, we compute the second cost function by multiplying the energy price, e.g., price per Wh, with the server energy prediction which is estimated in the same way to the one described above. This policy allocates more tasks into cheaper clusters, as long as the slave machines belonging to it are available.

2.7 Experimental Setup

The proposed P^4 framework has been implemented using Python 2.7 with Scikit-learn 0.17.1 library for the statistical analysis [62]. We also use the Tensorflow framework [63] to process the neural network models on GPGPU. The offline modeling procedure has been run on a system that has Intel i7-6700k quad-core CPU and Nvidia GTX 1080 Ti. We conducted the measurements on three servers and a mobile platform: Intel SR1560SF, Sun X4270, Dell PowerEdge R810, and Odroid XU3. Table 2.3 summarizes the specifications of each platform. For the distributed workload evaluation, we deployed the Spark framework on a hierarchy of the servers using Docker [64] and Weave [65] that run Apache Spark 2.0 [25] and Hadoop environment 2.7.3 [66]. Each cluster is separated with a network switch whose bandwidth is

Table 2.3: Evaluated heterogeneous platforms

Type	Model	Hardware component description
Server	Intel SR1560SF	Intel Xeon E5440 1.99GHz $\sim$ 2.83GHz L1 cache sizes: 64KB, DRAM size: 8GB
Server	Sun Fire X4270	Intel Xeon E5500 1.6GHz $\sim$ 2.93 GHz L1 cache sizes: 64KB, DRAM size: 24GB
Server	Dell PowerEdge R810	Intel E7 4870 Westmere 2.39 GHz $\sim$ 1.06 GHz L1 cache sizes: 32KB, DRAM size: 128GB
Mobile	Odroid XU3 (ARM)	Exynos5422 ARM big.LITTLE Cortex-A15 big: 1.4 GHz $\sim$ 1.0 GHz, Cortex-A7 LITTLE: 2.0 GHz $\sim$ 1.2 GHz L1 cache sizes: 32KB, DRAM size: 2GB

100Mbps, while the intra-cluster bandwidth is 1Gbps.

For the server systems, we measure the supply power using the HIOKI 3334 power meter, while the power consumption of Odroid XU3 is measured by reading the embedded sensors on each core. The server systems have also been instrumented to measure power consumption of subcomponents by reading voltage drop of two 0.1Ω shunt resistors. All the power measurements are sampled at a rate of 100 ms.

The experimental results for the estimation and prediction are cross-validated using the "leave-one-out" strategy to evaluate each benchmark by separating the tested program from the training set. We pick a benchmark for testing the online identification stage while all other benchmarks are used to build the models in the offline learning stage. This cross-validation was performed for all benchmarks. The accuracy of the estimation and the prediction is evaluated using Mean Absolute Percentage Error (MAPE) [67].

2.7.1 Benchmarks

Computing-variety benchmarks: We use industry-standard benchmarks which represent a wide range of computing workloads. Benchmarks are executed on each platform with

varying number of threads and at various processor frequencies. For each platform, we execute the benchmarks at three processor frequency levels, i.e., lowest (L), medium (M) and highest (H). The benchmark set has both single-machine benchmarks and distributed benchmarks. The single-machine benchmarks includes SPEC2006 [56], PARSEC/SPLASH2x [57] with native inputs, NERSC datacenter benchmarks [68], and Linpack [53], which has been used for TOP500 runs [69]. The distributed benchmarks are run on Spark environments, and includes IBM Spark-Bench 2.0 [70], which has machine learning, graph computation, SQL, and streaming workload, and Intel BigDL [71] which runs popular deep learning models. To execute the same workload on the low-power ARM machine, we cross-compiled SPEC2006 suite and PARSEC benchmarks. Due to the ISA difference and library dependencies, the other benchmark suites could not be ported to the Odroid ARM processor. In total, we could execute 154 benchmarks on the three server platforms, and 88 benchmarks on the ARM platform.

IO benchmarks: The aforementioned benchmarks are useful to test target systems with diverse types of computing workloads. However, we observe that they typically represent limited variety in IO usage. To train accurate models for the disk and network, we use IOZone [72] and iperf [73] to create a synthetic mix for various disk and networking usage patterns, along with TPC-H [74] benchmark which is a business-oriented DB benchmark.

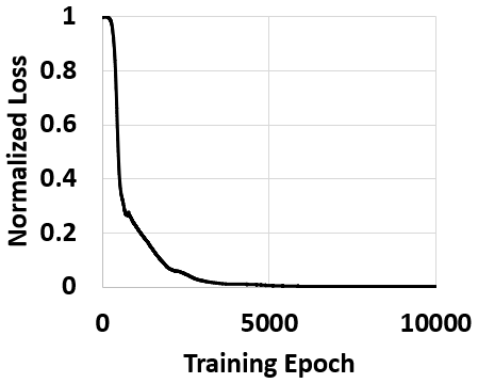
All the benchmark applications were run on each target system for more than 3 hours in total, while collecting the performance counter events using *perf* tool and the system IO events through sysfs at a rate of 250ms sampling interval.

2.7.2 Model Training Parameters

The neural network models for the cross-machine prediction have tunable hyperparameters that affect the model accuracy and computation overhead. We trained the models with the standard ML practice of grid search along with cross-validation. Figure 2.11a summarizes the parameters used for each model. The first layer of each model uses the hyperbolic tangent activation function

	Hidden Layer Topology	Training Epoch	Batch Size
NN_{power}	30	100	64
NN_{perf}	30	100	64
NN_{energy}	50/30/10	10000	128
Common parameters			
Activation	Tangent (First) / Relu (Middle) / Linear (Last layer)		
Loss Metric	Mean Squared Error		
Others	L2 Regularization, Adam Optimizer		

(a) Neural Network Configuration



(b) Training Loss Change

Figure 2.11: Cross-platform NN model configurations

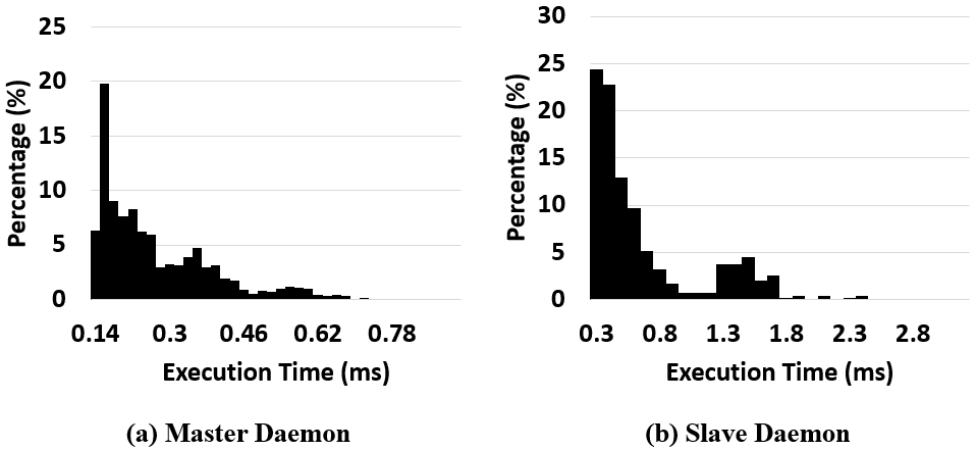
to convert the event values to nonlinear hyperplanes; the last layer uses the linear activation function which combines multiple neuron outputs predicted by ReLu to perform the desired regression tasks. As discussed in Section 2.5.3, the NN_{energy} model needs more complex structure and higher training epochs. Figure 2.11b shows the training loss change of NN_{energy} training procedure. Since the loss converges after 5000 epochs, we train for 10000 epochs to evaluate with the sufficiently learned model.

2.7.3 Overhead

Table 2.4 shows the overhead to process the models supported in the P^4 framework. We report the average process time of each event vector for the online stage and the model training time of each platform pair for the offline stage. In the online stage, P^4 computes the feed forward network and only requires selected event counters. The runtime overhead to process each event vector is less than 676 μs. Compared to the PMC sampling rate of 250 ms, the runtime overhead

Table 2.4: Overhead of P^4 models

	P_{system}	C_{phase}	NN_{perf}	NN_{power}	NN_{energy}
Online	1.8 μs	*N/A*	88 μs	34 μs	252 μs
Offline	4.6 s	161 s	27.7 s	24.5 s	36.9 m



(a) Master Daemon **(b) Slave Daemon**

Figure 2.12: Overhead of model-driven management

is negligible. Most overhead of the offline learning stage comes from the benchmark execution. In our evaluation, the offline learning stage was performed for 270 minutes including the benchmark application execution. Since the learning happens only once for each machine at offline, the overhead of the modeling stage is negligible.

We also evaluate the overhead when integrating P^4 with the Spark framework for the management described in Section 2.6. Figure 2.12 shows the histograms of the execution time overhead for the master and slave daemon. The master daemon obtains and reorders the machine priorities for each task with a minimal overhead of 0.27ms on average. It also takes 3.9 ms to update the machine priorities in a separated thread at background. For the slave overhead, the runtime overhead to make each prediction is 0.7ms, including the system event monitoring. The CPU utilization of the slave daemon is always less than 0.1%.

2.8 Evaluation of P^4 Models

2.8.1 Full-System Power Estimation

The learning procedure of P^4 trains the power estimation models for each system component and the system total power by monitoring the key system events. In this section, we evaluate

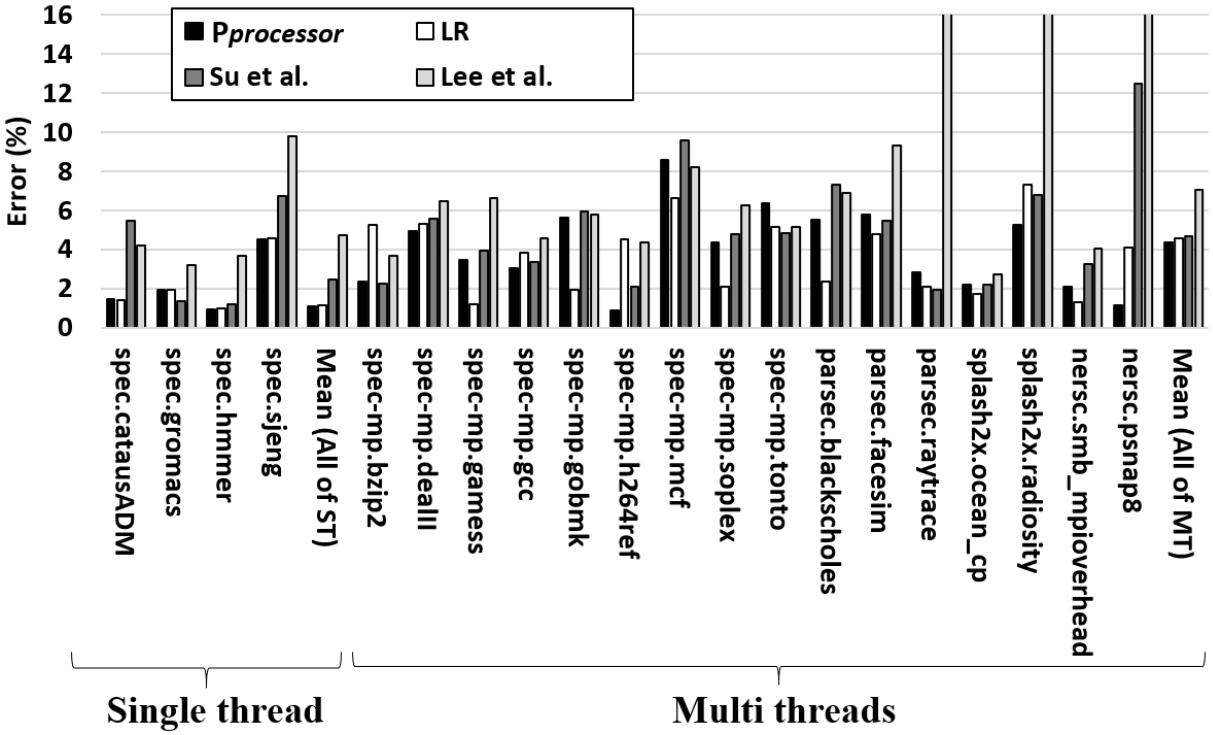


Figure 2.13: Processor power estimation errors (Intel SR1560SF)

each power model described in Section 2.5.2.

CPU: The $P_{processor}$ model is built by the Lasso method that performs the automated event selection with building a regression model. We compare it against linear regression which state-of-the-art methods have used. The linear regression method uses 10 additional performance counters, including TLB misses, thermal trip, SSE execution, and snoop-related events, on top of the event counters that P^4 selected. We also compare the results with two state-of-the-art processor models published in Su et al. [49] and Lee et al. [34]. The models exploit 8 and 3 events, respectively, selected based on the domain knowledge of their architecture.

Figure 2.13 shows the comparison of estimation accuracy for processor power of the Intel server for the single thread and the multi threads cases running at the highest frequency. Because of the limited space, we show 19 representative benchmarks and the average error for all tested benchmarks. The results show that Lasso estimates processor power accurately with 4.3% error on average, even though Lasso is using only a subset of available performance counters. The

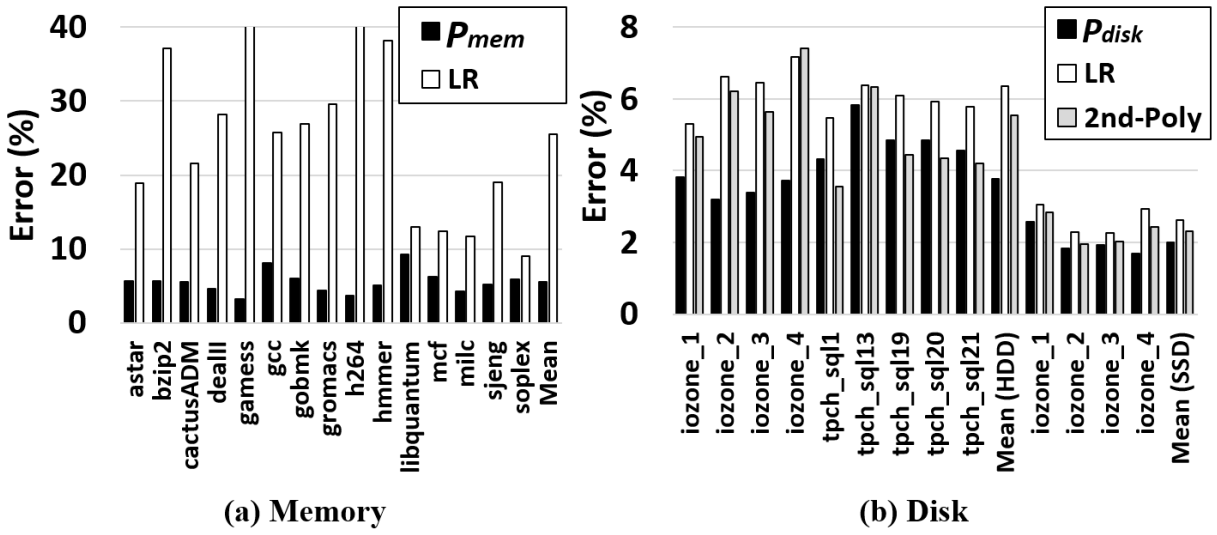


Figure 2.14: Power estimation error of subcomponents (Intel SR1560SF)

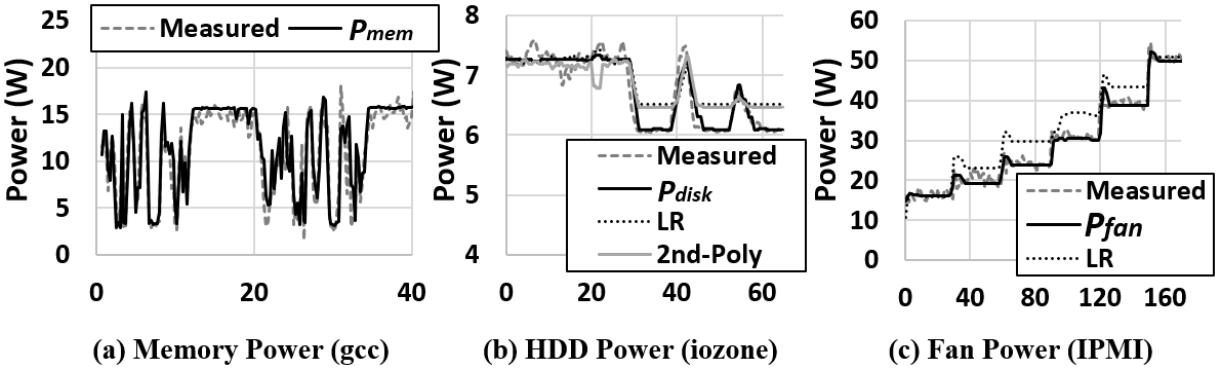


Figure 2.15: Runtime subcomponent power estimation (Intel SR1560SF)

Lasso estimation error is similar to the linear regression (LR) model, which uses 22 events, with only 0.2% difference. Lasso model has better accuracy than the two published models, since P^4 automatically selects strongly power-related events for the given target platforms. In addition, it shows comparable results to the NN-based model. Thus, we conclude that the events statistically selected by P^4 provide very accurate power estimates without the need for domain knowledge as was done by Su et al. [49] and Lee et al. [34].

Subcomponents: As discussed in 2.4.1, the P^4 framework also creates the power models specialized for each system subcomponent, such as memory, disk, network and fans,

using different automated modeling strategies beyond the linear regression. Figure 2.14a shows the power estimation error for the memory compoment. The results show that the P_{mem} model accurately identifies the memory subcompoments with less than 6.5% error in MAPE. In contrast, the linear regression-based power model performs the inaccurate prediction for most benchmarks since it could not capture the non-linear relationship to the memory bandwidth utilization.

Figure 2.14b reports the estimation error of P_{disk} for the disk power consumption as compared to the linear regression-based model [27] and second-order polynomial regression method (2nd-Poly.) We trained the models using the IO benchmarks explained in Section 2.7.1. The P_{disk} model utilizes Isotonic regression method to automatically identify the internal disk states which can be changed with the disk IO. The results show that, the proposed model outperforms the other stateless regression models. The estimation error is 3.77% for HDD and 2.01% for SSD on average for all the benchmarks. In our evaluation, the Isotonic modeling method also accurately identifies the networking power model. The key event is identified as ethernet connection, as the network consumes 2W while ethernet is connected to the network for packet transfer, i.e., in case of $\mathbf{e}_{networkIO} > 0$. However, network bandwidth utilization does not have an observable effect on its power usage of the tested machines.

Figure 2.15 shows the runtime power estimation results for the memory, HDD, and fan subcomponents. The proposed compoment models accurately estimates the power of the major subcompoments by only monitoring the relevant key events. For example, P_{mem} performs accurate estimates that follow the high power fluctuation of the memory component during the benchmark execution, and P_{disk} automatically identifies the low-power state unlike the LR and 2nd-Poly model. P_{fan} also accurately estimates the fan power consumption with 0.8% error for diverse fan speeds controlled by IPMI.

System Supply Power: Figure 2.16 summarizes the power estimation error of P_{system} for the tested platforms. For this evaluation, we use the computing-variety benchmarks described in 2.7.1, and report the average error of the MAPE values cross-validated for each benchmark

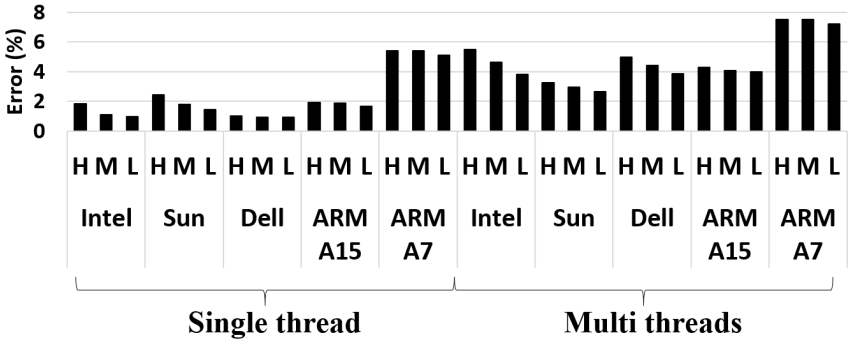


Figure 2.16: Average error of single-machine supply power estimation. ARM A15 and A7 represents respectively either Cortex A15 or A7 processor

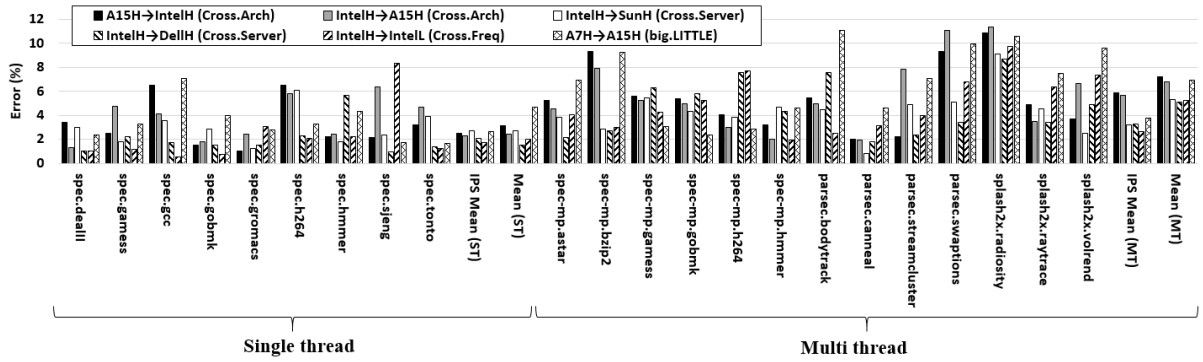


Figure 2.17: Summary of time-variant power prediction accuracy

application. The result shows that P^4 accurately estimates the total power consumption for different target platforms. The power estimates of multicore and higher frequency cases are more challenging due to the larger fluctuations in power levels. Nevertheless, P^4 estimates power with 5.4%, 3.23%, 4.98%, 4.28%, and 7.5% of average error for the Intel, Sun and Dell servers, Cortex A15, and Cortex A7 respectively. The error on Cortex A7 is a bit higher than others, since the processor has relatively low static power, making it highly sensitive even to small errors. However, even for the worst case benchmark, the model can estimate within 13% of error.

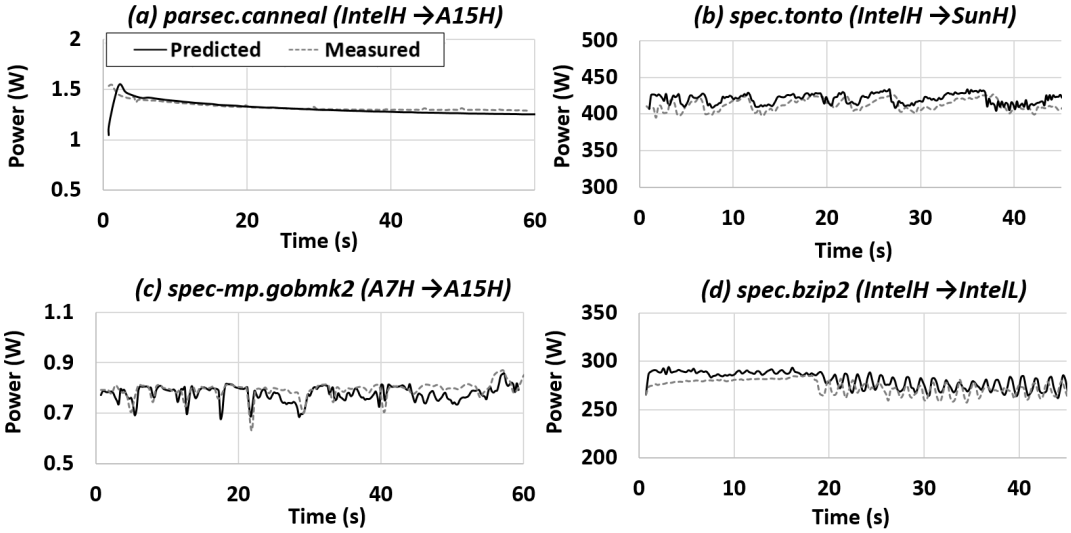


Figure 2.18: Time-variant power level prediction for four heterogeneous platform combinations

2.8.2 Cross-Platform Prediction

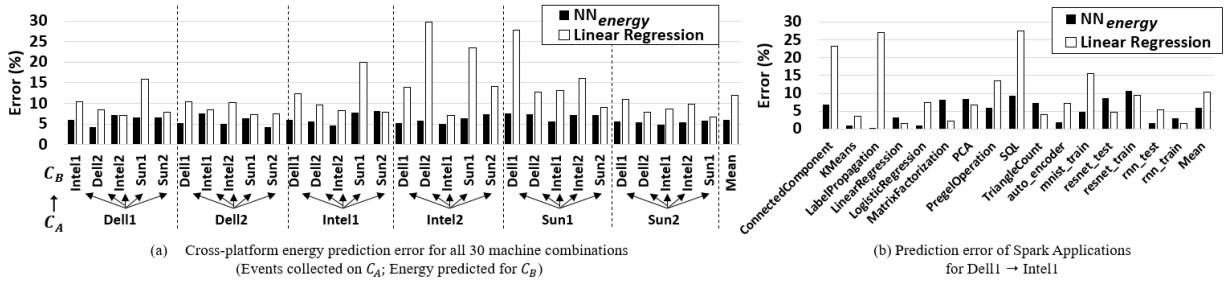
One of our key contributions is the generalized power prediction capability across heterogeneous computing platforms and system configurations. In the followings, we evaluate the cross-machine prediction scenarios discussed in Section 2.5.3, i) detailed time-variant power levels using NN_{power} and NN_{perf}, and ii) the task energy using NN_{energy}.

Time-variant power level prediction: Figure 2.17 reports the prediction results for the six different cases. In the evaluation, we observed that our methodology can accurately predict performance and power consumption. P^4 gets 5.2% error on average for predicting time-varying power consumption on servers for multi-threading benchmarks. Similarly, when comparing completely different architectures for multi-threading benchmarks, P^4 gets 7.2% and 6.8% error on average, for A15H-to-IntelH and IntelH-to-A15H cases respectively. Note that, for in this case the number of threads is also different, i.e., 8 on Intel vs. 4 on ARM A15, as well as their frequency levels. When predicting power consumption for the big-LITTLE example (A7-to-A15), the error is 6.9%. We also compute the performance prediction error with the IPS metric. The results show that the average error is less than 6% for even the most challenging

cross-architectural prediction cases such as A15H-to-IntelH and IntelH-to-A15H. Thus, P^4 can accurately predict the power and performance for the complex combinations, including changes in the number of threads, CPU frequencies, platforms (mobile to server), and CPU architectures (x86 to ARM).

Figure 2.18 shows how the proposed P^4 predicts power consumption using the online prediction network described in Section 2.5.3. The results show four heterogeneous platform combinations, (a) two different-architecture, Intel x86 to ARM Cortex A15, (b) a cross-server case from IntelH to SunH, (c) a big-to-LITTLE example in moving from A7 to A15, and (d) a frequency change from IntelH to IntelL, for four representative multi-threaded benchmarks. The results show that based on the trained neural networks, P^4 can accurately predict power changes over time for all the heterogeneous platform combinations. The execution time for each benchmark is also predicted for each of the platforms. For example, the prediction of *parsec.canneal* of the 60 seconds on Cortex A15 is made with the events for 18.5 seconds observed on IntelH. This means that P^4 can predict both instantaneous power and performance changes using monitored events.

Task energy prediction: P^4 also performs the cross-machine energy prediction for a long-term interval with a single NN structure. To evaluate the task energy prediction in a practical scenario, we used a distributed Spark environment deployed with six servers in which two for each of the Intel, Sun, and Dell servers are included. Figure 2.19a shows the average prediction errors of the Spark benchmark applications for all the 30 cross-machine cases. The result shows that the NN_{energy} model accurately predicts energy for Spark tasks with 6.1% error on average for the 30 combinations. We also compare the prediction results to the linear regression models which are trained with the same dataset used in the NN_{energy} modeling. When predicting the cross-machine energy consumption across the same machine settings (e.g., 'Dell1-to-Dell2' and 'Intel1-to-Intel2'), the error of the linear regression model is 8.3% error on average. However, the linear regression model presents the relatively high error to predict across completely different



(a) Cross-platform energy prediction error for all 30 machine combinations (Events collected on C_A; Energy predicted for C_B)

(b) Prediction error of Spark Applications for Dell1 → Intel1

Figure 2.19: Cross-platform energy prediction accuracy. The error for each case shown in (a) is the average error cross-validated for all benchmark applications.

machines (e.g., 'Dell1-to-Intel1' and 'Sun1-to-Dell1'), showing the average error of 13.0% and the worst-case error of 29.8%. In contrast, the deep learing-based model, NN_{energy}, provides stable prediction for both the same machine setting and different machine cases with 5.3% and 6.3% error, respectively. Figure 2.19b shows the detailed evaluation results for a representative case of 'Dell1-to-Intel1'. The results show that the NN_{energy} model accurately predicts all the benchmark applications by capturing the non-linear relationship between the events and cross-machine energy consumption, while the linear regression model often fails to provide accurate cross-machine energy estimates.

2.9 Evaluation of Model-Based ML Task Allocation

In this section, we evaluate the predictive management framework for the ML task allocation described in Section 2.6. The evaluation have been done on the hierarchy of six servers running Apache Spark. This experimental hierarchical systems has two clusters, where each cluster has three servers, i.e., one each of Intel SR1560SF, Sun Fire X4270, Dell PowerEdge R810. In followings, we evaluate two case studies for the task allocation discussed in Section 2.6.3, i) energy use optimization and ii) energy cost reduction.

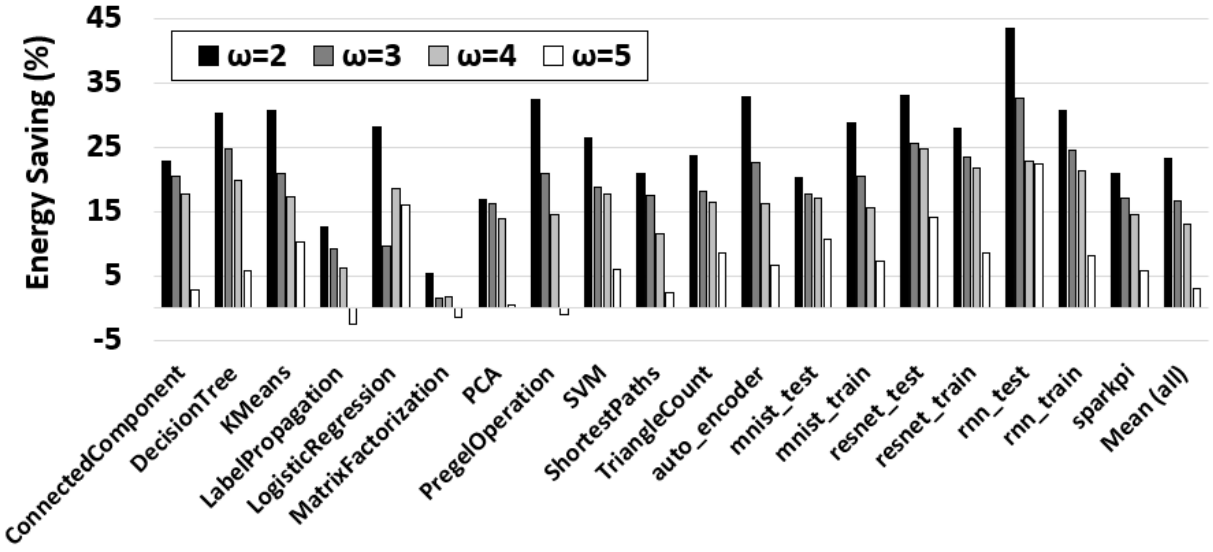


Figure 2.20: Summary of energy use optimization

2.9.1 Energy Use Optimization

In this case study, the policy in the management framework allocates tasks into the machine whose is predicted to use minimal energy. We compute how much energy can be saved by the model-based management policy as compared to the default policy which chooses a random machine to allocate a task. For this evaluation, we define the number of parallelized tasks for each benchmark application as ω. Since we have six servers, the optimization policy has a chance to save the energy when $\omega < 6$. We vary $\omega < 6$ from 2 to 5 for each benchmark.

Figure 2.20 shows the evaluation results for different Spark benchmarks. The model-based management policy saves energy by 16.5% when P = 3, i.e., when allocating each of three tasks among one of six servers. When P is small, there is more chance to choose more energy-efficient machines against the default randomization policy, resulting in the higher energy saving. Figure 2.23 compares the breakdown of the energy consumption for each server. The results show that it achieves high energy efficiency by utilizing the energy-efficient servers. For example, since Intel and Sun servers processed more tasks, since they are likely to be more energy-efficient for many applications in our setting. In contrast, in the default policy, each server

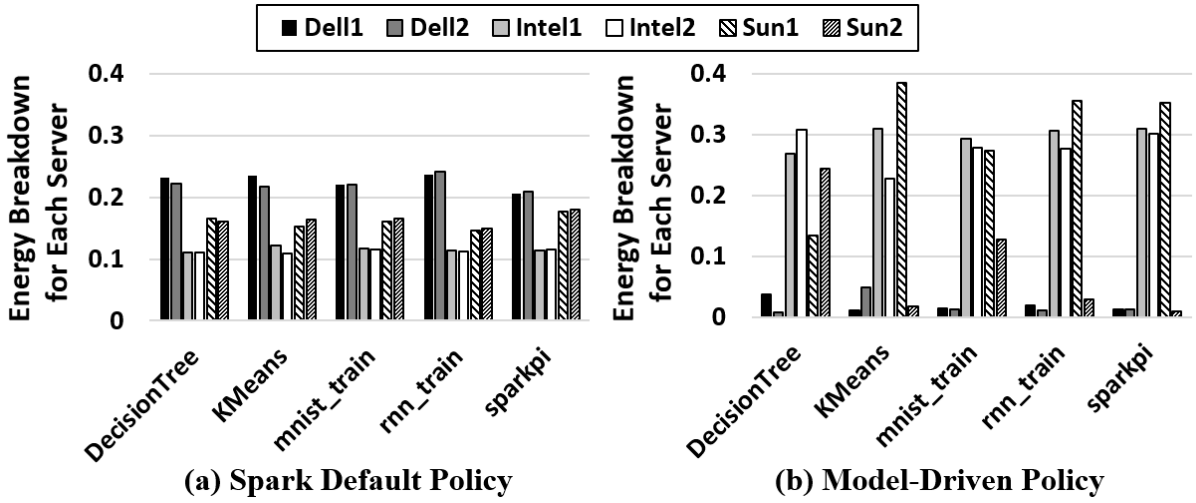


Figure 2.21: Energy breakdown comparison between Spark default and model-driven policy

executes the similar amount of tasks, since it randomly distributes them to all machines.

2.9.2 Energy Cost Reduction

In this evaluation, to understand how the policy balances Spark tasks, we vary the energy costs of the two geographically distributed clusters, the cluster 1 (C1) and cluster 2 (C2). We define the price ratio, $F = \$B_{C2}/\$B_{C1}$, where $\$B_{C1}$ and $\$B_{C2}$ are the energy cost (price per Wh) for each cluster, respectively. We compare the estimated energy bill of the model-based policy with the Spark default policy for different F values. For the experiments, the number of paralleled tasks (ω) is set to 3, i.e., the half of the servers can be utilized.

Figure 2.22 shows the summary of evaluation for energy cost reduction. The price-aware policy successfully allocates the jobs between the two clusters by considering the local price differences. Our estimate shows energy savings of 16.8% (F=0.5) and 12.5% (F=2), i.e., the energy price of one cluster is two times more expensive than of the other cluster. Figure 2.23 shows the breakdown of the server energy for different price ratios. The results show that it allocates more tasks to the cluster that has lower energy prices. For example, with a high F value, the energy price of C1 is much cheaper than the C2, resulting in allocating more tasks to C1.

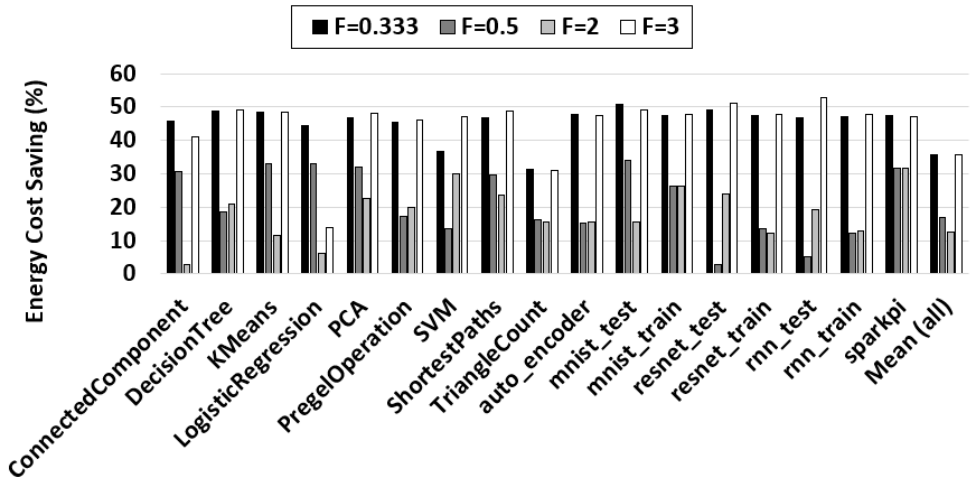


Figure 2.22: Summary of cluster-level energy cost reduction

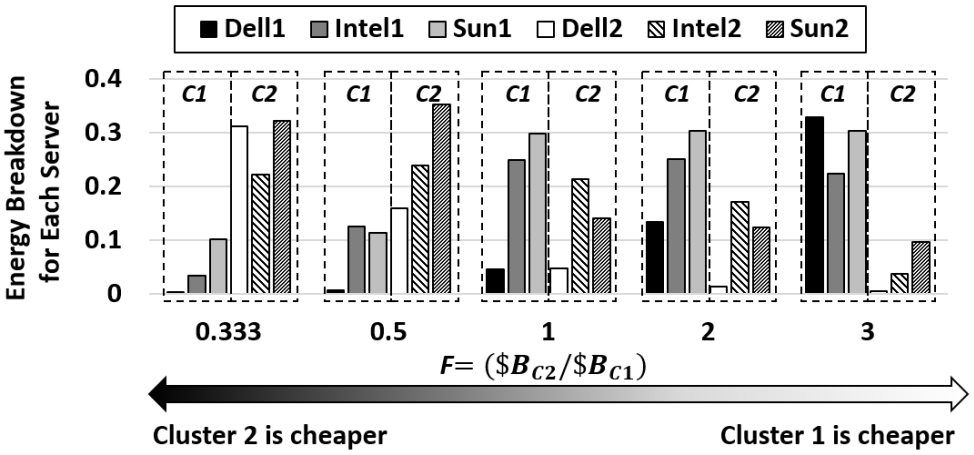


Figure 2.23: Energy breakdown over different price ratios between clusters

2.10 Conclusion

In this chapter, we propose P^4, which characterizes the diverse workload for heterogeneous computing ecosystems and accurately predicts power and performance across different CPU architectures and computing platform configurations. Our technique automatically selects the event counters strongly related to the power consumption and extracts application phases which represent the groups of similar system usage behavior without a priori knowledge. Then, it automatically trains neural networks to predict power and performance across different platforms

at runtime with negligible overhead. In our evaluation conducted on four heterogeneous computing platforms, we showed that our framework successfully recognizes the distinct application power states, and accurately predicts power consumption with less than 7.2% of error for all diverse configuration changes, including frequency levels, platforms, and CPU architectures. Based on the prediction technique, we also propose a predictive management technique which performs intelligent task allocation for ML workloads. Our evaluation results show that we can save energy and cost by 16%. In the next chapter, we present how we can further advance the efficiency of learning by designing a new class of ML algorithms.

This chapter contains material from "P4: Phase-Based Power/Performance Prediction of Heterogeneous Systems via Neural Networks", by Yeseong Kim, Pietro Mercati, Ankit More, Emily Shriver, and Tajana S. Rosing, which appears in International Conference on Computer-Aided Design, November 2017. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Hyperdimensional Computing for Efficient Learning in IoT Systems

3.1 Introduction

In the previous chapter, we presented how we can improve the efficiency of the state-of-the-art learning algorithms by intelligently predicting their resource usage and carefully allocating their tasks. Our predictive management framework primarily aims to distribute the tasks to clusters of powerful machines since the focus is on the allocation of computationally-expensive ML algorithms. However, the emergence of the IoT raises several other issues. The amount of data created by billions of distributed devices adds a significant computation burden to the centralized cloud. In addition, sending the sensitive user information may pose privacy and security concerns. An alternative solution is to run these tasks in a more localized way, e.g., on the IoT gateways at the edge [9, 75]. The local IoT devices typically have less computing resources than the cloud servers and run on low-power processors, such as ARM or Intel Atom. Today, ML algorithms are too complex to be trained on IoT devices [76, 77, 78]. We need a new ML technique that can be efficiently processed even on the embedded devices.

To achieve this goal, we have developed a new methodology which can efficiently perform learning tasks based on hyperdimensional (HD) computing. HD computing is recently developed as an alternative computing method inspired by the human brain [79]. It represents the brain's memory using data encoded into vectors of large dimensionality, called *hypervectors*. Earlier works show that HD computing can offer high efficiency for many classification tasks, e.g., voice recognition [80] and language identification [81]. HD computing is in particular suitable for sensor-based classification tasks like human activity recognition in IoT devices since it is robust against most hardware failure mechanisms and thrives on noisy and incomplete data that the IoT sensors often provide [82].

In this chapter, we describe how the HD computing can be applied to solve the classification problems, focusing on human activity recognition as an example of IoT applications. Human-aware system design has been widely investigated to offer high interactivity and enhanced

efficiency under limited device resources on IoT environment. Earlier researchers recognized that understanding human behavior is an important task to accomplish such goals. For example, diverse techniques exploited human activities and contexts as key control knobs of various system managements including mobile systems [83] and smart homes [84]. Human activity recognition such as motion detection is a key part of these techniques. Machine learning (ML) techniques are often used to automatically identify the activities from various information, where devices in the loop need to collect the data using sensors, e.g., accelerometers and GPS.

Our approach encodes the collected sensor samples with hypervectors, and combines the samples for each class into a single hypervector using robust algebra in HD space. Once the modeling is completed, we identify the human activity class for a newly observed data encoded with a hypervector. To this end, we match the most similar hypervector in the model to the sample. We design different variants of the HD computing-based classification method for higher efficiency and classification accuracy. In this chapter, we present two key approaches, *hypervector retraining* and *hypervector binarization*. The hypervector retraining refines the models to achieve higher classification accuracy. Unlike previous work [85], during the retraining step, we exploit non-binarized hypervectors to get higher accuracy. The hypervector binarization then converts the trained hypervectors back to hypervectors of bitstreams, making the HD computation more suitable for less-powerful IoT devices.

In our evaluation, we compare our approach with the state-of-the-art ML solutions. Our experimental results show that the proposed method can provide high accuracy and computing efficiency for popular human activity recognition problems. For example, as compared to the neural networks-based modeling [63], the HD-computing method is 486x faster when running on x86 processor. In addition, our design improves the performance of HD model-based inference tasks by up to 7x on a low-power ARM processor as compared to the deep learning model, while providing comparable classification accuracy.

3.2 Related Work

The hyperdimensional (HD) computing was first introduced in the field of neuroscience [79]. Prior researchers recognized that HD computing is effective for pattern-based cognitive tasks, and showed diverse applications, such as language recognition [81], text classification [86], the prediction from multimodal sensor fusion [87, 88], and speech recognition [80]. The work in [89] showed that bio signal sensory data can be represented with hyperdimensional data. Some work have also presented that HD tasks can be efficiently performed with diverse computing devices. For example, the hardware accelerator design has been proposed to efficiently compute binarized hypervectors. Some works also presented new memory architectures that perform HD operations inside memory arrays [90, 82]. Digital circuits for HD computing have been also designed, e.g., computation of Hamming distance distance search [82].

In this chapter, we focus on an example problem: "how the human activity recognition problem can be effectively mapped using HD computing." In addition, we show how the HD computing can be further optimized for IoT devices. Prior researchers have been investigated to understand and identify human activities and contexts. For example, the work in [91] showed a monitoring framework for human activity recognition which collects data from inertial measurement units (IMU). Some works have shown that daily activities can be captured by the sensors equipped in smartphone systems [92, 93]. Another line of research has focused on how to exploit the human activity and context information for diverse problems. For example, prior research has shown that understanding user's behavior and exploiting the behavioral characteristics can be used to improve system efficiency. In this context, earlier work proposed diverse system optimization techniques by identifying user behaviors and interactions for mobile systems [83] and smart homes [84]. Prior work often utilized ML techniques to identify the activities, while relying on computing capability of clouds through offloading, e.g., [94]. However, due to the massive data stream created in the IoT systems, more light-weight alternatives are considered as

a key requirement in the system design.

3.3 HD Computing Primitives

In this section, we discuss the primitives for HD computing.

Data type: Unlike conventional computing methods, the basic data type of the HD computing is the hypervector which often has many elements, e.g., more than one thousand. We denote the dimensionality of the hypervector using D. For example, collected data are converted to hypervectors for future HD procedures, e.g., classification tasks. In the earlier HD work, e.g., [80], each element of hypervector is assumed to be a bit. In contrast, since the hypervector containing numbers may include diverse information, some recent HD applications choose this data type to implement [95]. We call these two different types as *binary* hypervector and *non-binary* hypervector. An element of a binary hypervector can be either 0 or 1. For the non-binary hypervector, the elements can have any real number.

Property of hypervectors: An important characteristic used in HD computing is the orthogonality of hypervectors. Let us assume that there are two hypervectors, A and B. The non-binary hypervectors are defined to be orthogonal if the cosine similarity of A and B is zero. For binary hypervectors, we can define the orthogonality by mapping the hypervector element of 0 to -1. Since a hypervector has a large number of elements, we can easily find many pairs of two orthogonal hypervectors by randomly selecting their elements. For example, let us assume that we randomly choose elements of two non-binary hypervectors, A and B, among -1 and 1. In the cosine similarity computation, the element-wise multiplication make each element to either -1 or 1 with 50% chance, and the summation of all elements are very close to zero, i.e., near orthogonal. In contrast, if two hypervectors are computed somehow to be similar, the cosine similarity has a high value.

HD arithmetic operations: HD arithmetic operations enable to associate multiple hypervectors. In this chapter, we utilize three major operations.

- Binding: Two hypervectors A and B are combined into a hypervector. We denote this operation with $A \times B$. For the binary hypervectors, the element-wise XORing accomplishes this procedure; the element-wise multiplication is used for non-binary hypervectors. The binding operation preserves orthogonality of hypervectors. For example, when we have three hypervectors randomly created, say X, Y, and Z, the hypervector X is still near-orthogonal to the binding of the rests, $Y \times Z$.

- Bundling: This operation is denoted with the $+$ symbol. The component-wise addition implements the bundling for non-binary hypervectors. Since the bundling for two binary hypervector yields a non-binary hypervector, a majority function is applied afterward. For example, when n binary hypervectors are bundled, we first apply the elements-wise addition, and make each element whose value is greater than $n/2$ to 0; 1 for the other case. We denote this operation by $[A_0 + A_1 + \cdots + A_n]$. The bundling operation preserves the similarity with the combined hypervectors. For example, for two hypervectors A and B, the cosine similarity between A and $A + B$ is $cos(\pi/4)$, i.e., greater than zero.

- Detaching: This is a counter operation of the bundling for non-binary hypervectors. The component-wise subtraction implements this operation, and we denote it using the $-$ symbol. This makes the cosine similarity between two operand hypervectors either smaller or negative.

Associative search: The binary and non-binary hypervectors respectively use Hamming distance and cosine distance as their distance metrics. For simplicity, we denote the distance metric, which is appropriate for each case, by $\delta(A,B)$. When we have multiple hypervectors, the associative search is used to find the most similar hypervectors using the distance metric.

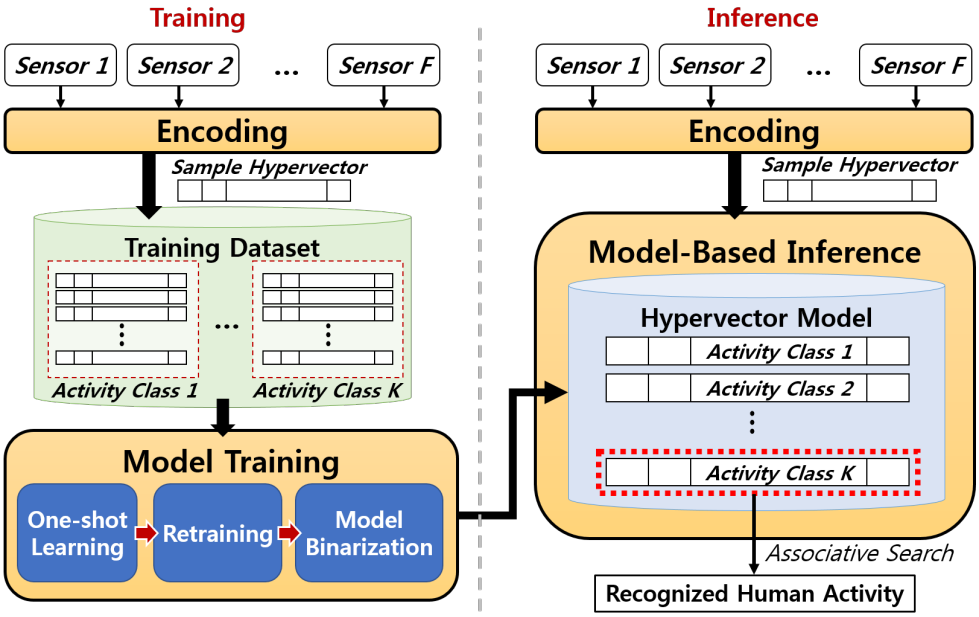


Figure 3.1: Overview of HD-Based Classification (Example: Human Activity Recognition)

For example, when we have m hypervectors, $H_1, \cdots, H_m$, the associative search for another hypervector A looks for a hypervector, H_i, whose $\delta(H_i, A)$ is the highest.

3.4 HD-Based Classification

3.4.1 Design Overview

Figure 3.1 describes our design that performs classification tasks, such as human activity recognition, based on HD computing. We collect multiple raw data from the external sensors in IoT devices, e.g., IMUs of wireless embedded devices and accelerometers in smartphones. Instead of using real numbers, we convert each collected sample, which includes multiple measurements, to a hypervector. We call this step by *encoding*. With the encoded hypervectors and its original activity (label), e.g., walking, running, and standing, we train the hypervector model. To classify K classes, the trained model includes K hypervectors for each class. The training procedure consists of three parts, one-shot learning, retraining, and model binarization. In the one-shot learning,

our design reads and process the hypervectors for each sample one by one. Then, the retraining refines the hypervector models considering the samples again with multiple iterations. In the next step, we update the model to the binarized hypervectors for performance improvements. With the trained model, we perform the inference of the class. The goal of the inference procedure is to classify a collected sample with an unknown label into an activity class. Our design accomplishes the inference by performing the associative search with the model hypervectors.

3.4.2 Sensor Data Encoding

To enable HD computing, we encode the collected raw data to hypervectors. Let us assume that a sample collected at a time includes F values, i.e., $S = \langle v_1, \cdots, v_F \rangle$, where each v_i is different raw values that each sensor measures. To find the patterns of sample hypervectors for each human activity, the encoding procedure considers the impact of i) the value for each sensor measurement and ii) differences of all the sensors in the system.

The first step of the encoding is to convert a measurement value, v_i, into a hypervector. As discussed in the background section, the similarity between two hypervectors, A and B, is determined with a metric, i.e., $\delta(A,B)$. Thus, we encode each value so that the corresponding hypervector keeps the relative difference across the measurement values of different samples under the distance metric. To this end, we utilize the measurement range of each sensor. For example, if a sensor produces a value in a range of $[V_{min}, V_{max}]$, the minimum and maximum values correspond to two hypervectors, L_{min} and L_{max}, where L_{min} and L_{max} are orthogonal to each other.

We represent any measurement value using the two hypervectors. L_{min} with D dimension is first created by randomly choosing its elements. Using the L_{min}, we create another hypervector, say L_1, by flipping $D/2Q$ elements, where Q is a configurable value. We repeat this procedure by Q times to decide $L_1, L_2, \cdots, L_Q$, e.g., flipping elements of L_1 creates L_2. Note that L_Q is orthogonal to L_{min}, thus $L_Q = L_{max}$. We call these created hypervectors as *level hypervectors*. A

level hypervector corresponds with each measurement value by considering the relative difference of the measurement values. To this end, where the measurement range is quantized to Q levels, and each quantized subrange is mapped to a level hypervector.

In the second step of the encoding, we combine different sensor values of a sample to represent it with a single hypervector. To distinguish different sensors in the hypervector representation, we utilize another set of hypervectors, $B_1, \cdots, B_F$, called *base hypervectors*, whose elements are randomly chosen for the orthogonality. Let assume that each v_i value corresponds to a level hypervector, L_i. The encoded hypervector for the sample is computed by:

$$H = L_1 \times B_1 + \cdots + L_F \times B_F.$$

Since the B_i hypervectors are orthogonal, even though we use the same set of the level hypervectors for different sensors, our training step still distinguishes the impact of different sensors within the encoded hypervector. All of the random hypervectors, i.e., L_{min} and B_i, are required to be created only once and exploited for the entire recognition procedure. Note that the elements of the encoded hypervectors, say *sample hypervectors*, are 0 or 1 if using the binary hypervectors; -1 or 1 for the non-binary hypervector case.

3.4.3 Model Training

In this procedure, our design trains the model by combining the sample hypervectors. The goal is to learn the patterns of sensor values which exist within a class. Let assume that the training dataset includes N samples, and each sample is encoded with N hypervectors, $H_1, \cdots, H_N$. Each sample hypervector corresponds to an activity class, say c_i.

One-shot training: The first step of the training is to bundle the hypervectors for each class. We call this computation as *one-shot* training. For example, let us assume that there are l hypervectors, $H_1, H_2, \cdots, H_l$, where all of them are included in the same class. The bundling

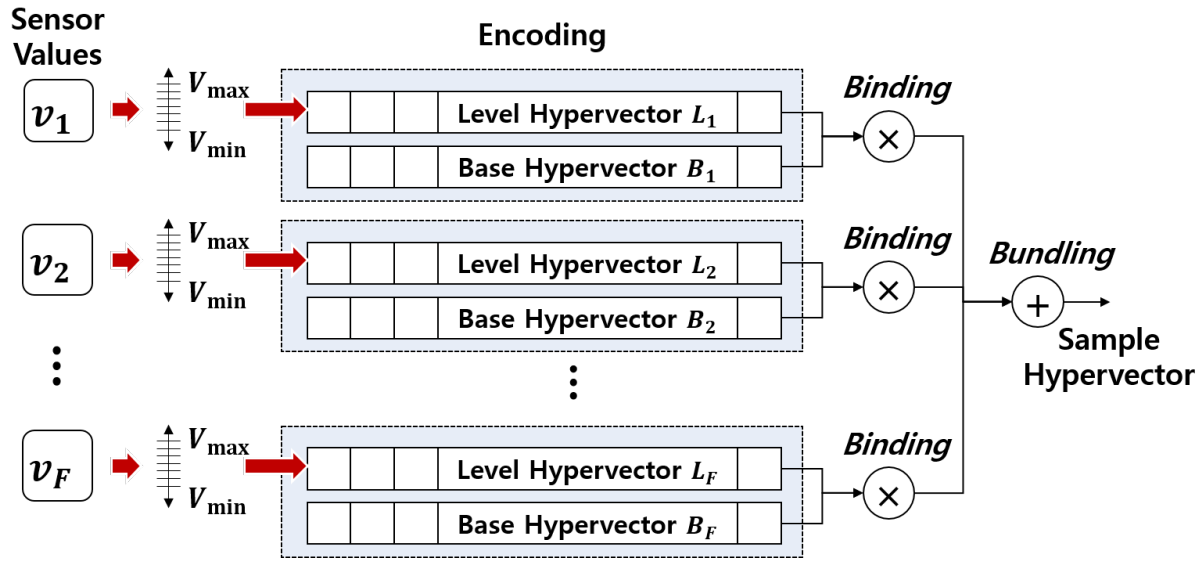


Figure 3.2: Encoding of Sensor Measurements

operation makes another hypervector, $M = H_1 + \cdots + H_l$. For example, let us assume that we have another hypervector H_{test}, which is very similar to H_1, by the distance metric. In this case, $\delta(M, H_{test})$ is likely to be a positive value. Furthermore, if H_{test} is similar to the majority of the hypervectors combined into M, $\delta(M, H_{test})$ yields a much higher value. Based on this observation, we create the one-shot model, say $M_1, \cdots, M_K$, by bundling all sample hypervectors included in each activity of K classes.

Retraining: An issue of the one-shot model is that, although the bundled hypervectors captures the major similarity *within* each class, it does not understand hypervector differences *across* classes. In addition, bundling a large number of hypervectors may degrade classification quality when a large variety of patterns exists in each class. Thus, we refine the model to i) better identify the discrepancy between different classes and ii) recognize the common pattern existing in each class.

Algorithm 1 illustrates our retraining procedure to reduce the misclassification rate of the activity recognition. From the one-shot model, our design verifies the classification accuracy for each sample using the associative search. If a sample is wrongly classified, we modify two

Algorithm 1: Pseudo code of retraining precedure

```
t ← 0
while t < # of Iterations do
    t ← t + 1
    for each sample hypervector, H_i do
        ρ ← associative search for H_i in the model
        if ρ ≠ c_i then
            M_{c_i} ← M_{c_i} + H_i
            M_ρ ← M_ρ − H_i
        end
    end
end
```

model hypervectors, i.e., the hypervector of the target class and the other hypervector of the misclassified class. We first bundle the sample hypervector once more to the correct class so that the model hypervector converges faster to the misclassified sample. The second task is detaching the hypervector from the wrong class to enlarge the difference between the two model hypervectors. We repeat this updating process multiple times for the training dataset, and the accuracy converges with sufficient iterations.

Model Binarization: Since our model retraining algorithm exploits the element-wise addition and subtraction in the bundling and detaching operations, it consequently creates non-binary hypervectors as the model. Even though it makes the model more accurate, the model size and computation costs of the inference also increase. Since many devices in IoT environments which run the activity recognition is less-powerful, we optimize the model by converting the model to the binary hypervectors. We update the model depending on the sign of each hypervector element, i.e., choosing 1 if the element value is positive; 0 for the negative value.

3.4.4 Model-Based Inference

Once the model is trained, it is ready to process the inference step for samples whose labels are unknown. We first encode the values using the level and base hypervectors used in

training step. Then, our design finds which model hypervectors is the most similar to the given sample hypervector using the associative search. Note that, in the associative search, we use different distance metrics based on the data type of the model. In general, the non-binarized model provides better accuracy. In contrast, the binarized model processes the inference in a more efficient way, since the Hamming distance can be computed with bitwise XOR operations for the smaller model, unlike the element-wise integer additions for the cosine distance computation.

3.5 Evaluation

3.5.1 Experimental Setup

To evaluate how the proposed design works on the heterogeneous IoT environment, we utilize two different devices running on 2.8 GHz Intel Core i7 (x86) and 1.4 GHz ARM Cortex-A53 (ARM) processors. For both cases, we execute the same code implemented with Python 2.7 and Numpy which uses C++ backend. We compare our approach with the state-of-the-art deep neural network models (DNN) implemented using Google TensorFlow. Since our design can create binarized hypervector models, for fair comparison, we also evaluate the binarized neural network (BNN) models. The neural network models have three hidden layers of 512 neurons, and DNN and BNN models are trained with ADAM optimizer for 10 and 100 epochs, respectively, so that the accuracy converges. For the efficiency comparison, we measure the execution time of the training and testing procedures.

We evaluate our approach using three practical datasets as follows.

UCIHAR: This dataset includes the sensor measurements for accelerometers and gyroscopes of a smartphone, which are measured on the waist of users. The goal is to classify twelve activity classes, e.g., walking, walking up/downstairs, sitting and standing.

PAMAP2: The dataset contains data measured from three IMUs located at the wrist, chest, and ankle of users with a heart rate monitor. The goal is to classify five basic activities,

Table 3.1: Evaluated Dataset (F: the number of features, K: the number of activity classes, N_{train}: the number of samples in the training dataset, N_{test}: the number of samples in the testing dataset)

Name	Data Size	F	K	N_{train}	N_{test}
UCIHAR [92]	10MB	561	12	6213	1554
PAMAP2 [91]	240MB	75	5	611142	101582
EXTRA [93]	140MB	225	4	146869	16343

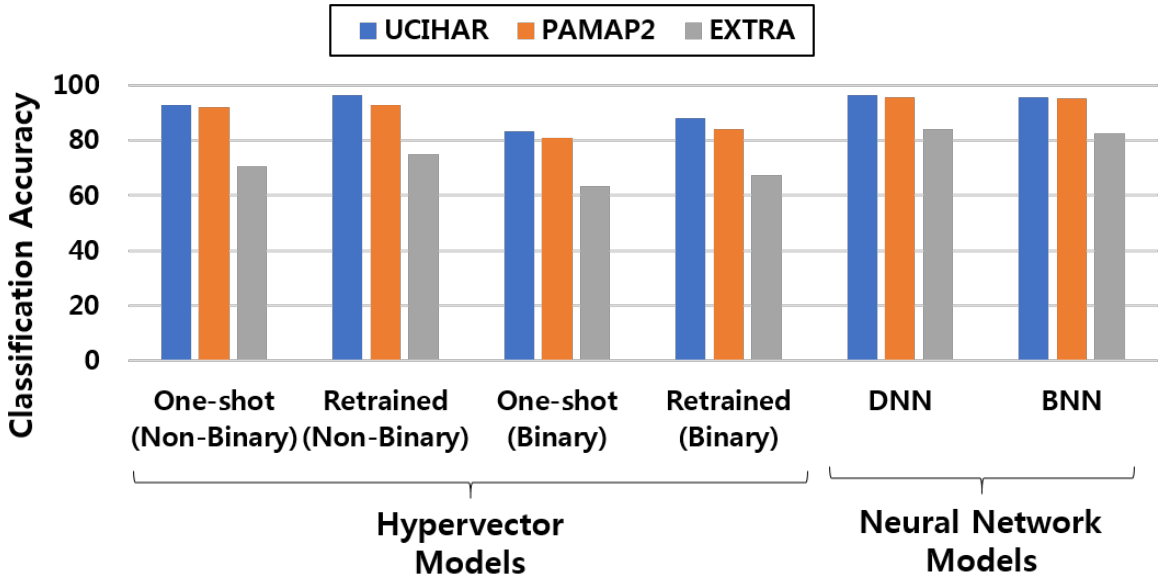


Figure 3.3: Accuracy Comparison for Different Modeling Methods

e.g., walking and sitting. We exploit the feature extraction method suggested by the author.

EXTRA: The dataset has measurements of heterogeneous sensors from smartphones and smartwatches. We choose to classify the activity labels for phone locations, e.g., whether it is located on the table, in the pocket, bag, and hand. Note that the activities are related to diverse device control problems, e.g., thermal management of mobile devices [96].

Table 3.1 summarizes the dataset sizes. In our evaluation, we set the quantization level to 8, the retraining iterations to 20, and the dimension of hypervectors to 1000, since there is no accuracy gain with larger values.

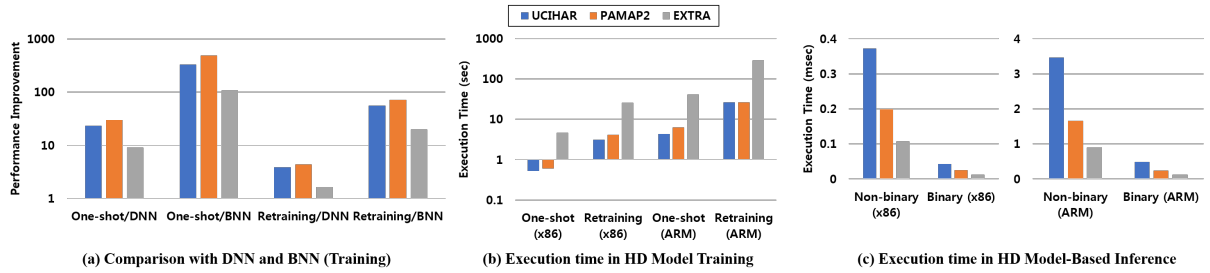


Figure 3.4: Efficiency Comparison for Training and Inference

3.5.2 Classification Accuracy

Figure 3.3 shows the comparison results of the accuracy for different modeling methods. The results show that the proposed retraining method improves the classification accuracy. For example, when using the non-binary hypervector models, the accuracy improvement is 3% on average. We observe higher accuracy improvements for the binarized hypervector models by 4% on average. Throughout the retraining procedure, we train the HD model which have comparable accuracy to the DNN and BNN models. For example, for UCIHAR dataset, the accuracy difference between the non-binary model and DNN is only 0.2%. The accuracy difference between binary and non-binary models is 8% on average. In the next section, we evaluate how much performance can be improved by the model binarization.

3.5.3 Efficiency Comparison

Training Efficiency We evaluate the efficiency of different modeling methods. Figure 3.4(a) shows the efficiency comparison of our design with the state-of-the-art DNN and BNN model. The results are reported for the non-binarized models, since the overhead of the model binarization is negligible.[1] In this comparison, the HD modeling and the neural network training were both executed on x86 processor. The results show that the proposed method presents higher performance efficiency as compared to the neural network training. For example, for UCIHAR

[1]The model binarization requires to update the trained hypervectors only once after all the retraining procedure.

dataset, training the HD model with the retraining procedure is 4x and 56x faster than the DNN and BNN models, respectively. Note that the accuracy difference between the two model is only 0.2% as presented in the previous section.

In addition, when a small amount of accuracy loss is acceptable, our design can also train the model without retraining. In that case, we observe the speedup up to 486x compared to the BNN approach.

Figure 3.4(b) compares the execution time of the training procedure on the two different processors. The results suggest that the proposed design can efficiently train the hypervector model even on the low-power processor. For example, for PAMAP2 dataset, the training time including the retraining only takes 26 seconds on the ARM processor. To train the one-shot model, it only takes 4 seconds. Thus, we conclude that the proposed design may efficiently process the activity recognition tasks in the IoT systems, since many IoT devices in the loop is expected to run on low-power processors with resource budgets.

Inference Efficiency With the trained model, our design performs the inference tasks for each collected data. Figure 3.4(c) shows how much the execution time takes to process the inference procedure for each sample. In this evaluation, we compare the non-binary model to the binary model. The result shows that the model binarization significantly improves the inference procedure. The speedup is 8.4x and 7.1x for the x86 and ARM case, respectively.

For the ARM processor case, the inference based on the non-binarized model takes 2 ms on average, while the binarized model only takes 0.28 ms. In IoT systems, the sensors are often equipped with the same device running on these low-power processors. Thus, when a small amount of accuracy loss is acceptable, the binarized model is more preferable, e.g., serving real-time needs for the activity recognition.

3.6 Conclusion

In this chapter, we present a new algorithm which utilizes HD computing to enable efficient learning on low-power embedded devices. We also show optimization techniques that improve the accuracy of the HD-based classification and performance efficiency in the inference procedure. In our evaluation conducted with practical human activity recognition tasks, the proposed design is 486x faster for training, compared to the neural network models [63]. The algorithm proposed in this chapter runs the learning on a single IoT device. In the next chapter, we discuss how to extend and deploy the proposed learning method into the hierarchy of multiple IoT nodes.

This chapter contains material from "Efficient Human Activity Recognition Using Hyperdimensional Computing", by Yeseong Kim, Mohsen Imani, and Tajana S. Rosing, which appears in IEEE Conference on Internet of Things, October 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Collaborative Learning with Hyperdimensional Computing

4.1 Introduction

In the previous chapter, we proposed an HD computing-based learning which encodes the data to the hypervectors and performs the rest of the learning procedure running on a single node. In practice, the learning in many IoT systems is done with data that is held by a large number of devices. To analyze the collected data using machine learning algorithms, IoT systems typically send the data to a centralized location, e.g., local servers, cloudlets and data centers [76, 77, 78]. However, sending the data not only consumes lots of bandwidth, battery power, but is also undesirable due to privacy and security concerns [97, 98, 99, 100]. Many machine learning models usually require unencrypted data original images, to train models and perform inference. When offloading the computation tasks, sensitive information is exposed to the *untrustworthy cloud system* which is susceptible to internal and external attacks [101, 102] . The users may also be unwilling to share the original data with the cloud and other users [103, 104, 105, 106].

An existing strategy applicable to this scenario is to use Homomorphic Encryption (HE). HE enables encrypting the raw data and allowing certain operations to be performed directly on the ciphertext without decryption [16]. However, this approach significantly increases computation burden. For example, in our evaluation, with Microsoft SEAL, a state-of-the-art homomorphic encryption library [17], it takes around 14 days to encrypt all of the 28x28 pixel images in the entire MNIST dataset, and increases the data size 28 times. More recently, Google presented a protocol for secure aggregation of high-dimensional data that can be used in *federated* learning model [107]. This approach trains Deep Neural Networks (DNN) when data is distributed over different users. In this technique, the users' devices run the DNN training task locally to update the global model. However, IoT edge devices often do not have enough computation resources to perform such complex DNN training.

In this chapter, we design SecureHD, an efficient, scalable, and secure collaborative learning for the distributed computing in the IoT hierarchy. HD computing does not require

complete knowledge for the original data that the conventional learning algorithms need – it runs with a mapping function that encodes data to a high-dimensional space. The original data cannot be reconstructed from the mapped data without knowing the mapping function, resulting in secure computation.

We address several technical challenges to enable HD-based *trustworthy, collaborative* learning. To map the original data into hypervectors, it uses a set of randomly-generated *base hypervectors* as described in Chapter 3. Since the base hypervectors can be used to estimate the original data, every user has to have different base hypervectors to ensure the confidentiality of the data. However, in this case, the HD computation cannot be performed with the data provided by different users.

SecureHD fills the gap between the existing HD computing and trustworthy, collaborative learning by providing the following contributions:

i) We design a novel *secure collaborative learning protocol* that securely generates and distributes public and secret keys. SecureHD utilizes Multi-Party Computation (MPC) techniques which are proven to be secure when each party is untrusted [108]. With the generated keys, the user data are not revealed to the cloud server, while the server can still learn a model based on the data encoded by users. Since MPC is an expensive protocol, we carefully optimize it by replacing a part of tasks with two-party computation. In addition, our design leverages MPC only for a one-time key generation operation. The rest of the operations such as encoding, decoding, and learning are performed without using MPC.

ii) We propose a new encoding method that maps the original data with the secret key assigned to each user. Our encoding method significantly improves classification accuracy as compared to the state-of-the-art HD work [109, 82]. Unlike existing HD encoding functions, the proposed method encodes both the data and the metadata, e.g., data types and color depths, in a recover-friendly manner. Since the secret key of each user is not disclosed to anyone, although one may know encoded data of other users, they cannot be decoded.

iii) SecureHD provides a robust decoding method for the authorized user who has the secret key. We show that the cosine similarity metric widely used in HD computing is not suitable to recover the original data. We propose a new decoding method which recovers the encoded data in a lossless manner through an iterative procedure.

iv) We present scalable HD-based classification methods for many practical learning problems which need the collaboration of many users, e.g., human activity and face image recognition. We propose two collaborative learning approaches, cloud-centric learning for the case that end-node devices do not have enough computing capability, and edge-based learning that all the user devices participate in secure distributed learning.

v) We also show a hardware accelerator design that significantly minimizes the costs paid for security. This enables secure HD computing on less-powerful edge devices, e.g., gateways, which are responsible for data encryption/deception.

We design and implement the proposed SecureHD framework on diverse computing devices in IoT systems, including a gateway-level device, a high-performance system, and our proposed hardware accelerator. In our evaluations, we show that the proposed framework can perform the encoding and decoding tasks 145.6× and 6.8× faster than a state-of-the-art homomorphic encryption library when both are running on the Intel i7-8700K. The hardware accelerator further improves the performance efficiency by 35.5× and 20.4× as compared to the CPU-based encoding and decoding of SecureHD. In addition, our classification method presents high accuracy and scalability for diverse practical problems. It successfully performs learning tasks with 95% average accuracy for six real-world workloads, ranging from datasets collected in a small IoT network, e.g., human activity recognition, to a large dataset which includes hundreds of thousands of images for the face recognition task. Our decoding method also provides high quality in the data recovery. For example, SecureHD can recover the encoded data in a lossless manner, where the size of the encoded data is 4 times smaller than the one encrypted by the state-of-the-art homomorphic encryption library [110].

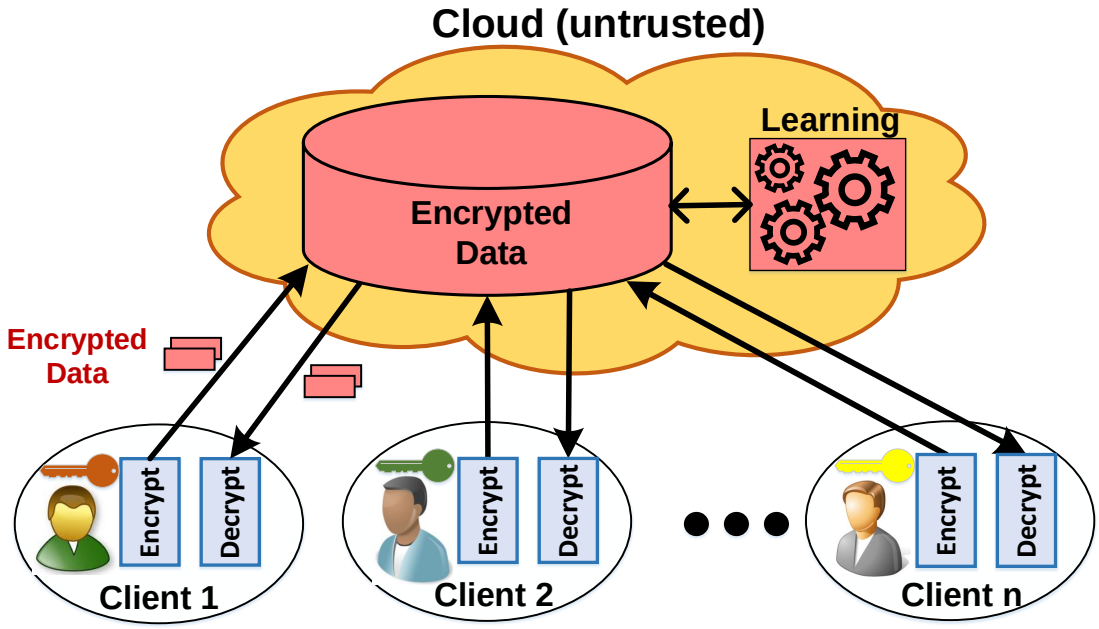


Figure 4.1: Motivational scenario

4.2 Motivational Scenario

Figure 4.1 shows the scenario that we focus in this chapter. The clients, e.g., user devices, send either their sensitive data or partially trained models in an encrypted form to the cloud. The cloud performs a learning task by collecting the encrypted information received from multiple clients. In our security model, we assume that a client cannot trust the cloud as well as other clients. When requested by the user, the cloud sends back the encrypted data to clients. The client then decrypts the data with its private key.

As an existing solution, homomorphic encryption enables processing on the encrypted version of data [16]. Figure 4.2 shows the execution time of a state-of-the-art homomorphic encryption library, Microsoft SEAL [17], for MNIST training dataset, which includes 60000 images of 28×28 pixels. We execute the library on two platforms that a client in IoT systems may use, a high-performance computer (Intel i7-8700K) and a Raspberry Pi 3 (ARM Cortex A53). The result shows that, even with the simple dataset of 47 MBytes, it takes significantly

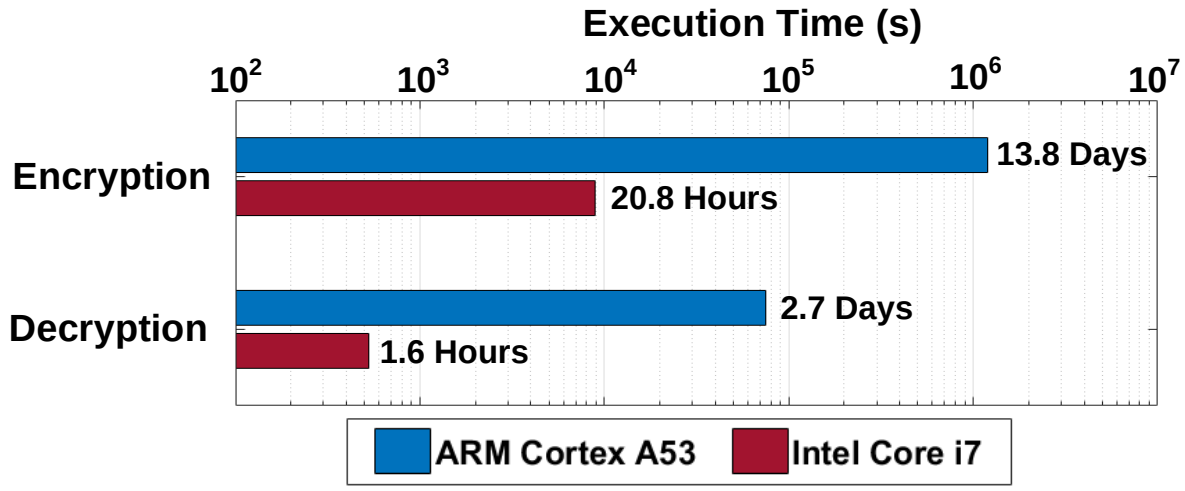


Figure 4.2: Execution time of homomorphic encryption and decryption over MNIST dataset

large execution time, e.g., more than 13 days on ARM to encrypt.

Another approach is to utilize secure Multi-Party Computation (MPC) techniques [111, 108]. In theory, any function, which can be represented as a Boolean circuit with inputs from multiple parties, can be evaluated securely without disclosing each party's to anyone else. For example, by describing the machine learning algorithm as a Boolean circuit with learning data as inputs to the circuit, one can securely learn the model. However, such solutions are very costly in practice and are computation and communication intensive. In SecureHD, we only use MPC to securely generate and distribute users' private keys which is orders of magnitude less costly than performing the complete learning task using MPC. The key generation step is a one-time operation so the small cost associated with it is quickly amortized over time for future tasks.

4.3 Related Work

Privacy-preserving deep learning and classification has been an active research area in recent years [107, 112, 113, 114, 115, 116, 117]. Shokri and Shmatikov [113] have proposed a solution for collaborative deep learning where the training data is distributed among many parties.

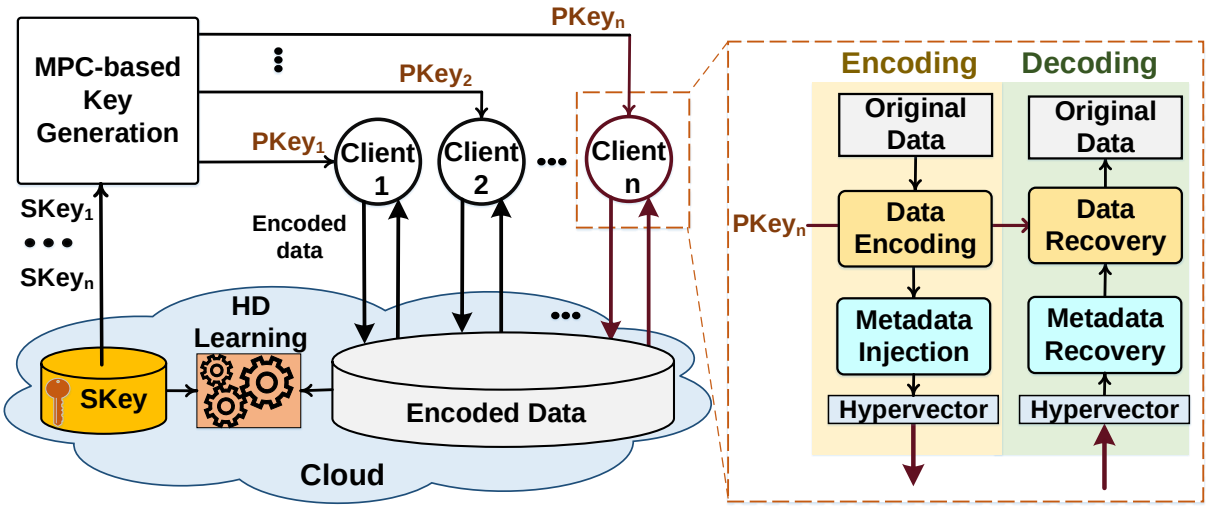


Figure 4.3: Overview of SecureHD

Each party locally trains her model and sends the parameter updates to the server. However, it has been shown that Generative Adversarial Networks (GANs) can be used to attack this method [118].

SecureML [115] is a framework for secure training of machine learning models. All of the computation of SecureML is performed by the two servers using MPC protocols, whereas, SecureHD only relies on the MPC protocol for secure key generation and distribution. Chameleon [112] is a privacy-preserving machine learning framework that utilizes different cryptographic protocols for different operations within the machine learning task. In contrast to SecureML and Chameleon, our solution does not require two non-colluding servers and only involves one server.

Google has also proposed a federated learning approach [107] for collaborative learning. In their approach, each client needs to learn the local model based on the private training data to update the central model in the cloud. However, our solution is more light-weight to be run on less-powerful IoT devices and also applicable to other cloud-oriented tasks, e.g., data storage services.

4.4 Secure Learning in HD Space

4.4.1 Security Model

In SecureHD, we consider the server and other clients to be untrusted. More precisely, we consider Honest-but-Curious (HbC) adversary model where each party, server or a client, is untrusted but follows the protocol. Both the server and other clients are not able to extract any information based on the data that they receive and send during the secure computation protocol. For the task of key generation and distribution, we utilize a secure MPC protocol which is proven to be secure in the HbC adversary model [108]. We also use two-party Yao's Garbled Circuits (GC) protocol which is also to be secure in the HbC adversary model as well [119]. The intermediate results are stored as additive unique shares of PKey by each client and the server.

4.4.2 Proposed Framework

In this section, we describe the proposed SecureHD framework which enables trustworthy, collaborate HD computing. Figure 4.3 illustrates the overview of SecureHD. The first step is to create different keys for each user and cloud-based on an MPC protocol. To perform a HD learning task, the data are encoded with a set of base hypervectors. The MPC protocol creates the base hypervectors for the learning application, called global keys (*GKeys*). Instead of sharing the original GKeys with clients, the server distributes permutations of each GKey, i.e., a hypervector whose dimensions are randomly shuffled. Since each user has different permutations of GKeys, called personal keys (*PKeys*), no one can decode encoded data of others. The cloud has dimension indexes used in the GKey shuffling, called shuffling keys (*SKeys*). Since the cloud does not have the GKeys, it cannot decrypt the encoded data of clients. This MPC-based key generation runs only once.

After the key generation, each client can encode their data with its PKeys. SecureHD securely injects a small amount of information into the encoded data. We exploit this technique to

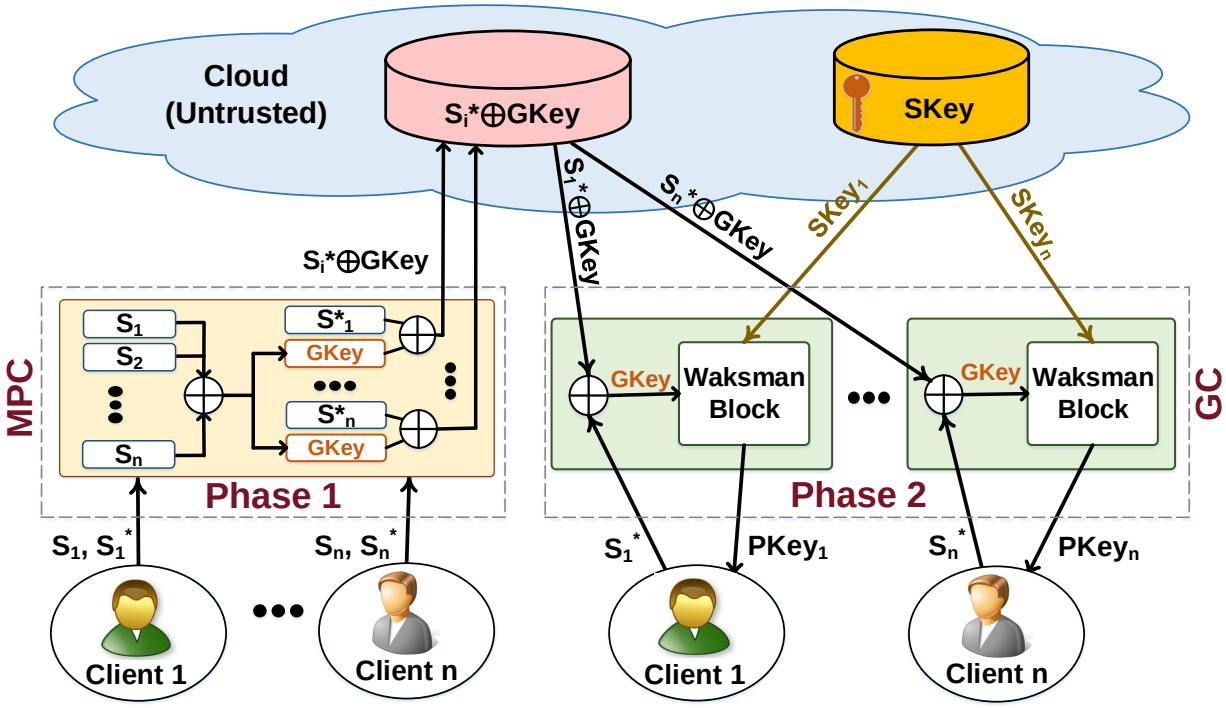


Figure 4.4: MPC-based key generation

store the metadata, e.g., data types, which are important to recover the entire original data. Once the encoded data is sent to the cloud, the cloud reshuffles the encoded data with the *SKeys* for the client. This allows the cloud to perform the learning task with no need for accessing GKeys and PKeys. With the SecureHD framework, the client can also decode the data from the encoded hypervectors. For example, once a client fetches the encoded data from the cloud storage service, it can exploit the framework to recover the original data using its own PKeys. Each client may also utilize the specialized hardware to accelerate both the encoding and decoding procedures.

4.4.3 Secure Key Generation and Distribution

Figure 4.4 illustrates how our protocol securely create the key hypervectors. The protocol runs two phases: *Phase 1* that all clients and the cloud participate, and *Phase 2* that two parties, a single client and cloud, participate. Recall that in order for the cloud server to be able to learn the model, all have to be projected based on the same base hypervectors. Given the base hypervector

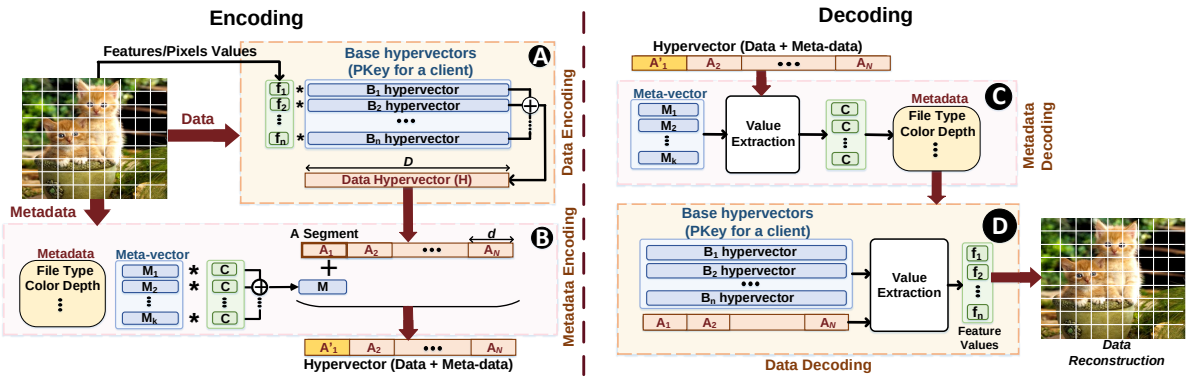


Figure 4.5: Illustration of SecureHD encoding and decoding procedures

and the encoded result, one can reconstruct the plaintext data. Therefore, all clients have to use the same key without anyone having access to the base hypervectors. We realize these two constraints at the same time with a novel hybrid secure computation solution.

In the first phase, we generate the base hypervectors, which we denote by *GKey*. The main idea is that the base hypervectors are generated collaboratively inside the secure Multi-Party Computation (MPC) protocol. At the beginning of the first phase, each party i inputs two sets of random strings called S_i and S_i^*. Each stream length is D, where D is the dimension size of a hypervector. The MPC protocol computes element-wise XOR ($\oplus$) of all the provided bitstreams, and the substream of D elements represent the global base hypervector, i.e., GKey. Then, it performs XOR for the GKeys again with S_i^* provided by each client. At the end of the first MPC protocol phase, the cloud receives $S_i^* \oplus GKey$ corresponding to each user i and stores these secret keys. Note that since S_i and S_i^* are inputs from each user to the MPC protocol, it is not revealed to any other party during the joint computation. It can be seen that the server has a unique XOR-share of the global key *GKey* for each user. This, in turn, enables the server and each party to continue their computation in a point-to-point manner without involving other parties during the second phase.

Our approach has a strong property that even if all other clients are dishonest and provide zero vectors as their share to generate the Gkey, the security of our system is not hindered. The

reason is that the Gkey is generated with XOR of S_i for *all* clients. That is, if one generates its seed randomly, the global key will have a uniform random distribution. In addition, the server only receives an XOR-share of the global key. The XOR-sharing technique is equivalent to One-Time Pad encryption and is information-theoretic secure which is superior to the security against computationally-bounded adversaries in standard encryption schemes such as Advanced Encryption Standard (AES). We only use XOR gates in MPC which are considerably less costly than non-XOR gates [120].

In the second phase, the protocol distributes the secret key for each user. Each party engages in a two-party secure computation using the GC protocol. Server's inputs are $SKey_i$ and $S_i^* \oplus GKey$, while the client's input is S_i^*. The global key $GKey$ is securely reconstructed inside the GC protocol by XOR of the two shares: $GKey = S_i^* \oplus (S_i^* \oplus GKey)$. The global key is then shuffled based on the unique permutation bits held by the server ($SKey_i$). In order to avoid costly random accesses inside the GC protocol, we use the Waksman permutation network with $SKey_i$ being the permutation bits [121]. The shuffled global key is sent back to the user, and we perform a single rotational shift for the GKey to generate the next base hypervector. We repeat this n times where n is the required number of base hypervectors, e.g., the feature size. The permuted base hypervectors serve as user's personal keys, called $PKey$, for the projection. Once a user performs the projection with $PKey$, she can send the result to the server, and the server permutes back based on the $SKey_i$ for the learning process.

4.5 SecureHD Encoding and Decoding

Figure 4.5 shows how the SecureHD framework performs the encoding and decoding of a client with the generated PKeys. The example has been shown for an image input data with n pixel values, $\{f_1, \ldots, f_n\}$. Our design encodes each input data into a high-dimensional vector from the feature values (❹). It exploits the PKeys, i.e., a set of the base hypervectors for the client,

where 0 and 1 in the PKeys correspond to -1 and 1 to form a bipolar hypervector ($\{-1,+1\}^D$). We denote them by $PKeys = \{\mathbf{B}_1,\ldots,\mathbf{B}_n\}$. To store the metadata with negligible impact on the encoded hypervector, we devise a method which injects several metadata to small segments of an encoded hypervector. This method exploits another set of base vectors, $\{\mathbf{M}_1,\ldots,\mathbf{M}_k\}$ (Ⓑ). We call them as *metavector*. The encoded data are sent to the cloud to perform HD learning.

Once the encoded data is received from the cloud, SecureHD can also decode them back to the original domain. This is useful for other cloud services, e.g., cloud storage. This procedure starts with identifying the injected metadata (Ⓒ). Based on the injected metadata, it figures out the base hyperevectors that will be used in the decoding. Then, it reconstructs the original data from the decoded data (Ⓓ). The key of the data recovery procedure is the value extraction algorithm, which retrieves both metadata and data.

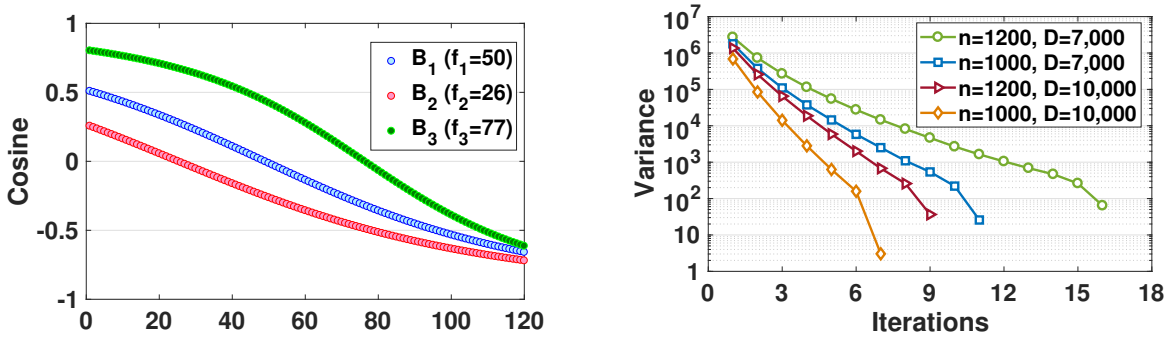
4.5.1 Encoding in HD Space

Data Encoding The first step of SecureHD is to encode input data into hypervector, where an original data point has n features. We associate each feature with a hypervector. The features can have discrete value (e.g., alphabets in the text), in which we perform a straight mapping to hypervectors, or they can have a continuous range, in which case the values can be quantized and then mapped similar to discrete features. Our goal is to encode each feature vector to a hypervector that has D dimensions, e.g. $D = 10,000$.

To differentiate each feature, we exploit a PKey for each feature value, i.e., $\{\mathbf{B}_1,\mathbf{B}_2,\ldots,\mathbf{B}_n\}$, where n is the feature size of an original data point. Since the PKeys are generated from the random bit streams, the similarity of different base hypervectors are nearly orthogonal [122]:

$$\delta(B_i,\, B_j) \simeq 0 \qquad (0 < i, j \leq n,\;\; i \neq j).$$

The orthogonality of feature hypervectors is ensured as long as the hypervector dimension, D, is



(a) (b)

Figure 4.6: Value extraction example

large enough compared to the number of features ($D >> n$) in the original data.

Different features are combined by multiplying feature values with the corresponding base hypervector, $\mathbf{B}_i \in \{-1, +1\}^D$ and adding them for all the features. For example, where f_i is a feature value, the following equation represents the encoded hypervector, $\mathbf{H}$:[1]

$$H = f_1 * \mathbf{B}_1 + f_2 * \mathbf{B}_2 + \ldots + f_n * \mathbf{B}_n.$$

If two original feature values are similar, their encoded hypervectors are also similar, thus providing the learning capability for the cloud without any knowledge for the PKeys. Please note that, with this encoding scheme, although an attacker intercepts sufficient hypervectors, the upper bound of the information leakage is the distribution of the data. It is because the hypervector does not preserve any information of the feature order, e.g., pixel positions in an image, and there are extremely large combinations of values in hypervector elements which exponentially grow as n increases. In the case that n is small, e.g., $n < 20$, we can simply add extra features drawn from a uniform random distribution, and it does not affect the data recovery accuracy and HD computation results.

Metadata Injection A client may receive an encoded hypervector where SecureHD processes multiple data types. In this case, to identify base hypervectors used in the prior

[1]The scalar multiplication, denoted by *, can make a hypervector that has integer elements, i.e., $\mathbf{H} \in \mathbb{N}^D$.

encoding, it needs to embed additional information of the data identifier and metadata, such as data type (e.g., image or text) and color depth. One naive way is to store this metadata as attached bits to the original hypervector. However, this does not keep the metadata secure.

To embed the additional metadata into hypervectors, we exploit the fact that HD computing is robust to small modification of hypervector elements. Let us consider a data hypervector as a concatenation of several partial vectors. For example, a single hypervector with the D dimension can be viewed as the concatenation of different d-dimensional vectors, $\mathbf{A}_1, \ldots, \mathbf{A}_N$:

$$\mathbf{H} = \mathbf{A}_1 \frown \mathbf{A}_2 \frown \cdots \frown \mathbf{A}_N$$

where $D = N \times d$, and each $\mathbf{A}_i$ vector is called as a *segment*. We inject the metadata in a minimal number of segments.

Figure 4.5 shows the concatenation of a hypervector to $N = 200$ segments with $d = 50$ dimensions. We first generate a random d dimensional vector with bipolar values, $\mathbf{M}_i$, i.e., *metavector*. A metavector corresponds to a metadata type. For example, $\mathbf{M}_1$ and $\mathbf{M}_2$ can correspond to the image and text types, while $\mathbf{M}_3$, $\mathbf{M}_4$, and $\mathbf{M}_5$ correspond to each color depth, e.g., 2-bit, 8-bit, and 32-bit. Our design injects each $\mathbf{M}_i$ into one of the segments in the data hypervector. We add the metavector multiple times to better distinguish it against the values already stored in the segment. For example, if we inject the metavector in the first segment, the following equation denotes the metadata injection procedure:

$$\mathbf{A}'_1 = \mathbf{A}_1 \; + \; C * \mathbf{M}_1 \; + \; C * \mathbf{M}_2 \; + \; \ldots \; + \; C * \mathbf{M}_k$$

where C is the number of injections for each metavector.

4.5.2 Decoding in HD Space

Value Extraction In many of today's applications, the clouds are used as a storage, so the clients should be able to recover the original data from encoded ones. The key component of the decoding procedure is a new data recovery method that extracts the feature values stored in the encoded hypervectors. Let us consider an example of $\mathbf{H} = f_1 * \mathbf{B}_1 + f_2 * \mathbf{B}_2 + f_3 * \mathbf{B}_3$, where $\mathbf{B}_i$ is a base hypervector with D dimensions and f_i is a feature value. The goal of the decoding procedure is to find a f_i for a given $\mathbf{B}_i$ and $\mathbf{H}$. A possible way is to exploit the cosine similarity metric, δ. For example, if we measure the cosine similarity of $\mathbf{H}$ and $\mathbf{B}_1$ hypervectors, $\delta(\mathbf{H}, \mathbf{B}_1)$, the higher δ value represents higher chance of the existence of $\mathbf{B}_1$ in $\mathbf{H}$. Thus, one method may iteratively subtracts one instance of $\mathbf{B}_1$ from $\mathbf{H}$ to check when the cosine similarity is zero, i.e., $\delta(\mathbf{H}', \mathbf{B}_1)$ where $\mathbf{H}' = \mathbf{H} - m * \mathbf{B}_1$.

Figure 4.6a shows an example of the cosine similarity for each $\mathbf{B}_i$ when $f_1 = 50$, $f_2 = 26$ and $f_3 = 77$ and m changes from 1 to 120. The result shows that the similarity decreases as subtracting more instances of $\mathbf{B}_1$ from $\mathbf{H}$. For example, the similarity is zero when m is close to f_i as expected, and it gets negative values for further subtractions, since $\mathbf{H}'$ has the term of $-\mathbf{B}_1$. Regardless of the initial similarity of $\mathbf{H}$ with $\mathbf{B}$, the cosine similarity is around zero when m is close to each feature value f_i.

However, there are two main issues in the cosine similarity-based value search. First, finding the feature values in this way needs iterative procedures, slowing down the runtime of data recovery. In addition, it is more challenging when feature values are represented in floating points. Second, the cosine similarity metric may not give accurate results in the recovery. In our earlier example, the similarity of each f_i is zero, when m_i is 49, 29 and 78 respectively.

To efficiently estimate f_i values, we exploit another approach that utilizes the random distribution of the hypervector elements. Let us consider the following equation:

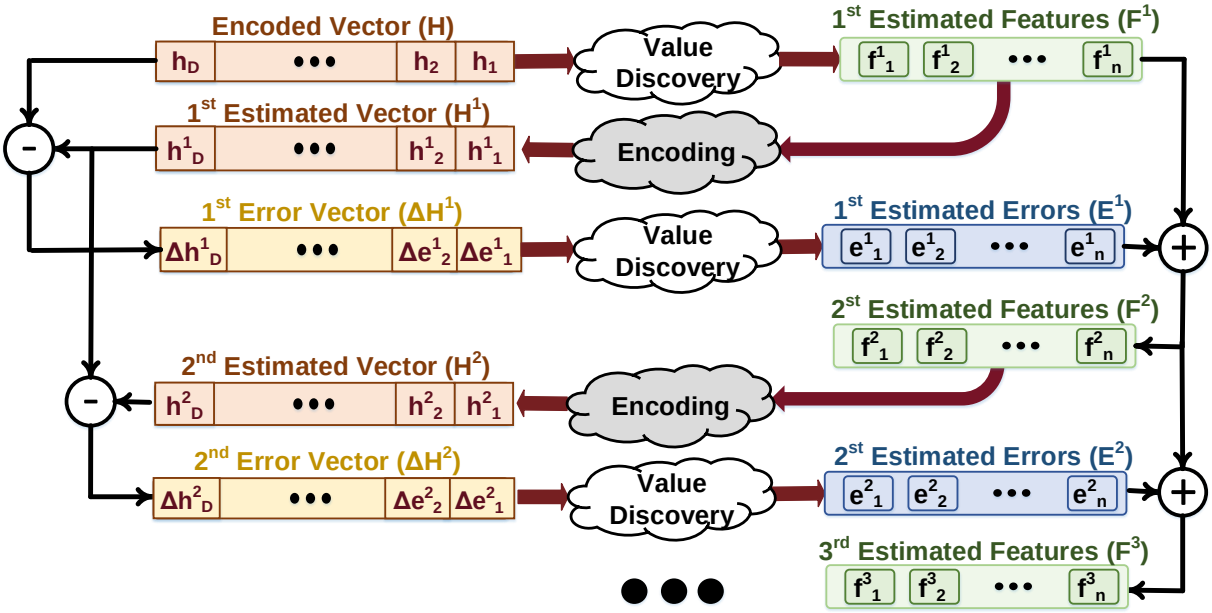


Figure 4.7: Iterative error correction procedure

$$\mathbf{H} \cdot \mathbf{B}_i = f_i * (\mathbf{B}_i \cdot \mathbf{B}_i) + \sum_{j, \forall j \neq i} f_j * (\mathbf{B}_i \cdot \mathbf{B}_j).$$

$\mathbf{B}_i \cdot \mathbf{B}_i$ is D since each element of the base hypervector is either 1 or -1, while $\mathbf{B}_i \cdot \mathbf{B}_j$ is almost zero due to their near-orthogonal relationship. Thus, we can estimate f_i with the following equation, called *value discovery metric*:

$$f_i \simeq \mathbf{H} \cdot \mathbf{B}_i / D.$$

This metric yields an initial estimate of all feature values, say $\mathbf{F}^1 = \{f_1^1, ..., f_n^1\}$. Starting with the initial estimation, SecureHD minimizes the error through an iterative procedure. Figure 4.7 shows the iterative error correction mechanism. We encode the estimated feature vector, $\mathbf{F}^1$, into the high dimensional space, $\mathbf{H}^1 = \{h_1^1, ..., h_D^1\}$. We then compute $\Delta\mathbf{H}^1 = \mathbf{H} - \mathbf{H}^1$, and apply the value extraction metric for $\Delta\mathbf{H}^1$. Since this yields the estimated error, E^1, in the original domain, we add it to the estimated feature vector for the better estimate of the actual features, i.e., $\mathbf{F}^2 = \mathbf{F}^1 + \mathbf{E}^1$. We repeat this procedure until the estimated error converges. To determine the termination condition, we compute the variance of the *error hypervector*, $\Delta\mathbf{H}^i$, at

the end of each iteration. Figure 4.6b shows the variance changes when decoding four example hypervectors. For this experiment, we used two feature vectors whose size is either $n = 1200$ or 1000, where the feature values are uniform-randomly generated. We encoded each feature vector to two hypervectors with either $D = 7,000$ or $D = 10,000$. As shown in the results, the iterations required for accurate recovery depends on both the number of features in the original domain and hypervector dimensions. In the rest of the chapter, we use the ratio of the hypervector dimension to the number of features in the original domain, i.e., $R = D/n$, to evaluate the quality of the data recovery for different feature sizes. The larger R ratio, the larger the retraining iterations are expected to sufficiently recover the data.

Metadata Recovery We utilize the value extraction method to recover the metadata. We calculate how many times each metavector $\{\mathbf{M}_1, ..., \mathbf{M}_k\}$ presents in a segment. If the extracted instances of metavector are similar to the actual C value that we injected, such metavector is considered to be in the segment. However, since the metavector has a small number of elements, i.e., $d << D$ dimensions, it might have a large error in finding the exact C value. Let's assume that, when injecting a metavector C times, the value extraction method identifies a value, $\widehat{C}$, in a range of $[C_{min}, C_{max}]$. The range also includes C. If the metavector does not exist, the value $\widehat{C}$ will be approximately zero, i.e., a range of $[-\varepsilon, \varepsilon]$. The amount of ε depends on the other information stored in the segment.

Figure 4.8a shows the distribution of extracted values, $\widehat{C}$, when injecting 5 metadata 10 times ($C = 10$) into a single segment of a hypervector. These distributions are reported using a Monte Carlo simulation with 1500 randomly generated metavectors. The results show that the distributions of the existing and non-existing cases are overlapped, making the estimation difficult. However, as shown in Figure 4.8b, when using $C = 128$, there is a clear margin between these two distributions which identify the existence of a metadata. Figure 4.8c shows the distributions when we inject 8 metadata into a single segment with $C = 128$. In that case, two distributions overlap, i.e., there are a few cases when we cannot fully recover the metadata.

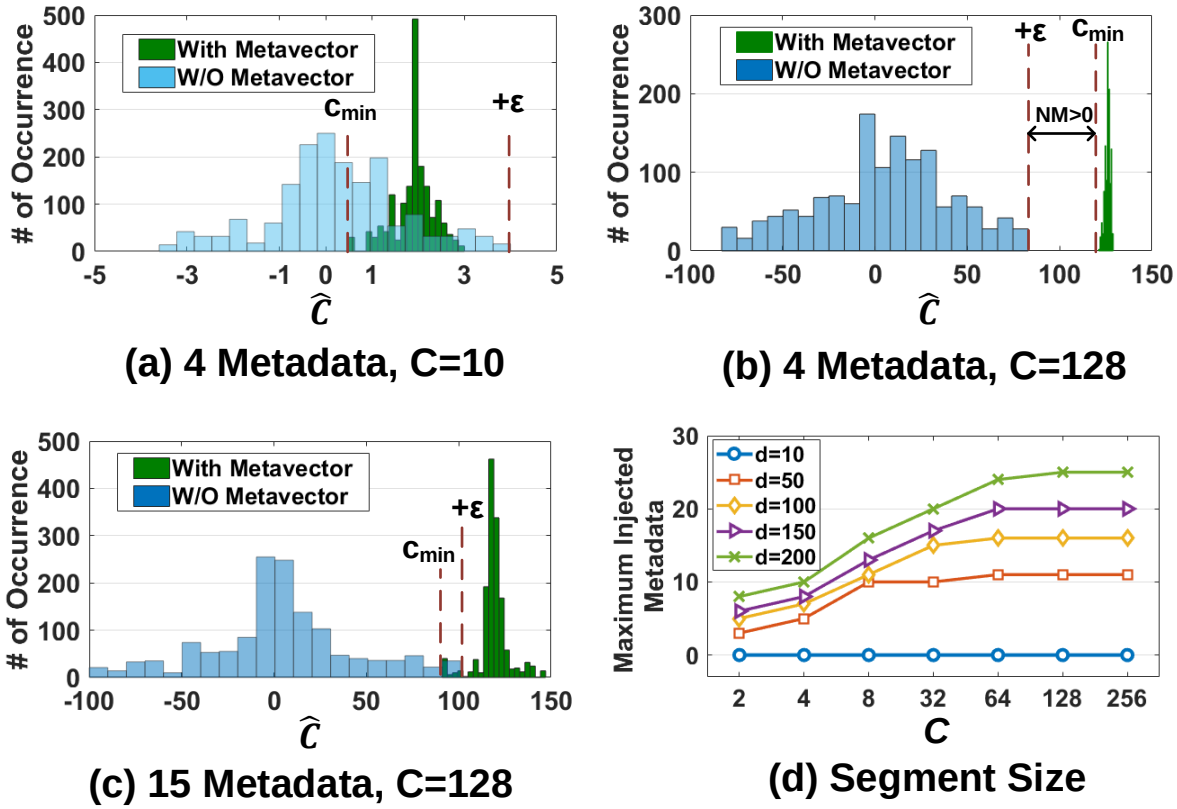


Figure 4.8: Relationship between the number of metavector injections and segment size

We determine C so that the distance between C_{min} and ε is larger than 0. We define the distance as the noise margin, $NM = C_{min} - \varepsilon$. Figure 4.8d shows how many metavectors can be injected for different C values. The results show that the number of meta vectors that we can inject saturates for larger C values. Since the large number of C and segment size, d, also have a higher chance to influence on the accuracy of the data recovery, we choose $C = 128$ and $d = 50$ for our evaluation. In Section 4.7.5, we present a detailed evaluation for different settings of the metavector injection.

Data Recovery After recovering the metadata, SecureHD can recognize the data types and choose the base hypervectors for decoding. We subtract the metadata from the encoded hypervector and start decoding the main data. SecureHD utilizes the same value extraction method to identify the values for each base hypervector. The quality of data recovery depends

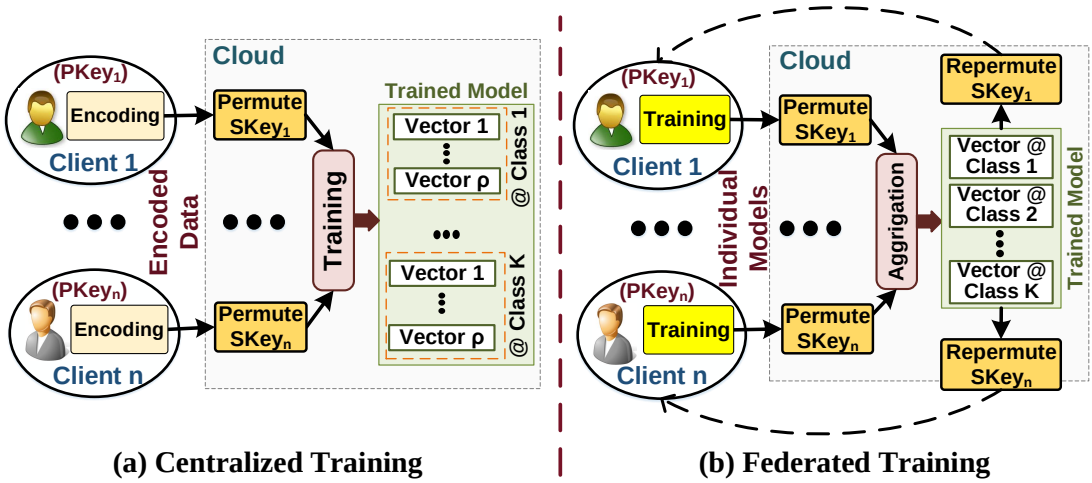


Figure 4.9: Illustration of the classification in SecureHD

on the dimension of hypervectors in the encoded domain (D) and the number of features in the original space (n), i.e., $R = D/n$ defined in Section 4.5.2. Intuitively, with the larger the R value, we can achieve a higher accuracy during the data recovery at the expense of the size of encoded data. For instance, when storing an image with $n = 1000$ pixels in a hypervector with $D = 10,000$ dimensions ($R = 10$), it is expected to achieve high accuracy for the data recovery. In our evaluation, we observed that, with $R = 7$, it is enough to ensure lossless data recovery in the worst case. In Section 4.7.4, we explore more detailed discussion about how R impacts on the accuracy of the recovery procedure.

4.6 Collaborative Learning in HD Space

4.6.1 Hierarchical Learning Approach

Figure 4.9 shows the HD-based collaborative learning in the high-dimensional space. In this chapter, we show two training approaches, *centralized* and *federated* training, which performs classification learning with a large amount of data provided by many clients. The cloud can perform the training procedures using the encoded hypervectors without explicit decoding. It

only needs to permute the encoded data using the SKey of each client. Note that the permutation aligns the encoded data on the same GKey base, even though the cloud does not have the GKeys. It reduces the cost of the learning procedure, and the data can be securely classified even on the untrustworthy cloud. The training procedure creates multiple hypervectors as the trained model, where each hypervector represents the pattern of data points in one class. We refer them to *class hypervectors*.

Approach 1: Centralized Training In this approach, the clients send the encoded hypervectors to the cloud. The cloud permutes them with the SKeys, and a trainer module combines the permuted hypervectors. The training is performed with the following sub-procedures.

(i) Initial training: At the initial stage, it creates the class hypervectors for each class. As an example, for a face recognition problem, SecureHD creates two hypervectors representing "face" and "non-face". These hypervectors are generated with element-wise addition for all encoded inputs which belong to the same class, i.e., one for "face" and the other one for "non-face".

(ii) Multivector expansion: After training the initial HD model, we expand the initial model with cross-validation, so that each class has multiple hypervectors of the size of ρ. The key idea is that, when training with larger data, it may need to capture more distinct patterns with different hypervectors. To this end, we first check cosine similarity for each encoded hypervector again to the trained model. If an encoded data does not correctly match with its corresponding class, it means that the encoded hypervector has a distinct pattern as compared to the majority of all the inputs in the class. For each class, we create a set that includes such mismatched hypervectors and the original model. We then choose two hypervectors, whose similarity is the highest among all pairs in the set, and update the set by adding the selected two into a new hypervector. This is repeated until the set includes only ρ hypervectors.

(iii) Retraining: As the last step, we iteratively adjust the HD model over the same dataset to give higher weights for misclassified samples that may often happen in a large dataset. We check the similarity for each encoded hypervector again with all existing classes. Let us assume that $\mathbf{C}_k^p$ is

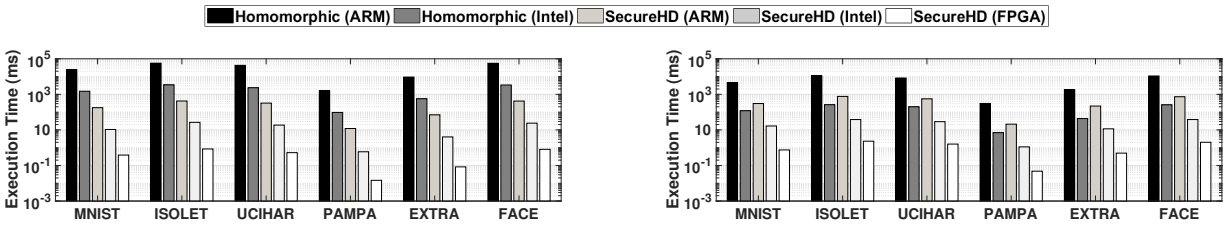
one of the class hypervectors belonging to k^{th} class, where p is the index of multiple hypervectors in the class. If an encoded hypervector $\mathbf{Q}$ belonging to i^{th} class is incorrectly classified to $\mathbf{C}_j^{miss}$, we update the model by

$$\mathbf{C}_j^{miss} = \mathbf{C}_j^{miss} - \alpha\mathbf{Q} \text{ and } \mathbf{C}_i^{\tau} = \mathbf{C}_i^{\tau} + \alpha\mathbf{Q}$$

where $\tau = \text{argmax}_t\, \delta(\mathbf{C}_i^t, \mathbf{Q})$ and α is a learning rate in a range of [0.0, 1.0]. In other words, in the case of misclassification, we subtract the encoded hypervector from the class which it is incorrectly classified to, while adding it to the class hypervector which has the highest similarity in the correct class. This procedure is repeated for predefined iterations, and the final class hypervectors are used for the future inference.

Approach 2: Federated Training The clients may not have enough network bandwidth to send every encoded hypervector. To address this issue, we present the second approach, called *federated* training, as an edge computing. In this approach, the clients individually train initial models, i.e., one hypervector for each class, only using their own encoded hypervectors. Once the cloud receives the initial models of all the clients, it permutes the models with the SKeys and performs element-wise additions to create a global model, $\mathbf{C}_k$, for each k^{th} class.

Since the cloud only knows the initial models for each client, the multivector expansion procedure is not performed in this approach, but we can still execute the retraining procedure explained in Section 4.6.1. To this end, the cloud re-permutes the global model and sends it back to each client. With the global model, each client performs the same retraining procedure. Let us assume that $\widetilde{\mathbf{C}}_k^i$ is the retrained model by the i^{th} client. After the cloud aggregates all $\tilde{\mathbf{C}}_k^i$ with the permutation, it updates the global models by $\mathbf{C}_k = \sum_i \widetilde{\mathbf{C}}_k^i - (n-1) * \mathbf{C}_k$. This is repeated for the predefined iterations. This approach allows the clients to send the trained class hypervectors only for each retraining iteration, thus significantly reducing the network usage.



(a) Encoding (b) Decoding

Figure 4.10: Comparison of SecureHD efficiency to homomorphic algorithm in encoding and decoding

4.6.2 HD Model-Based Inference

With the class hypervectors generated by either approach, we can perform the inference in any device including the cloud and clients. For example, the cloud can receive an encoded hypervector from a client, and permute the dimension with the SKey in the same way to the training procedure. Then, it checks cosine similarity of the permuted hypervector to all trained class hypervectors to label with the corresponding class to the most similar class hypervector. In the case of the client-based inference, once the cloud sends re-permuted class hypervectors to a client, the client can perform the inference for its encoded hypervector with the same similarity check.

4.7 Evaluation

4.7.1 Experimental Setup

We have implemented the SecureHD framework including encoding, decoding, and learning in high-dimensional space using C++. We evaluated the system on three different platforms: Intel i7 7600 CPU with 16GB memory, Raspberry Pi 3, and Kintex-7 FPGA KC705. We also exploit a network simulator, NS-3 [123], for large-scale simulation. We verify the FPGA timing and the functionality of the encoding and decoding by synthesizing Verilog using Xilinx Vivado Design Suite [124]. The synthesis code has been implemented on the Kintex-7 FPGA

KC705 Evaluation Kit. We compare the efficiency of the proposed SecureHD with SEAL, the state-of-the-art C++ implementation of a homomorphic library, Microsoft SEAL [110]. For SEAL, we used the default parameters: polynomial modulus of $n = 2048$, coefficient modulus of $q = 128 - bit$, plain modulus of $t = 1 << 8$, noise standard deviation of 3.9, and decomposition bit count of 16. We evaluate the proposed SecureHD framework with real-world datasets including human activity recognition, phone position identification, and image classification. Table 4.1 summarizes the evaluated datasets. The tested benchmarks range from relatively small datasets collected in a small IoT network, e.g., PAMAP2, to a large dataset which includes hundreds of thousands of images of facial and non-facial data. We also compare the classification accuracy of SecureHD for the datasets with the state-of-the-art learning models shown in the table.

4.7.2 Encoding and Decoding Performance

As explained in Section 4.4.3, SecureHD performs a one-time key generation to distribute the PKeys to each user using the MPC and GC protocols. Table 4.2 listed the number of required logic gates evaluated in the protocol and the amount of required communication between clients. This overhead comes mostly from the first phase of the protocol, since the second phase has been simplified with the two-party GC protocol. The cost of the protocol is dominated by network communication. In our simulation conducted under our in-house network of 100 Mbps, it takes around 9 minutes to create $D = 10,000$ keys for 100 participants. Note that the runtime overhead is negligible since the key generation happens only once before all future computation.

We have also evaluated the encoding and decoding procedure running on each client. We compare the efficiency of SecureHD with the Microsoft SEAL [110]. We run both the SecureHD framework and homomorphic library on ARM Cortex 53 and Intel i7 processors. Figure 4.10 shows the execution time of the SecureHD and homomorphic library to process a single data point. For SecureHD, we used $R = 7$ to ensure 100% data recovery rate for all benchmark datasets. Our evaluation shows that SecureHD achieves on average 133× and 14.7× (145.6× and

Table 4.1: Datasets (n: feature size, K: number of classes)

	n	K	Data Size	Train Size	Test Size	Description/State-of-the-art Model
MNIST	784	10	220MB	60,000	10,000	Handwritten Recognition/DNN[125, 126]
ISOLET	617	26	19MB	6,238	1,559	Voice Recognition/DNN [127, 128]
UCIHAR	561	12	10MB	6,213	1,554	Activity recognition(Mobile)/DNN[129, 128]
PAMAP2	75	5	240MB	611,142	101,582	Activity recognition(IMU)/DNN[91]
EXTRA	225	4	140MB	146,869	16,343	Phone position recognition/AdaBoost[93]
FACE	608	2	1.3GB	522,441	2,494	Face recognition/Adaboost[130]

Table 4.2: Overhead for key generation and distribution

Phases		**Phase 1**			**Phase 2**
# of Clients		*10*	*50*	*100*	
D=1000	*# of Gates*	11K	51K	101K	8.9K
	Communication	7.1MB	160MB	650MB	284MB
D=5000	*# of Gates*	55K	255K	505K	56.4K
	Communication	35MB	813MB	3.24GB	1.8MB
D=10,000	*# of Gates*	110K	510K	101K	122.9K
	Communication	70.34MB	1.64GB	6.46GB	3.93MB

6.8×) speedup for the encoding and decoding, respectively, as compared to the homomorphic technique running on the ARM architecture (Intel i7). The encoding of SecureHD running on embedded devices (ARM) is still 8.1× faster than the homomorphic encryption running on the high-performance client (Intel i7). We also compare the SecureHD efficiency on the FPGA implementation. We observe that the encoding and decoding of SecureHD achieve 626.2× and 389.4× (35.5× and 20.4×) faster execution as compared to the SecureHD execution on the ARM (Intel i7). For example, the proposed FPGA implementation is able to encode 2,600 data points and decode 1,335 for the MNIST images in a second.

4.7.3 Evaluation of SecureHD Learning

Learning Accuracy Based on the proposed SecureHD, clients can share the information with the cloud in a secure way, such that the cloud cannot understand the original data while

still performing the learning tasks. Along with the proposed two learning approaches, we also evaluate the state-of-the-art HD classification approach, called *one-shot HD model*, which trains the model using a single hypervector per class with no retraining [82, 109]. For the centralized training, we trained two models, one that has 64 class hypervectors for each class and the other one that has 16 for each class. We call them as *Centralized-64* and *Centralized-16*. The retraining procedure was performed for 100 times with $\alpha = 0.05$, since the classification accuracy was converged with this configuration for all the benchmarks.

Figure 4.11 shows the classification accuracy of the SecureHD for the different benchmarks. The results show that the centralized training approach achieves high classification accuracy comparable to the state-of-the-art learning methods such as DNN models. We also observed that, by training more hypervectors per class, it can provide higher classification accuracy. For example, for the federated training approach, which does not use multivectors, the classification accuracy is 90% on average, which is 5% lower than the Centralized-64. As compared to the state-of-the-art one-shot HD model which does not retrain models, Centralized-64 achieves 15.4% higher classification accuracy on average.

Scalability of SecureHD Learning As discussed in Section 4.6.1, the proposed learning method is designed to effectively handle a large amount of data. To understand the scalability of the proposed learning method, we evaluate how the accuracy is changed when the training data are come from different numbers of clients, with simulation on NS-3 [123]. In this experiment, we exploit three datasets, PAMAP2, EXTRA, and FACE, which include information of where data points are originated. For example, PAMAP2 and EXTRA are gathered from 7 and 56 individual users. Similarly, the FACE dataset includes 100 clients that have different facial images with each other. Figure 4.12a and b show the accuracy changes for the centralized and federated training approaches. The result shows that increasing the number of clients improves classification accuracy by training with more data. Furthermore, as compared to the one-shot HD model, the two proposed approaches show better scalability in terms of accuracy. For example,

the accuracy difference between the proposed approach and the one-shot model grows as more clients engage in the training. Considering the centralized training, the accuracy difference for the FACE dataset is 5% when trained with one client, while it is 14.7% for the 60-client case. This means that the multivector expansion and retraining techniques are effective to learn with a large amount of data.

We also verify how the SecureHD learning methods work with constrained network conditions that often happen in IoT systems. In our network simulation, we assume the worst-case network condition, i.e., all clients share the bandwidth of a standard WiFi 802.11 network. Note that it is a worst-case scenario and in practice, each embedded device may not share the same network. Figure 4.12c shows that the network bandwidth limits the number of hypervectors that can be sent for each second as multiple clients involve the learning task. For example, a network with 100 clients can send the lower number of hypervectors by 23.6× than a single-client case.

As discussed before, the federated learning can be exploited to overcome the limited network bandwidth at the expense of the accuracy loss. Another solution is to use a reduced dimension in the centralized learning. As shown in Figure 4.12c, when $D = 1,000$, clients can send the data to the cloud with 353 samples per second, which is 10 times higher than the case of $D = 10,000$. Figure 4.12d shows how learning accuracy changes for different dimension settings. The results show that reducing the hypervector dimensions to 4000 and 1000 dimensions has less than 1.4% and 5.3% impact on the classification accuracy. This strategy gives another choice of the trade-off between accuracy and network communication cost.

4.7.4 Data Recovery Trade-offs

As discussed in Section 4.5.2, the proposed SecureHD framework provides a decoding method for the authorized user that has the original Pkeys used in the encoding. Figure 4.13a shows the data recovery rate on images with different pixel sizes. To verify the proposed recovery method in the worst case scenario, we created 1000 images whose pixel values are randomly

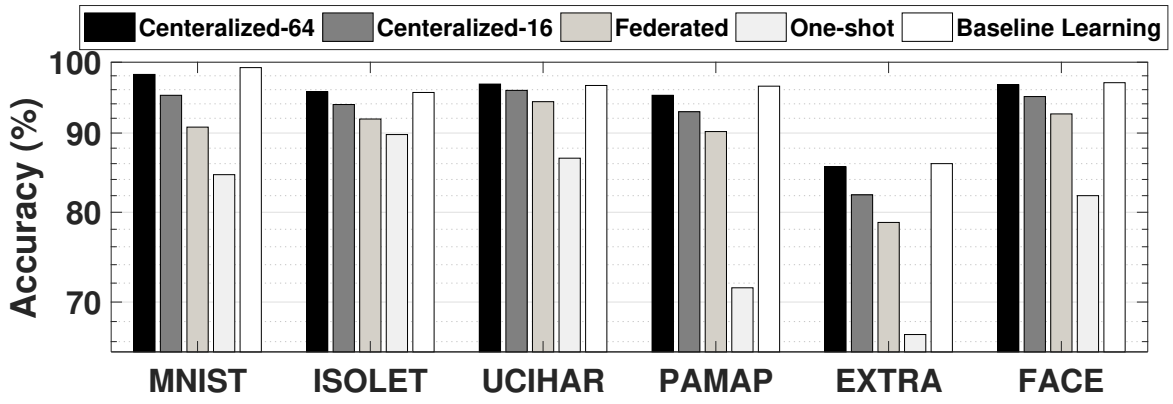


Figure 4.11: SecureHD classification accuracy

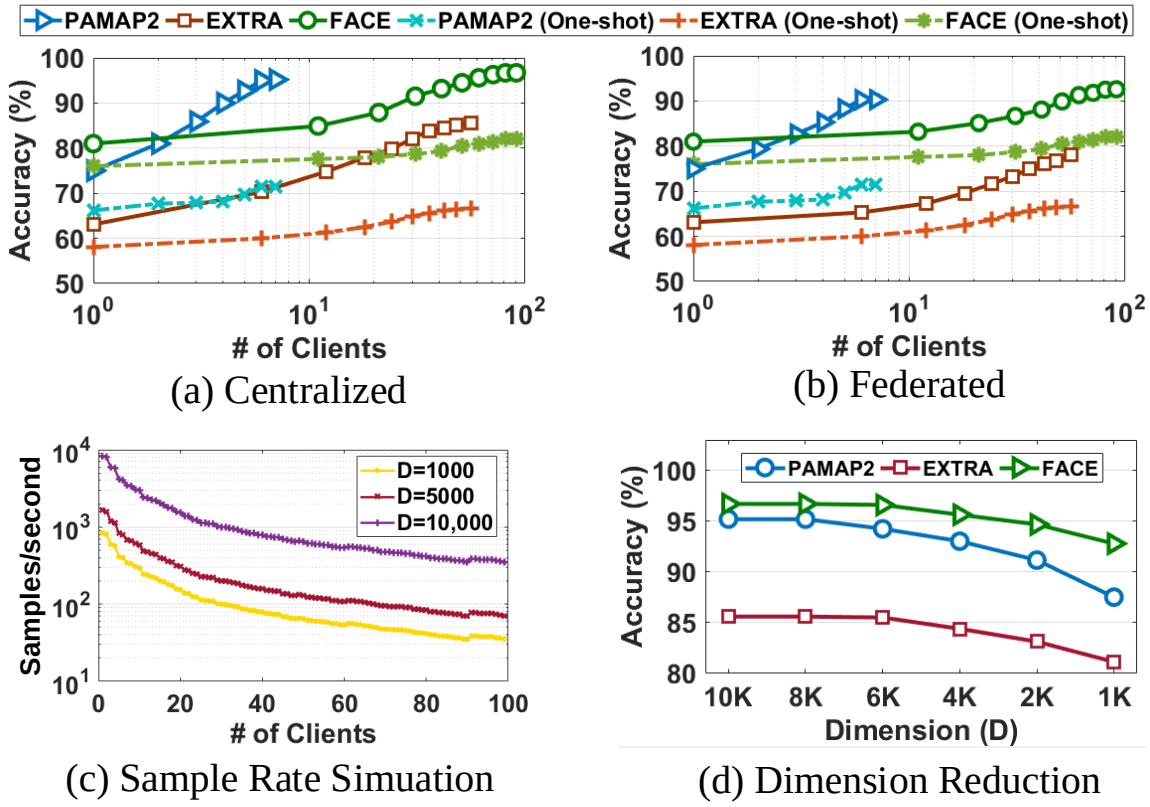


Figure 4.12: Scalability of SecureHD classification

chosen, and report the average error when we map the 1000 images to $D = 10,000$ dimension. The x-axis shows the ratio R, i.e., D/n where the number of hypervector dimension (D) to the number of pixels (n) in an image. The data recovery rate depends on the precision of the pixel values. Using high-resolution images, SecureHD requires a larger R value to ensure 100%

(a) Image (b) Text

Figure 4.13: Data recovery accuracy of SecureHD

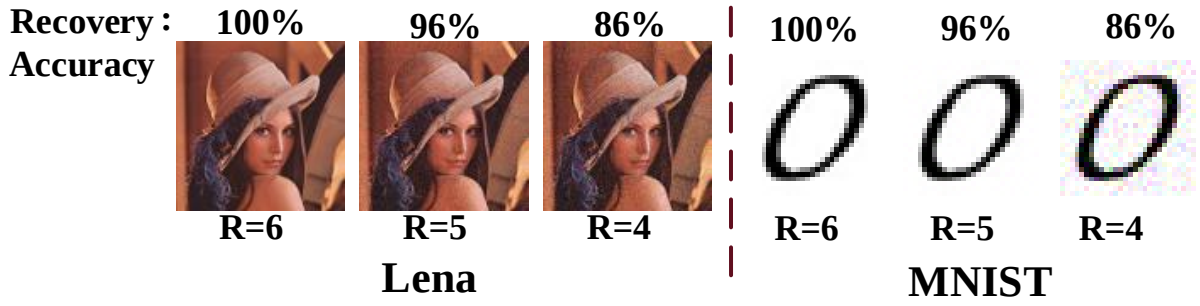


Figure 4.14: Example of image recovery

accuracy. For instance, for images with 32-bit pixel resolution, SecureHD can achieve 100% data recovery using $R = 7$, while lower resolution images (e.g., 16 and 8-bits) requires $R = 6$ to ensure 100% data recovery. Our evaluation shows that our method can decode any input image with 100% data recovery rate using $R = 7$. This means that we can securely encode data with $4\times$ smaller size compared to the homomorphic encryption library which increases the data size by 28 times through the encryption.

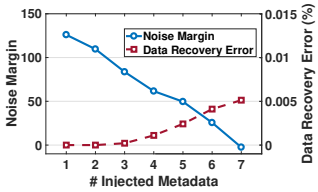
We also evaluate the SecureHD framework with a text dataset written in three different European languages [131]. Figure 4.13b shows the accuracy of data recovery for the three languages. The x-axis is the ratio between the length of hypervectors to the number of characters in the text when $D = 10,000$. Our method assigns a single value to each alphabet letter and encodes the texts with the hypervectors. Since the number of characters in these languages is less than 49, we require at most 6 bits to represent each alphabet. In terms of the data recovery, it is equal to encoding the same size image with the 6-bit pixel resolution. Our evaluation shows that SecureHD can provide 100% data recovery rate with $R = 6$.

Figure 4.14 shows the quality of the data recovery for two example images. The Lena and MNIST image have 100×100 pixels and 28×28 pixels, respectively. Our encoding maps the input data to hypervectors with different dimensions. For example, the Lena image with $R = 6$ means that the image has been encoded with $D = 60,000$ dimensions. Our evaluation shows that SecureHD can achieve lossless data recovery on Lena photo when $R \geq 6$, while using $R = 5$ and $R = 4$ the data recovery rates are 93% and 68%. Similarly, $R = 5$ and $R = 4$ provide 96% and 56% data recovery for the MNIST images.

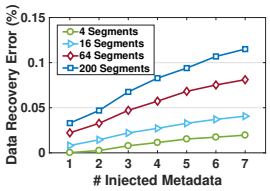
4.7.5 Metadata Recovery Trade-offs

As discussed in Section 4.5.2, the metadata injection method needs to be performed such that it ensures 100% metadata recovery and it has minimal impacts on the original hypervector for the learning and data recovery. The solid line in Figure 4.15a shows the noise margin when injecting multiple metavectors into a single segment of hypervector when the number of elements in the segment is chosen by $50 (= d)$. We report the results based on the worst case for 5000 Monte Carlo simulation. The results show that each segment can store 6 metavectors at most to take a positive noise margin that ensures 100% metadata recovery. The dotted line shows the data recovery error rate for different numbers of metavectors injected into a single segment. Our evaluation shows that adding 6 metavectors has less than 0.005% impact on the data recovery rate.

Since the number of metavectors which can be injected in one segment is limited, we may need to distribute the metadata in different segments. Figure 4.15b presents the impact of the metadata injection on the data recovery error rate. When we inject 6 metadata into each of all 200 segments, i.e., 1200 metavectors in total, the impact on the recovery accuracy is still minimal, i.e., less than 0.12%.



(a) Single Segment



(b) Multiple Segments

Figure 4.15: Data recovery rate for different settings of metavector injection

4.8 Conclusion

In this chapter, we presented a novel framework, called SecureHD, which provides secure data encoding and learning based on HD computing. With our framework, clients can securely send their data to untrustworthy cloud, while the cloud can perform the learning tasks without the knowledge of the original data. Our proof-of-concept implementation demonstrates that the proposed SecureHD framework successfully performs the encoding and decoding tasks with high efficiency, e.g., 145.6× and 6.8× faster than the state-of-the-art encryption/decryption library [17]. Our learning method achieves accuracy of 95% on average for diverse practical learning tasks, which is comparable to the state-of-the-art learning methods [125, 128, 128, 91, 93, 130]. In addition, SecureHD provides lossless data recovery with 4× reduction in the data size compared to the existing encryption solution. In the next chapter, we show how we can solve other cognitive tasks such as DNA pattern matching based on HD computing.

This chapter contains material from "A Framework for Collaborative Learning in Secure High-Dimensional Space", by Mohsen Imani, Yeseong Kim, Sadegh Riazi, John Merssely, Patrick Liu, Farinaz Koushanfar and Tajana S. Rosing, which appears in IEEE Cloud Computing, July 2019. The dissertation author was the primary investigator and author of this paper.

Chapter 5

HD Computing Beyond Classical Learning: DNA Pattern Matching

5.1 Introduction

In Chapters 3 and Chapter 4, we discussed how to solve classification problems using HD computing by mapping feature vectors into hypervectors. IoT applications often need to process other data types such as text data. For the other learning algorithms, such as deep learning, various techniques are developed to handle such data types, e.g., Word2vec [6]. In this chapter, we show how we can describe such non-feature vector types using HD computing, focusing on DNA sequence data used in diverse bioinformatics applications [132, 133]. Our key focus is the acceleration of the pattern matching problem which is an important ingredient of many DNA alignment techniques to enable personalized IoT-based healthcare [18] and on-site disease detection [19].

In general, a DNA sequence is a special case of text data, i.e., a string which consists of four nucleotide characters, *A, C, G,* and *T*. The pattern matching problem is to examine the occurrence of a given *query* string in a *reference* string. For example, the technique can discover possible diseases by identifying which reads (short strings) match a reference human genome consisting of 100 millions of DNA bases [19]. The efficient acceleration of the DNA pattern matching is still an open question. Although prior research has developed acceleration systems on parallel computing platforms, e.g., GPU [20] and FPGA [21], they offer only limited improvements. The primary reason is that existing pattern matching algorithms they relied on, e.g., Boyer-Moore (BM) and Knuth-Morris-Pratt (KMP) algorithms [19], are at heart sequential processes. Their acceleration strategies parallelize the workloads by either scheduling multiple DNA searching tasks or streaming long-length DNA sequences, consequently resulting in high memory requirements and runtime. In this context, the pattern matching problem should be revisited not only to accelerate the existing algorithms on the parallel computing platforms, but also to redesign a hardware-friendly algorithm itself.

In this chapter, we propose a novel hardware-software codesign of genome identity

extractor using hyperdimensional computing, in short GenieHD. Our work includes a new pattern matching algorithm and the accelerator design. Based on the HD computing which are specialized for pattern-based computations, GenieHD transforms the inherent sequential processes of the pattern matching task to highly-parallelizable computations. The followings summarize the contributions shown in this chapter:

1) We propose a novel hardware-friendly pattern matching algorithm based on HD computing. GenieHD encodes DNA sequences to hypervectors and discover multiple patterns with a light-weight HD operation. Besides, we can reuse the encoded hypervectors to query many DNA sequences newly sampled which are common in practice.

2) We show an acceleration architecture to execute the proposed algorithm efficiently on general parallel computing platforms. The proposed design significantly reduces the number of memory accesses to process the HD operations, while fully utilizing the available parallel computing resources. We also present how to implement the proposed acceleration architecture on the three parallel computing platforms, GPGPU, FPGA, and ASIC.

3) We evaluate GenieHD with practical datasets, human and Escherichia Coli (E. coli) genome sequences. The experimental results show that GenieHD significantly accelerates the DNA matching algorithm, e.g., 44.4× speedup and 54.1× higher energy efficiency when comparing our FPGA-based design to a state-of-the-art FPGA-based design. As compared to an existing GPU-based implementation, our ASIC design which has the similar die size outperforms the performance and energy efficiency by 122× and 707×. We also show that the power consumption can be further saved by 50% by allowing minimal accuracy loss of 1%.

5.2 Related Work

Hyperdimensional Computing HD computing is originated from a human memory model, called sparse distributed memory developed in neuroscience [7]. Recently, computer

scientists recapped the memory model as a cognitive, pattern-oriented computing method. For example, prior researchers showed that the HD computing-based classifier is effective for diverse applications, e.g., text classification, multimodal sensor fusion, speech recognition, and human activity classification [88, 134, 15, 135, 136]. The work in [137] recently uses HD computing for DNA sequence classification. Prior work also show application-specific accelerators on different platforms, e.g., FPGA [138, 139, 140, 141] and ASIC [82]. Processing in-memory chips were also fabricated based on 3D VRRAM technology [90]. The previous works mostly utilize HD computing as a solution for classification problems. In this chapter, we show that HD computing is an effective method for other pattern-centric problems, and propose a novel DNA pattern matching algorithm.

DNA Pattern Matching Acceleration The efficient pattern matching is an important task in many bioinformatics applications, e.g., single nucleotide polymorphism (SNP) identification, on-site disease detection and precision medicine development [19]. Many acceleration systems have been proposed on diverse platforms, e.g., multiprocessor [142] and FPGA [143]. For example, the work in [21] proposes an FPGA accelerator that parallelizes partial matches for a long DNA sequence based on KMP algorithm. The work in [20] proposed a parallel pattern matching method that streams the long-length reference into different CUDA cores. Our work is different in that we accelerate a new HD computing-based algorithm which is specialized for parallel systems and also effectively scales for the number of queries to process.

5.3 GenieHD Overview

Figure 5.1 illustrates the overview of the proposed GenieHD design. GenieHD exploits HD computing to design an efficient DNA pattern matching solution (Section 5.4.) During the offline stage, we convert the *reference* genome sequence into hypervectors and store into the *HV database*. In the online stage, we also encode the *query* sequence given as an input. GenieHD

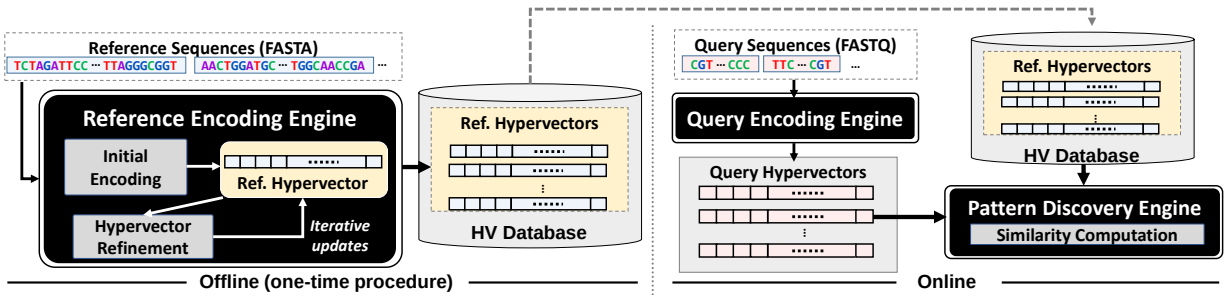


Figure 5.1: Overview of GenieHD

in turn identifies if the query exists in the reference or not, using a light-weight HD operation that computes hypervector similarities between the query and reference hypervectors. All the three processing engines perform the computations with highly-parallelizable HD operations. Thus, many parallel computing platforms can accelerate the proposed algorithm. We present the implementation on GPGPU, FPGA, and ASIC based on a general acceleration architecture (Section 5.5.)

Nowadays, raw DNA sequences are publicly downloadable in standard formats, e.g., FASTA for references [144]. Likewise, the HV databases can provide the reference hypervectors encoded in advance, so that users can efficiently examine different queries without performing the offline encoding procedure repeatedly. For example, it is typical to perform the pattern matching for billions of queries streamed by a DNA sequencing machine. In this context, we also evaluate how GenieHD scales better than state-of-the-art methods when handling multiple queries (Section 5.6.)

5.4 DNA Pattern Matching Using HD Computing

The major difference between HD and conventional computing is the computed data elements. Instead of booleans and numbers, HD computing performs the computations on ultra-wide words, i.e., hypervectors, where all words are responsible to represent a datum in a distributed manner. HD computing mimics important functionalities of the human memory [7]. For example,

the brain efficiently aggregates/associates different data and understands similarity between data. The HD computing implements the aggregation and association using the hypervector addition and multiplication, while measuring the similarity based on a distance metric between hypervectors. The HD operations can be effectively parallelized in the granularity of the dimension level.

In this work, we represent DNA sequences with hypervectors, and perform the pattern matching procedure using the similarity computation. To encode a DNA sequence to hypervectors, GenieHD uses four hypervectors corresponding to each base alphabet in $\Sigma = \{A, C, G, T\}$. We call the four hypervectors as *base hypervectors*, and denote with $\Sigma_{HV} = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$[1]. Each of the hypervectors has D dimensions where a component is either -1 or +1 (biopolar), i.e., $\{-1, +1\}^D$. The four hypervectors should be uncorrelated to represent their differences in sequences. For example, $\delta(\mathbf{A}, \mathbf{C})$ should be nearly zero, where δ is the dot-product similarity. The base hypervectors can be easily created, since any two hypervectors whose components are randomly selected in $\{-1, 1\}$ have almost zero similarity, i.e., *nearly orthogonal*.

5.4.1 DNA Sequence Encoding

DNA pattern encoding: GenieHD maps a DNA pattern by combining the base hypervectors. Let us consider a short query string, 'GTACG'. We represent the string with $\mathbf{G} \times \rho^{\mathbf{1}}(\mathbf{T}) \times \rho^{\mathbf{2}}(\mathbf{A}) \times \rho^{\mathbf{3}}(\mathbf{C}) \times \rho^{\mathbf{4}}(\mathbf{G})$, where $\rho^n(\mathbf{H})$ is a *permutation* function that shuffles components of $\mathbf{H}$ ($\in \Sigma_{HV}$) with n-bit(s) rotation. For the sake of simplicity, we denote $\rho^n(\mathbf{H})$ as $\mathbf{H}^n$. $\mathbf{H}^n$ is nearly orthogonal to $\mathbf{H} = \mathbf{H}^0$ if $n \neq 0$, since the components of a base hypervector are randomly selected and independent of each other. Hence, the hypervector representations for any two different strings, $\mathbf{H}_\alpha$ and $\mathbf{H}_\beta$, are also nearly orthogonal, i.e., $\delta(\mathbf{H}_\alpha, \mathbf{H}_\beta) \simeq 0$. The hyperspace of D dimensions can represent 2^D possibilities. The enormous representations are sufficient to map different DNA patterns to near-orthogonal hypervectors.

Since the query sequence is typically short, e.g., 100 to 200 characters, the cost for

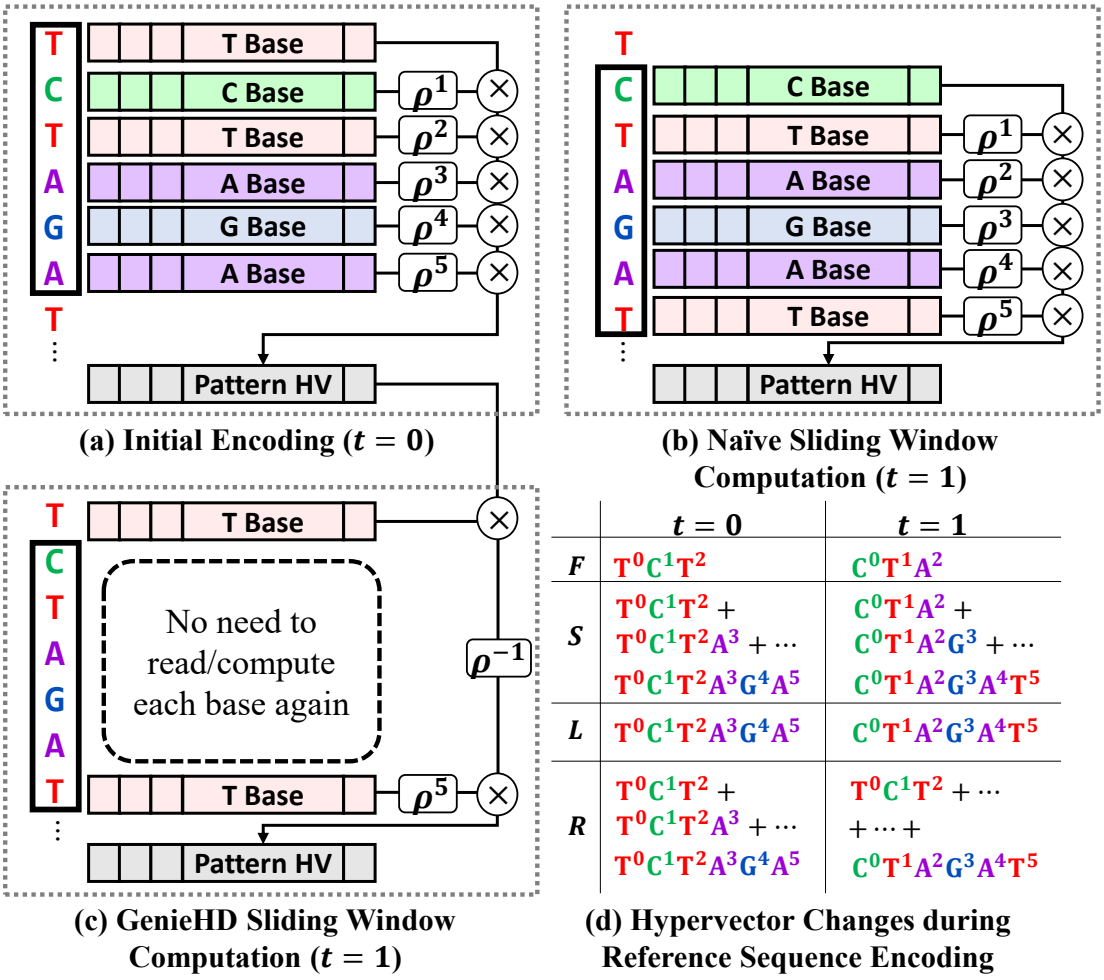
[1] In this chapter, we use bold Latin symbols to represent hypervectors.

the online query encoding step is negligible. In the followings, we discuss how GenieHD can efficiently encode the long-length reference sequence.

Reference encoding: The goal of the reference encoding is to create hypervectors that include all combinations of patterns. In practice, the approximate lengths of the query sequences are known, e.g., the DNA read length of the sequencing technology. Let us defined that the lengths of the queries are in a range of $[\perp, \top]$. The length of the reference sequence, $\mathcal{R}$, is denoted by N. We also use following notations: (i) $\mathbf{B_t}$ denotes the base hypervector for the t-th character in $\mathcal{R}$ (0-base indexing), and (ii) $\mathbf{H}_{(a,b)}$ denotes the hypervector for a subsequence, $\mathbf{B}_a^0 \times \mathbf{B}_{a+1}^1 \times \cdots \times \mathbf{B}_{a+b-1}^{b-1}$.

Let us first consider a special case that encodes every substring of the size n from the reference sequence, i.e., $n = \perp = \top$. The substring can be extracted using a sliding window of the size n to encode $\mathbf{H}_{(t,n)}$. Figure 5.2(a) illustrates the encoding method for the first substring, i.e., $t = 0$, when $n = 6$. A naive way to encode the next substring, $\mathbf{H}_{(1,n)}$, is to run the permutations and multiplications again for each base, as shown in Figure 5.2(b). Figure 5.2(c) shows how GenieHD optimizes it based on HD computing specialized to remove and insert new information. We first multiply $\mathbf{T}^0$ with the previously encoded hypervector, $\mathbf{T}^0\mathbf{C}^1\mathbf{T}^2\mathbf{A}^3\mathbf{G}^4\mathbf{A}^5$. The multiplication of two identical base hypervectors yields the hypervector whose elements are all 1s. Thus, it removes the first base from $\mathbf{H}_{0,n}$, producing $\mathbf{C}^1\mathbf{T}^2\mathbf{A}^3\mathbf{G}^4\mathbf{A}^5$. After performing the rotational shift (ρ^{-1}) and element-wise multiplication for the new base of the sliding window ($\mathbf{T}^5$), we obtain the desired hypervector, $\mathbf{C}^0\mathbf{T}^1\mathbf{A}^2\mathbf{G}^3\mathbf{A}^4\mathbf{T}^5$. This scheme only needs two permutations and multiplications regardless of the substring size n.

Algorithm 2 describes how GenieHD encode the reference sequence in the optimized fashion; Figure 5.2(d) shows how the algorithm runs for the first two iterations when $\perp = 3$ and $\top = 6$. The outcome is $\mathbf{R}$, i.e., the reference hypervector, which combines all substrings whose sizes are in $[\perp, \top]$. The algorithm starts with creating three hypervectors, $\mathbf{S}$, $\mathbf{F}$, and $\mathbf{L}$, (Line 1$\sim$3). $\mathbf{S}$ includes all patterns of $[\perp, \top]$ in each sliding window; $\mathbf{F}$ and $\mathbf{L}$ keep tracks of the first and last



	$t = 0$	$t = 1$
F	$T^0C^1T^2$	$C^0T^1A^2$
S	$T^0C^1T^2 + T^0C^1T^2A^3 + \cdots + T^0C^1T^2A^3G^4A^5$	$C^0T^1A^2 + C^0T^1A^2G^3 + \cdots + C^0T^1A^2G^3A^4T^5$
L	$T^0C^1T^2A^3G^4A^5$	$C^0T^1A^2G^3A^4T^5$
R	$T^0C^1T^2 + T^0C^1T^2A^3 + \cdots + T^0C^1T^2A^3G^4A^5$	$T^0C^1T^2 + \cdots + \cdots + C^0T^1A^2G^3A^4T^5$

Figure 5.2: Illustration of Encoding. For (a), (b), and (c), the window size is 6. (d) illustrates the reference encoding steps described in Algorithm 2.

hypervectors for the $\perp$-length and $\top$-length patterns, respectively. Intuitively, this initialization needs $O(\top)$ hypervector operations. The main loop implements the sliding window scheme for multiple lengths in $[\perp, \top]$. It computes the next $\mathbf{L}$ using the optimized scheme (Line 5). In Line 6, it subtracts $\mathbf{F}$, i.e., the shortest pattern in the previous iteration, and multiply $\mathbf{B}_t^{-1}$ to remove the first base from all patterns combined in $\mathbf{S}$. Then, $\mathbf{S}$ includes the patterns in the range of $[\perp, \top - 1]$ for the current window. After adding $\mathbf{L}$ whose length is $\top$, we accumulate $\mathbf{S}$ to $\mathbf{R}$. Lastly, we update the first pattern $\mathbf{F}$ in the same way to $\mathbf{L}$ (Line 7). The entire iterations need $O(N)$ operations regardless of the pattern length range, thus the total complexity of this

Algorithm 2: Reference Encoding Algorithm

```
1 S ← H_(0,⊥) + H_(0,⊥+1) + ··· + H_(0,⊤)
2 F ← H_(0,⊥); L ← H_(0,⊤)
3 R ← S
4 for t ← 0 to N − ⊤ do
5 |   L ← B_t^{-1} × L^{-1} × B_{t+⊤}^{⊤}
6 |   S ← B_t^{-1} × (S − F)^{-1} + L; R ← R + S
7 |   F ← B_t^{-1} × F^{-1} × B_{t+⊥}^{⊥}
8 end
```

algorithm[2] is $O(N+\top)$. Finally, $\mathbf{R}$ includes all the hypervector representations of the desired lengths existing in the reference.

5.4.2 Pattern Matching

GenieHD performs the pattern matching by computing the similarity between $\mathbf{R}$ and $\mathbf{Q}$. Let us assume that $\mathbf{R}$ is the addition of P hypervectors (i.e., P distinct patterns), $\mathbf{H}_1 + \cdots + \mathbf{H}_P$. The dot product similarity is computed as follows:

$$\delta(\mathbf{R},\mathbf{Q}) = \delta(\mathbf{H}_\lambda,\mathbf{Q}) + \underbrace{\sum_{i=1,i\neq\lambda}^{P} \delta(\mathbf{H}_i,\mathbf{Q})}_{\textbf{Noise}}.$$

If $\mathbf{H}_\lambda$ is equal to $\mathbf{Q}$, since the similarity for the two identical biopolar hypervectors are D, i.e., $\delta(\mathbf{H}_\lambda,\mathbf{Q}) = D$. The similarity between any two different patterns is nearly zero, i.e., $\delta(\mathbf{H}_i,\mathbf{Q}) \simeq 0$ of the noise term. Thus, the following criteria checks if $\mathbf{Q}$ exists in $\mathbf{R}$:

$$\frac{\delta(\mathbf{R},\mathbf{Q})}{D} > T \tag{5.1}$$

where T is a threshold. We call $\frac{\delta(\mathbf{R},\mathbf{Q})}{D}$ as the *decision score*.

The accuracy of this decision process depends on (i) the amount of the noise and (ii) threshold value, T. To precisely identify patterns in GenieHD, we develop a concrete statistical

[2]Due to the limited space, we omit the finalization step which combines the patterns for $t > N - \top$; it can be implemented in a straight-forward way by modifying the main loop so that it does not use $\mathbf{L}$.

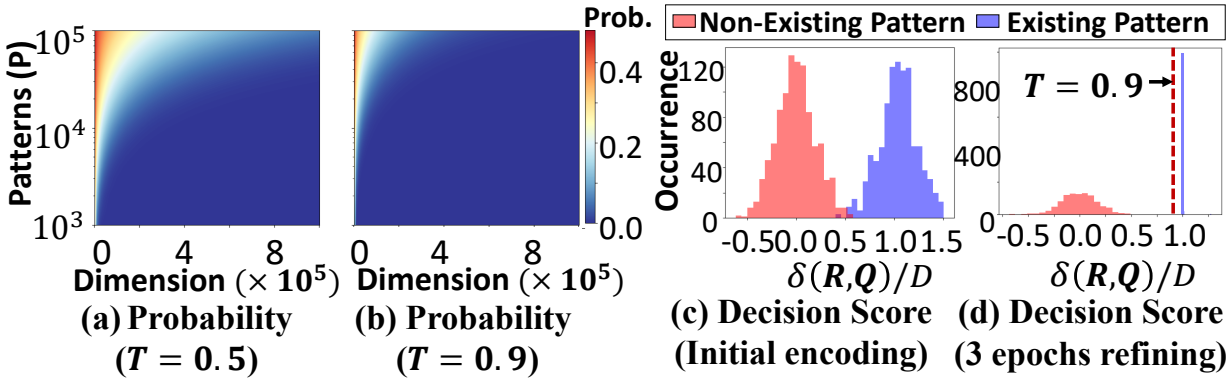


Figure 5.3: Similarity Computation in Pattern Matching. (a) and (b) are computed using Equation 5.2. The histograms shown in (c) and (d) are obtained by testing 1,000 patterns for each of the existing and non-existing cases when **R** is encoded for a random DNA sequence using $D = 100,000$ and $P = 5,000$.

method that estimates the worst-case accuracy. The similarity metric computes how many components of **Q** are the same to the corresponding components for each $\mathbf{H}_i$ in **R**. There are $P \cdot D$ component pairs for **Q** and $\mathbf{H}_i$ ($0 \leq i < P$). The probability that each pair is the same is $\frac{1}{2}$ for all components if **Q** is a random hypervector. The similarity, $\delta(\mathbf{R}, \mathbf{Q})$, can be then viewed as a random variable, X, which follows a binomial distribution, $X \sim B(P \cdot D, \frac{1}{2})$. Since D is large enough, X can be approximated with the normal distribution:

$$X \sim N\left(\frac{P \cdot D}{2}, \frac{P \cdot D}{4}\right).$$

When x component pairs of **R** and **Q** have the same value, $(P \cdot D - x)$ pairs have different values, thus $\delta(\mathbf{R}, \mathbf{Q}) = 2x - P \cdot D$. Hence, the probability that satisfies Equation 5.1 is $Pr(X > \frac{(T+P) \cdot D}{2})$. We can convert X to the standard normal distribution, Z:

$$Pr\left(Z > T \cdot \sqrt{\frac{D}{P}}\right) = \frac{1}{\sqrt{2\pi}} \int_{T \cdot \sqrt{\frac{D}{P}}}^{\infty} e^{-t^2/2}\, dt \tag{5.2}$$

In other words, Equation 5.2 represents the probability that mistakenly determines that **Q** exists in **R**, i.e., false positive.

Figure 5.3(a) and (b) visualizes the probability of the error for different D and P com-

binations. For example, when $D = 100,000$ and $T = 0.5$, we can identify $P = 10,000$ patterns with 5.7% error using a single similarity computation operation. The results also show that using larger D values can improve the accuracy. However, the larger dimensionality requires more hardware resources. Another option to improve the accuracy is using a larger similarity threshold, T, however it may increase true negatives. GenieHD uses the following two techniques to address this issue.

Hypervector refinement The first technique is to refine the reference hypervector. Let us recall Algorithm 2. In the refining step, GenieHD reruns the algorithm to update $\mathbf{R}$. Instead of accumulating $\mathbf{S}$ to $\mathbf{R}$ (Line 6), we add $\mathbf{S} \times (1 - \delta(\mathbf{S}, \mathbf{R})/D)$. The refinement is performed for multiple epochs. Figure 5.3(c) and (d) show how the distribution of the decision scores changes for the existing and non-existing cases by the refinement. The results show that the refinement makes the decision scores of the existing cases close to 1. Thus, we can use a larger T for higher accuracy. The successful convergence depends on i) the number of patterns included in $\mathbf{R}$ with D dimensions, i.e., D/P, and ii) the training epochs. In our evaluation, we observe that, when $\mathbf{R}$ includes $P = D/10$ patterns and use $T = 0.9$, we only need five epochs, and GenieHD can find all patterns with the error of less than 0.003%.

Multivector generation To precisely discover patterns of the reference sequence, we also use multiple hypervectors so that they cover every pattern existing in the reference without loss. During the initial encoding, whenever $\mathbf{R}$ reaches the maximum capacity, i.e., accumulating P distinct patterns, we store the current $\mathbf{R}$ and reset its components to 0s to start computing a new $\mathbf{R}$. GenieHD accordingly fetches the stored $\mathbf{R}$ during the refinement. Even though it needs to compute the similarity values for the multiple $\mathbf{R}$ hypervectors, GenieHD can still fully utilize the parallel computing units by setting D to a sufficiently large number.

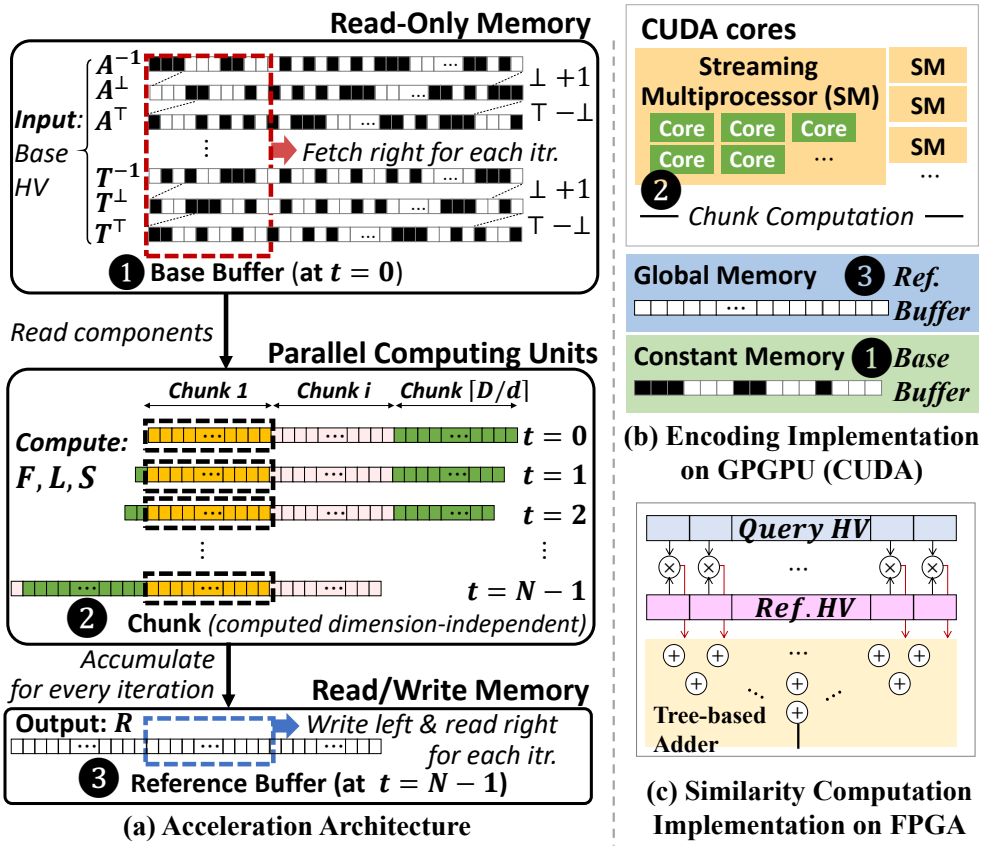


Figure 5.4: Hardware Acceleration Design. The dotted boxes in (a) show the hypervector components required for the computation in the first stage of the reference encoding. Recall that t is the index of the iteration.

5.5 Hardware Acceleration Design

5.5.1 Acceleration Architecture

Encoding Engine The encoding procedure runs i) the element-wise addition/multiplication and ii) permutation. The parallelized implementation of the element-wise operations is straightforward, i.e., computing each dimension on different computing units. For example, if a computing platform can compute d dimensions (out of D) independently in parallel, the single operation can be calculated with $\lceil D/d \rceil$ stages. In contrast, the permutation is more challenging due to memory accesses. For example, a naive implementation may access all hypervector components from memory, but on-chip caches usually have no such capacity.

The proposed method significantly reduces the amount of memory accesses. Figure 5.4a illustrates our acceleration architecture for the initial reference encoding procedure as an example. The acceleration architecture represents typical parallel computing platforms which have many computing units and memory. As discussed in Section 5.4.1, the encoding procedures uses the permuted bipolar base hypervectors, $\mathbf{B}^{-1}, \mathbf{B}^{\perp}$ and $\mathbf{B}^{\top}$, as the inputs. Since there are four DNA alphabets, the inputs are 12 near-orthogonal hypervectors. It calculates the three intermediate hypervectors, $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ while accumulating $\mathbf{S}$ into the output reference hypervector, $\mathbf{R}$.

Consider that the permuted base hypervectors and initial reference hypervector are pre-stored in the off-chip memory. To compute all components of $\mathbf{R}$, we run the main loop of the reference encoding $\lceil D/d \rceil$ times by dividing the dimensions into multiple groups, called *chunks*. In the first iteration, the *base buffer* stores the first d components of the 12 input hypervectors (❶). The same d dimensions of $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ for the first chunk are stored in the local memory of each processing unit, e.g., registers of each GPU core (❷). For each iteration, the processing units compute the dimensions of the chunk in parallel, and accumulate to the *reference buffer* that stores the d components of $\mathbf{R}$ (❸). Then, the base buffer fetches the next elements for the 12 input hypervectors from the off-chip memory. Similarly, the reference buffer flushes its first element to the off-chip memory and reads the next element. When it needs to reset $\mathbf{R}$ for the multivector generation, the reference buffer is stored to the off-chip memory and filled with zeros. The key advantage of this method is that we do not need to know entire D components of $\mathbf{F}, \mathbf{L}$ and $\mathbf{S}$ for the permutation. Instead, we can regard that they are the d components starting from the τ-th dimension where $\tau = t \mod D$, and accumulate them in the reference buffer which already has the corresponding dimensions. Every iteration only needs to read a single element for each base and a single read/write for the reference, while fully utilizing the computing units for the HD operations. Once completing N iterations, we repeat the same procedure for the next chunk until covering all dimensions.

The similar method is generally applicable for the other procedures, the query encoding

and refinement. For example, for the query encoding, we compute each chunk of $\mathbf{Q}$ by reading an element for each base hypervector and multiplying d components.

Similarity Computation The pattern discovery engine and refinement procedures use the similarity computation. The dot product is decomposed with the element-wise multiplication and the grand sum of the multiplied components. The element-wise multiplication can be parallelized on the different computing units, and then we can compute the grand sum by adding multiple pairs in parallel with $O(\log D)$ steps. The implementation depends on the parallel platforms. We explain the details in the following section.

5.5.2 Implementation on Parallel Computing Platforms

GenieHD-GPU We implement the encoding engine by utilizing the parallel cores and different memory resources in CUDA systems (refer to Figure 5.4b.) The base buffer is stored in the constant memory, which offers high bandwidth for read-only data. Each streaming core stores the intermediate hypervector components of the chunk in their registers; the reference buffer is located in the global memory (DRAM on GPU card). The data reading and writing to the constant and global memory are implemented with CUDA streams which concurrently copy data during computations. We implement the similarity computation using the parallel reduction technique [145]. Each stream core fetches and adds multiple components into the shared memory which provide high performance for inter-thread memory accesses. We then perform the tree-based reduction in the shared memory.

GenieHD-FPGA We implement the FPGA encoding engine by using Lookup Table (LUT) resources. We store the base hypervectors into block RAMs (BRAM), the on-chip FPGA memory. The base hypervectors are loaded to a distributed memory designed by the LUT resources. Depending on the reading sequence, GenieHD loads the corresponding base hypervector and combines them using LUT resources. In the pattern discovery, we use the DSP blocks of FPGA to perform the multiplications of the dot product and a tree-based adder

to accumulate the multiplication results (refer to Figure 5.4c.) Since the query encoding and discovery use different FPGA resources, we implement the whole procedure in a pipeline structure to handle multiple queries. Depending on the FPGA available resources, it can process a different number of dimensions in parallel. For example, for Kintex-7 FPGA with 800 DSPs, we can parallelize the computation of 320 dimensions.

GenieHD-ASIC The ASIC design has three major subcomponents: SRAM, interconnect, and computing block. We used the SRAM-based memory to keep all base hypervectors. The memory is connected to the computing block with the interconnect. To reduce the memory writes to SRAM, the interconnect implements n-bit shifts to fetch the hypervector components to the computing block with a single cycle. The computing units parallelize the element-wise operations. For the query discovery, it forwards the execution results to the tree-based adder structure located in the computing block in a similar way to the FPGA design. The efficiency depends on the number of parallel computing units. We design GenieHD-ASIC with the same size of the experimented GPU core, $471mm^2$. In this setting, our implementation parallelizes the computations for 8000 components.

5.6 Evaluation

5.6.1 Experimental Setup

We evaluate GenieHD on parallel various computing platforms. We implement GenieHD-GPU on NVIDIA GTX 1080 Ti (3584 CUDA cores) and Intel i7-8700K CPU (12 multithreads) and measure power consumption using Hioki 3334 power meter. GenieHD-FPGA is synthesized on Kintex-7 FPGA KC705 using Xilinx Vivado Design Suite. We used Vivado XPower tool to estimate the device power. We design and simulate GenieHD-ASIC using RTL System-Verilog. For the synthesis, we use *Synopsys Design Compiler* with the TSMC 45 nm technology library and the general purpose process with high V_{TH} cells. We estimate the power consumption

Table 5.1: Evaluated DNA Sequence Datasets

	Description	Length	$\perp, \top$	HV size
E.Coli (MG1655)	Escherichia coli	4.6M	199,201	53MB
Human (CHR14)	Human chromosome 14	107M	99,101	1.2GB
Synthetic (RND70)	Random sequence	70M	99,101	0.8GB

using *Synopsys PrimeTime* at (1V, 25°C, TT) corner. The GenieHD family is evaluated using $D = 100,000$ and $P = 10,000$ with five refinement epochs.

Table 5.1 summarizes the evaluated DNA sequence datasets. We use E.coli DNA data (MG1655) and the human reference genome, chromosome 14 (CHR14) [144]. We also create a random synthetic DNA sequence (RND70) having a length of 70 million characters. The query sequence reads with the length in $[\perp, \top]$ are extracted using SRA toolkit from the FASTQ format. The total size of the generated hypervectors for each sequence (HV size) is linearly proportional to the length of the reference sequence. Note that state-of-the-art bioinformatics tools also have the peak memory footprint in up to two orders of gigabytes for the human genome [133].

5.6.2 Efficiency Comparison

We compare the efficiency of GenieHD with state-of-the-art programs and accelerators, i) *Bowtie2* [132] running on Intel i7-8700K CPU and ii) *minimap2* [133], which runs on the same CPU, but tens of times faster than the previous mainstream such as BLASR and GMAP, iii) GPU-based design (*ADEY*) [20], and iv) FPGA-based design (*SCADIS*) [21] evaluated on the same chip to GenieHD-FPGA. Figure 5.5 presents that GenieHD outperforms the state-of-the-art methods. For example, even though including the overhead of the offline reference encoding, GenieHD-ASIC achieves up to 16× speedup and 40× higher energy efficiency as compared to Bowtie2. GenieHD can offer higher improvements if the references are encoded in advance. For example, when the encoded hypervectors are available, by eliminating the offline encoding costs, GenieHD-ASIC is 199.7× faster and 369.9× more energy efficient than Bowtie2. When comparing the same platforms, GenieHD-FPGA (GenieHD-GPU) achieves 11.1× (10.9×)

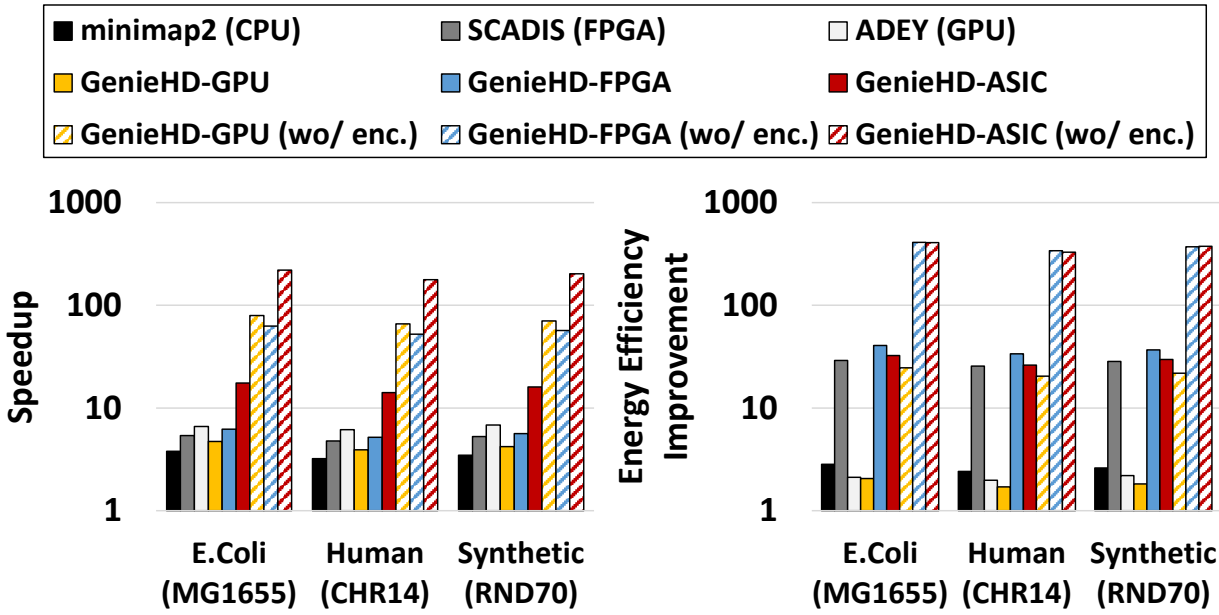


Figure 5.5: Performance and Energy Comparison of GenieHD for State-of-the-art Methods. All results are compared and normalized to Bowtie2.

speedup and 13.5× (10.6×) higher energy efficiency as compared to SCADIS running on FPGA (ADEY on the GPGPU).

5.6.3 Pattern Matching for Multiple Queries

Figure 5.6(a) shows the breakdown of the GenieHD procedures. The results show that most execution costs come from the reference encoding procedure, e.g., more than 97.6% on average. It is because i) the query sequence is relatively very short and ii) the discovery procedure examines multiple patterns using a single similarity computation in a highly parallel manner. As discussed in Section 5.3, GenieHD can *reuse* the same reference hypervectors for different queries newly sampled. Figure 5.6(b)-(d) shows the speedup of the accumulated execution time for multiple queries over the state-of-the-art counterparts. For fair comparison, we evaluate the performance of GenieHD based on the total execution costs including the reference/query encoding and query discovery engines. The results show that, by reusing the encoded reference hypervector, GenieHD achieves higher speedup as the number of queries increases. For example,

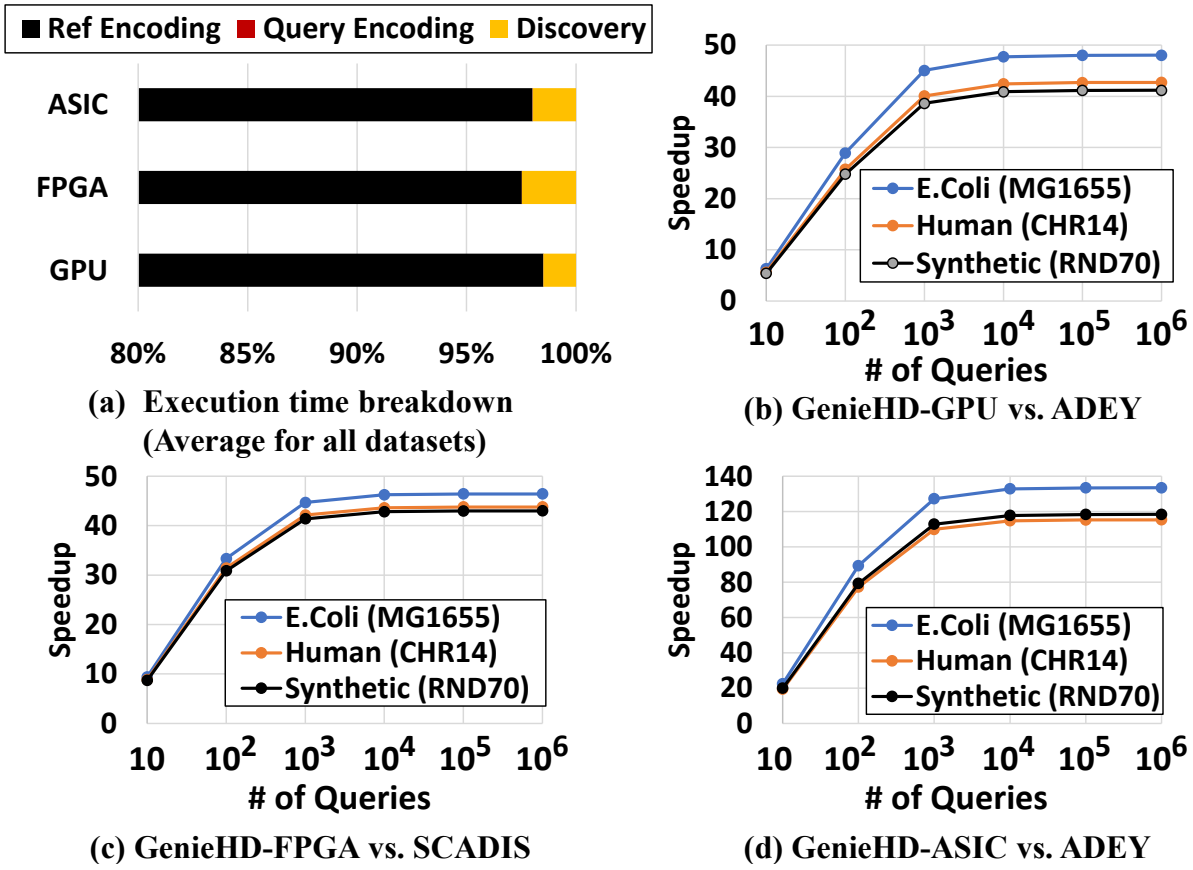


Figure 5.6: Scalability of GenieHD. (a) shows the execution time breakdown to process the single query and reference. (b)-(d) shows how the speedup changes as increasing the number of queries for a reference.

when comparing the designs running on the same platform, we observe 43.9× and 44.4× speedup on average for 10^6 queries on (b) GPU and (c) FPGA, respectively. The energy-efficiency improvement for each case is 42.5× and 54.1×. As compared to ADEY, GenieHD-ASIC offers 122× speedup and 707× energy-efficiency improvements with the same area (d). It is because GenieHD consumes much less cost from the second run. The speedup converges at around 10^3 queries as the query discovery takes a more portion of the execution time for a larger number of queries.

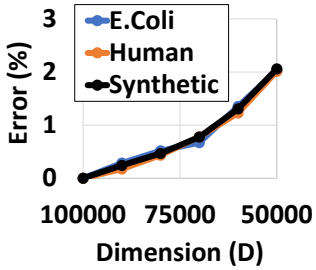


Figure 5.7: Accuracy Loss over Dimension Size

5.6.4 Dimensionality Exploitation

In practice, the higher efficiency would be more desired than the perfect discovery, since DNA sequences are often error-prone [19]. The statistical nature of GenieHD facilitates such optimization. Figure 5.7 shows how much the additional error occurs from the baseline accuracy of 0.003% as decreasing the dimensionality. As anticipated with the estimation model shown in Section 5.4.2, the error increases with a less dimensionality. Note that it does not need to encode the hypervectors again; instead, we can use only a part of components in the similarity computation. The results suggest that we can significantly improve the efficiency with minimal accuracy loss. For example, we can achieve 2× speedup for all the GenieHD family with 2% loss as it only needs the computation for half dimensions. We can also exploit this characteristic for power optimization. Table 5.2 shows the power consumption for the hardware components of GenieHD-ASIC, SRAM, interconnect (ITC), and computing block (CB) along with the throughput. We evaluated two power optimization schemes, i) **Gating** which does not use half of the resources, and ii) voltage over scaling (**VoS**) which uses all resources at a lower frequency. The frequency is set to obtain the same throughput of 640K/sec (the number of similarity computations per second.) The results show that **VoS** is the more effective method since the frequency non-linearly influences the speed. GenieHD-ASIC with **VoS** saves 49.6% and 60.6% power with accuracy loss of 1% and 2%, respectively.

Table 5.2: GenieHD-ASIC Designs under Loss

Accuracy		0%	1%		2%	
		Base	**Gating**	**VoS**	**Gating**	**VoS**
Power(W)	**SRAM**	3.4	2.5	3.4	1.8	3.4
	ITC	0.6	0.4	0.6	0.3	0.6
	CB	21.5	13.7	8.8	10.9	6.0
	Total	25.4	16.6	12.8	13.1	10.0
Power Saving (%)			34.6	49.6	48.4	60.6
Throughput		640K / sec				

5.7 Conclusion

In this chapter, we describe GenieHD algorithm, which performs the DNA pattern matching using HD computing. The proposed technique maps DNA sequences to hypervectors, and accelerates the pattern matching in a highly-parallelized way. We also show how to optimize the memory access patterns and perform pattern matching tasks with dimension-independent operations in parallel. The experimental results show that GenieHD significantly accelerates the pattern matching procedure, e.g., 44.4× speedup with 54.1× energy-efficiency improvements when comparing to the existing design on the same FPGA [21]. In the next chapter, we summarize our contributions and discuss the future work.

This chapter contains material from "GenieHD: Efficient DNA Pattern Matching Accelerator Using Hyperdimensional Computing", by Yeseong Kim, Mohsen Imani, Niema Moshiri and Tajana Rosing, which appears in IEEE/ACM Design Automation and Test in Europe Conference, March 2020. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Summary and Future Work

In the IoT ecosystems, many applications run machine learning algorithms to assimilate the data generated in the environment. However, sensors and embedded devices generate massive data streams, which poses huge technical challenges due to limited device resources. For example, although deep learning models have provided high classification accuracy for complex learning tasks, their high computational complexity and memory requirements hinder their usability for a broad variety of real-life embedded applications where the device resources and power budget are limited.

In this thesis, we focused on novel solutions which can enable efficient learning for IoT ecosystems. In general, we address the issue of the device heterogeneity with the intelligent cross-platform characterization and task allocation technique. We then utilize HD computing to enable the efficient learning solution for less-powerful resource-hungry IoT devices. Our new learning solution was applied to the hierarchy of the IoT systems, solving the concerns of the security and privacy. We lastly present that the HD-based application can cover the variety of data types that IoT devices generate. Our solution spans from the architecture level up to the hierarchy of IoT systems. The following sections summarize the contributions of this dissertation and outline the future directions.

6.1 Thesis Summary

This thesis proposes novel solutions for efficient learning in heterogeneous IoT systems, covering architecture, application, and IoT hierarchy levels. At the architecture level, we propose a cross-platform power/performance estimation technique to optimize for a given system power and performance objective of learning tasks in Chapter 2. Our technique identifies latent states of hardware and software behavior over different configurations and on different platforms that potentially exist in IoT systems. The proposed framework, P^4, a new Phase-based Power and Performance Prediction framework, enables to intelligently utilize refined data from performance counters and characterize application power consumption at runtime. We use this information for predictive task allocation for learning tasks. Unlike existing power estimation techniques, our framework automatically recognizes distinct application phases in a fine-grained level, without a priori knowledge of the program source code. The framework utilizes machine learning to identify the phases, which represent key application profiles related to system usage characteristics. Then, it performs what-if analysis to predict how application tasks behave on a different system and allocate the learning tasks among distributed systems. The proposed framework can predict the power levels of diverse applications with less than 7% error for completely different platforms from the ones applications are characterized on. The model-based task allocation technique integrated with Apach Saprk saves the energy consumption and costs by 16%.

At the application level, we show how to enable light-weight learning algorithm which are suitable for less-powerful IoT devices in Chapter 3. In IoT systems, sending all the data to the cloud cannot guarantee scalability and real-time response. It is also often undesirable due to privacy and security concerns. This leads to the need for alternative computing methods that can run a large amount of data at least partly on the less-powerful IoT devices. Brain-inspired Hyperdimensional (HD) computing is such an alternative. Our HD-based learning algorithm converts data collected from IoT sensors to hypervectors. With the hypervector, we perform

light-weight training and inference on IoT devices. The experimental results show that we can improve the learning performance by 486× and 6× for the training and inference, respectively, as compared to the state-of-the-art deep learning [63].

We apply the HD learning method to the IoT hierarchy in Chapter 4. We utilize the fact that the HD computing does not require complete knowledge for the original data that the conventional learning algorithms need. The proposed method uses with a *secure* mapping function that encodes a given data to a high-dimensional space. In that way, the original data cannot be recovered from the hypervectors without knowing the mapping function. Since the HD learning and inference procedures only depend on the encoded data, the framework performs learning tasks based on the data encoded by users, while the user data are not revealed to the cloud server. We present scalable HD-based methods which collaboratively learns diverse classification problems, a cloud-centric method for the case that end-node devices does not have enough computing capability, and an edge-based method that all the user devices participate in secure distributed learning. In our evaluation, we show that the proposed framework can perform the encoding and decoding tasks 145.6× and 6.8× faster than a state-of-the-art homomorphic encryption library when both are running on the Intel i7-8700K. In addition, our classification method presents high accuracy and scalability for diverse practical problems. It successfully performs learning tasks with 95% average accuracy for six real-world workloads, ranging from datasets collected in a small IoT network, e.g., human activity recognition, to a large dataset which includes hundreds of thousands of images for the face recognition task.

In Chapter 4, we examine the potential of HD computing-based learning for other data types, such as in bioinformatics applications, with a focus on DNA pattern matching. GenieHD converts the DNA sequences into the hypervector databases to effectively parallelize the pattern matching task. We also show optimization techniques to implement the algorithm on various platforms. The experimental results show that GenieHD outperforms the state-of-the-art DNA pattern matching procedures on various platforms, GPU [20] and FPGA [21]. For example,

the proposed design delivers 44.1× speedup with 54.1× energy-efficiency improvements when implemented on the same FPGA.

6.2 Future Directions

IoT systems and applications continue to evolve while introducing new problems such as automated online learning and edge-based computing. We are actively working to enable new learning solutions based on the HD computing. In this section, we provide future directions for further improvement on the topics discussed in this thesis: hyperdimensional processing system and software infrastructure for the HD computing.

6.2.1 Efficient Cognitive Processing with HD Computing

HD computing is an attractive solution for efficient learning. This thesis showed that many classification problems can be efficiently solved using HD computing. HD computing-based classification drastically reduces the number of operations as compared to deep learning, resulting in improved performance and high energy efficiency [80, 15]. We can also efficiently accelerate on a parallel computing platform since the hypervector operations are at heart parallelizable.

As the future work, we plan to integrate HD computing into todays machines to support other learning tasks. In this system, the hypervector and related operations will be supported as native data types and instructions so that user-level programs could implement diverse learning solutions with an augmented programming model. We also plan to design a co-processor, called hyperdimensional processing unit (HPU), using emerging hardware technology. In particular, we consider in-memory processing technology to (i) reduce the data movement overheads of hypervectors betwewen processing core and memory and (ii) drastically parallelize the HD operations such as the element-wise addition and multiplication, similarity computation, etc. The interface between CPU and HPU could be implemented based on existing technologies, e.g., PCI

express and non-uniform memory architecture (NUMA).

6.2.2 Software Infrastructure for HD Computing

Implementing the HD application from scratch is difficult as there is no software infrastructure. Current testbeds are implementd with existing libraries such as Scikit-learn [62] and Tensorflow [63] with various manual optimizations of the source code for different hardware. The ideal software library should bridge the hardware and applications so that such optimizations are automatically done for different applications and platforms.

Our current plan is the design of a software infrastructure which has two components: HD algorithm package and native HD compiler. The algorithm package will include various learning algorithms using HD computing. Currently, we are designing a new class of learning algorithm suites such as regression and reinforcement learning. They will be implemented in Python that is similar to existing machine learning libraries for better usability. The compiler translates the Python implementation to generate HD operations suitable for the target hardware such as GPU and FPGA. The compiler will also optimize the source code in an automated way. For example, to optimize memory usage, the compiler will consider the different characteristics of used hypervectors. If a set of hypervectors is frequently used in applications, our compiler identifies such hypervectors in the static program analysis so that we can proactively allocate the hypervector to the faster memory area in the hardware accelerator, e.g., performance-efficient cache.

Bibliography

[1] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: The evolution of data to life-critical don't focus on big data," *Framingham: IDC Analyze the Future*, 2017.

[2] I. Lee and K. Lee, "The internet of things (iot): Applications, investments, and challenges for enterprises," *Business Horizons*, vol. 58, no. 4, pp. 431–440, 2015.

[3] T. Yashiro, S. Kobayashi, N. Koshizuka, and K. Sakamura, "An internet of things (iot) architecture for embedded appliances," in *2013 IEEE Region 10 Humanitarian Technology Conference*. IEEE, 2013, pp. 314–319.

[4] X. Zheng, L. K. John, and A. Gerstlauer, "Lacross: Learning-based analytical cross-platform performance and power prediction," *International Journal of Parallel Programming*, vol. 45, no. 6, pp. 1488–1514, 2017.

[5] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017, pp. 1–12.

[6] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[7] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[8] Z. Ou, B. Pang, Y. Deng, J. K. Nurminen, A. Yla-Jaaski, and P. Hui, "Energy-and cost-efficiency analysis of arm-based clusters," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 115–123.

[9] W. Cui, Y. Kim, and T. S. Rosing, "Cross-platform machine learning characterization for task allocation in iot ecosystems," in *Computing and Communication Workshop and Conference (CCWC), 2017 IEEE 7th Annual*. IEEE, 2017, pp. 1–7.

[10] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade, "Decomposable and responsive power models for multicore processors using performance counters," in *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 2010.

[11] R. Bertran, M. Gonzàlez, X. Martorell, N. Navarro, and E. Ayguadé, "Counter-based power modeling methods: Top-down vs. bottom-up," *The Computer Journal*, 2013.

[12] S. Reda and A. N. Nowroz, "Power modeling and characterization of computing devices: a survey," *Foundations and Trends in Electronic Design Automation*, 2012.

[13] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.

[14] B. Babadi and H. Sompolinsky, "Sparseness and expansion in sensory representations." *Neuron*, vol. 83, no. 5, pp. 1213–1226, Sep. 2014. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/25155954

[15] Y. Kim, M. Imani, and T. S. Rosing, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, 2018, pp. 1–6.

[16] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2010, pp. 24–43.

[17] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine, "Simple encrypted arithmetic library v2.3.0," in *Microsoft*, 2017.

[18] M. M. Alam, H. Malik, M. I. Khan, T. Pardy, A. Kuusik, and Y. Le Moullec, "A survey on the roles of communication technologies in iot-based personalized healthcare applications," *IEEE Access*, vol. 6, pp. 36 611–36 631, 2018.

[19] P. Compeau and P. Pevzner, *Bioinformatics algorithms: an active learning approach*. Active Learning Publishers La Jolla, 2015, vol. 1.

[20] S. P. Adey, "Gpu accelerated pattern matching algorithm for dna sequences to detect cancer using cuda," *hgpu*, 2013.

[21] S. Lei, C. Wang, H. Fang, X. Li, and X. Zhou, "Scadis: A scalable accelerator for data-intensive string set matching on fpgas," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 1190–1197.

[22] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, "Fog computing: A platform for internet of things and analytics," in *Big Data and Internet of Things: A Roadmap for Smart Environments*. Springer, 2014, pp. 169–186.

[23] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges," *arXiv preprint arXiv:1006.0308*, 2010.

[24] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010.

[25] M. Zaharia, R. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: a unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[26] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 267–288, 1996.

[27] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-system power analysis and modeling for server environments," in *International Symposium on Computer Architecture-IEEE*, 2006.

[28] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[29] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006.

[30] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001.

[31] G. Tang, W. Jiang, Z. Xu, F. Liu, and K. Wu, "Zero-cost, fine-grained power monitoring of datacenters using non-intrusive power disaggregation," in *Proceedings of the 16th Annual Middleware Conference*. ACM, 2015.

[32] S. S. Bhargav, A. Kolb, and Y. H. Cho, "Accelerating physical level sub-component power simulation by online power partitioning," in *2016 17th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2016.

[33] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009.

[34] Y.-H. Lee and J. Kim, "Fast and accurate on-line prediction of performance and power consumption in multicore-based systems," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013.

[35] V. Petrucci, O. Loques, and D. Mosse, "Lucky scheduling for energy-efficient heterogeneous multi-core systems," in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems (HotPower)*, 2012.

[36] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf*, 2011.

[37] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[38] Y. Kim, F. Parterna, S. Tilak, and T. S. Rosing, "Smartphone analysis and optimization based on user activity recognition," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. IEEE, 2015.

[39] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, "edoctor: automatically diagnosing abnormal battery drain issues on smartphones," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[40] Y. Jin, X. Ma, M. Liu, Q. Liu, J. Logan, N. Podhorszki, J. Y. Choi, and S. Klasky, "Combining phase identification and statistic modeling for automated parallel benchmark generation," *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[41] A. Annamalai, R. Rodrigues, I. Koren, and S. Kundu, "An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE Press, 2013.

[42] B. Aksanli and T. Rosing, "Providing regulation services and managing data center peak power budgets," in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014.

[43] L. Luo, W. Wu, D. Di, F. Zhang, Y. Yan, and Y. Mao, "A resource scheduling algorithm of cloud computing based on energy efficient optimization methods," in *Green Computing Conference (IGCC), 2012 International*. IEEE, 2012, pp. 1–6.

[44] Í. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini, "Parasol and greenswitch: Managing datacenters powered by renewable energy," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 51–64.

[45] M. Dayarathna, Y. Wen, and R. Fan, "Data center energy consumption modeling: A survey," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 732–794, 2016.

[46] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future generation computer systems*, vol. 28, no. 5, pp. 755–768, 2012.

[47] I. Mavridis and H. Karatza, "Performance evaluation of cloud-based log file analysis with apache hadoop and apache spark," *Journal of Systems and Software*, vol. 125, pp. 133–151, 2017.

[48] P. Li, L. Dong, H. Xu, and T. F. Lau, "Spark's operation time predictive in cloud computing environment based on src-wsvr," *Journal of High Speed Networks*, vol. 24, no. 1, pp. 49–62, 2018.

[49] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang, "Ppep: Online performance, power, and energy prediction framework and dvfs space exploration," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014.

[50] J. Reinders, *VTune (TM) Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Intel Press, 2004.

[51] T. S. Shively, T. W. Sager, and S. G. Walker, "A bayesian approach to non-parametric monotone function estimation," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 71, no. 1, pp. 159–175, 2009.

[52] C. S. Chan, Y. Jin, Y.-K. Wu, K. Gross, K. Vaidyanathan, and T. Rosing, "Fan-speed-aware scheduling of data intensive jobs," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*. ACM, 2012, pp. 409–414.

[53] "Linpack benchmarks," https://software.intel.com/en-us/articles/intel-mkl-benchmarks-suite, 2019.

[54] D. Arthur and S. Vassilvitskii, "k-means++: The advantages of careful seeding," in *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2007.

[55] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a kneedle in a haystack: Detecting knee points in system behavior," in *2011 31st International Conference on Distributed Computing Systems Workshops*. IEEE, 2011.

[56] "Spec2006," https://www.spec.org/cpu2006/, 2006.

[57] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008.

[58] D. Birant and A. Kut, "St-dbscan: An algorithm for clustering spatial–temporal data," *Data & Knowledge Engineering*, vol. 60, no. 1, pp. 208–221, 2007.

[59] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's data center-wide power management system," in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 2016, pp. 469–480.

[60] S. Govindan, A. Sivasubramaniam, and B. Urgaonkar, "Benefits and limitations of tapping into stored energy for datacenters," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 341–351.

[61] B. Aksanli, J. Venkatesh, I. Monga, and T. S. Rosing, "Renewable energy prediction for improved utilization and efficiency in datacenters and backbone networks," in *Computational Sustainability*. Springer, 2016, pp. 47–74.

[62] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[63] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[64] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[65] "Weave net," https://www.weave.works/, 2019.

[66] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee, 2010, pp. 1–10.

[67] "Wikipedia, mean absolute percentage error," https://en.wikipedia.org/wiki/Mean_absolute_percentage_error/, 2019.

[68] "Nersc benchmarks," http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/trinity-nersc-8-rfp/nersc-8-trinity-benchmarks/, 2015.

[69] "Top 500 supercomputer sites," https://www.top500.org/, 2019.

[70] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura, "Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark," in *Proceedings of the 12th ACM International Conference on Computing Frontiers*. ACM, 2015, p. 53.

[71] J. J. Dai, Y. Wang, X. Qiu, D. Ding, Y. Zhang, Y. Wang, X. Jia, C. L. Zhang, Y. Wan, Z. Li, J. Wang, S. Huang, Z. Wu, Y. Wang, Y. Yang, B. She, D. Shi, Q. Lu, K. Huang, and G. Song, "Bigdl: A distributed deep learning framework for big data," *arXiv preprint arXiv:1804.05839*, 2018.

[72] D. Capps and W. Norcott, "Iozone filesystem benchmark," 2008.

[73] A. Tirumala, "Iperf: The tcp/udp bandwidth measurement tool," *http://dast. nlanr. net/Projects/Iperf/*, 1999.

[74] T. P. P. Council, "Tpc-h benchmark specification," *Published at http://www. tcp. org/hspec. html*, vol. 21, pp. 592–603, 2008.

[75] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proceedings of the 2015 workshop on mobile big data*. ACM, 2015, pp. 37–42.

[76] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server." in *OSDI*, vol. 14, 2014, pp. 583–598.

[77] J. Andriessen, M. Baker, and D. D. Suthers, *Arguing to learn: Confronting cognitions in computer-supported collaborative learning environments*. Springer Science & Business Media, 2013, vol. 1.

[78] A. Papadimitriou, R. Bhagwan, N. Chandran, R. Ramjee, A. Haeberlen, H. Singh, A. Modi, and S. Badrinarayanan, "Big data analytics over encrypted datasets with seabed." in *OSDI*, 2016, pp. 587–602.

[79] P. Kanerva, "Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors," *Cognitive Computation*, vol. 1, no. 2, pp. 139–159, 2009.

[80] M. Imani, D. Kong, A. Rahimi, and T. Rosing, "Voicehd: Hyperdimensional computing for efficient speech recognition," in *International Conference on Rebooting Computing (ICRC)*. IEEE, 2017, pp. 1–6.

[81] A. Joshi, J. Halseth, and P. Kanerva, "Language geometry using random indexing," *Quantum Interaction 2016 Conference Proceedings*, In press.

[82] M. Imani, A. Rahimi, D. Kong, T. Rosing, and J. M. Rabaey, "Exploring hyperdimensional associative memory," in *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 445–456.

[83] Y. Kim, F. Parterna, S. Tilak, and T. S. Rosing, "Smartphone analysis and optimization based on user activity recognition," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. IEEE, 2015, pp. 605–612.

[84] B. Aksanli, A. S. Akyurek, and T. S. Rosing, "User behavior modeling for estimating residential energy consumption," in *Smart City 360*. Springer, 2016, pp. 348–361.

[85] A. Rahimi, A. Tchouprina, P. Kanerva, J. d. R. Millán, and J. M. Rabaey, "Hyperdimensional computing for blind and one-shot classification of eeg error-related potentials," *Mobile Networks and Applications*, pp. 1–12, 2017.

[86] F. R. Najafabadi, A. Rahimi, P. Kanerva, and J. M. Rabaey, "Hyperdimensional computing for text classification," *Design, Automation Test in Europe Conference Exhibition (DATE), University Booth*, 2016.

[87] O. Räsänen and S. Kakouros, "Modeling dependencies in multiple parallel data streams with hyperdimensional computing," *IEEE Signal Processing Letters*, vol. 21, no. 7, pp. 899–903, 2014.

[88] O. Rasanen and J. Saarinen, "Sequence prediction with sparse distributed hyperdimensional coding applied to the analysis of mobile phone use patterns," *IEEE Transactions on Neural Networks and Learning Systems*, vol. PP, no. 99, pp. 1–12, 2015.

[89] A. Rahimi, S. Benatti, P. Kanerva, L. Benini, and J. M. Rabaey, "Hyperdimensional biosignal processing: A case study for emg-based hand gesture recognition," in *Rebooting Computing (ICRC), IEEE International Conference on*. IEEE, 2016, pp. 1–8.

[90] H. Li, T. F. Wu, A. Rahimi, K. Li, M. Rusch, C. Lin, J. Hsu, M. M. Sabry, S. B. Eryilmaz, J. Sohn, W. Chiu, M. Chen, T. Wu, J. Shieh, W. Yeh, J. M. Rabaey, S. Mitra, and H. P. Wong, "Hyperdimensional computing with 3d vrram in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition," in *Electron Devices Meeting (IEDM), 2016 IEEE International*. IEEE, 2016, pp. 16–1.

[91] A. Reiss and D. Stricker, "Introducing a new benchmarked dataset for activity monitoring," in *Wearable Computers (ISWC), 2012 16th International Symposium on*. IEEE, 2012, pp. 108–109.

[92] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones." in *ESANN*, 2013.

[93] Y. Vaizman, K. Ellis, and G. Lanckriet, "Recognizing detailed human context in the wild from smartphones and smartwatches," *IEEE Pervasive Computing*, vol. 16, no. 4, pp. 62–74, 2017.

[94] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 113–126.

[95] M. Imani, C. Huang, D. Kong, and T. Rosing, "Hierarchical hyperdimensional computing for energy efficient classification," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, p. 108.

[96] F. Paterna and T. Š. Rosing, "Modeling and mitigation of extra-soc thermal coupling effects and heat transfer variations in mobile devices," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. IEEE Press, 2015, pp. 831–838.

[97] S. Suthaharan, "Big data classification: Problems and challenges in network intrusion prediction with machine learning," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 4, pp. 70–73, 2014.

[98] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 2, pp. 1153–1176, 2016.

[99] N. Papernot, P. McDaniel, A. Sinha, and M. Wellman, "Towards the science of security and privacy in machine learning," *arXiv preprint arXiv:1611.03814*, 2016.

[100] M. I. Jordan and T. M. Mitchell, "Machine learning: Trends, perspectives, and prospects," *Science*, vol. 349, no. 6245, pp. 255–260, 2015.

[101] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[102] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019.

[103] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, p. 8, 2015.

[104] G. Zhao, C. Rong, J. Li, F. Zhang, and Y. Tang, "Trusted data sharing over untrusted cloud storage providers," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 97–103.

[105] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "Sporc: Group collaboration using untrusted cloud resources." in *OSDI*, vol. 10, 2010, pp. 337–350.

[106] S. Choi, G. Ghinita, H.-S. Lim, and E. Bertino, "Secure knn query processing in untrusted cloud environments," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2818–2831, 2014.

[107] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *CCS*. ACM, 2017.

[108] A. Ben-Efraim, Y. Lindell, and E. Omri, "Optimizing semi-honest secure multiparty computation for the internet," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 578–590.

[109] A. Rahimi, P. Kanerva, and J. M. Rabaey, "A robust and energy-efficient classifier using brain-inspired hyperdimensional computing," in *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ACM, 2016, pp. 64–69.

[110] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library-seal v2. 1," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 3–18.

[111] W. Du and M. J. Atallah, "Secure multi-party computation problems and their applications: a review and open problems," in *Proceedings of the 2001 workshop on New security paradigms*. ACM, 2001, pp. 13–22.

[112] M. S. Riazi, C. Weinert, O. Tkachenko, E. M. Songhori, T. Schneider, and F. Koushanfar, "Chameleon: A hybrid secure computation framework for machine learning applications," in *ASIACCS*. ACM, 2018.

[113] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *CCS*. ACM, 2015.

[114] M. S. Riazi and F. Koushanfar, "Privacy-preserving deep learning and inference," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 18.

[115] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning." in *IEEE S&P*, 2017.

[116] M. S. Riazi, B. D. Rouhani, and F. Koushanfar, "Deep learning on private data," in *IEEE Security and Privacy Magazine*. IEEE, 2019.

[117] M. S. Riazi, M. Samragh, H. Chen, K. Laine, K. Lauter, and F. Koushanfar, "XONN: XNOR-based oblivious deep neural network inference," *USENIX Security*, 2019.

[118] B. Hitaj, G. Ateniese, and F. Pérez-Cruz, "Deep models under the GAN: information leakage from collaborative deep learning," in *CCS*. ACM, 2017.

[119] A. C.-C. Yao, "How to generate and exchange secrets," in *Foundations of Computer Science, 1986., 27th Annual Symposium on*. IEEE, 1986, pp. 162–167.

[120] V. Kolesnikov and T. Schneider, "Improved garbled circuit: Free XOR gates and applications," in *Automata, Languages and Programming*. Springer, 2008.

[121] A. Waksman, "A permutation network," *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[122] P. Kanerva, J. Kristofersson, and A. Holst, "Random indexing of text samples for latent semantic analysis," in *Proceedings of the 22nd annual conference of the cognitive science society*, vol. 1036. Citeseer, 2000.

[123] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, "Network simulations with the ns-3 simulator," *SIGCOMM demonstration*, vol. 14, no. 14, p. 527, 2008.

[124] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[125] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[126] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 3642–3649.

[127] "Uci machine learning repository," http://archive.ics.uci.edu/ml/datasets/ISOLET, 1994.

[128] M. S. Razlighi, M. Imani, F. Koushanfar, and T. Rosing, "Looknn: Neural network with no multiplication," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1775–1780.

[129] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine," in *International workshop on ambient assisted living*. Springer, 2012, pp. 216–223.

[130] Y. Kim, M. Imani, and T. Rosing, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *Computer-Aided Design (ICCAD), 2017 IEEE/ACM International Conference on*. IEEE, 2017, pp. 25–32.

[131] U. Quasthoff, M. Richter, and C. Biemann, "Corpus portal for search in monolingual corpora," in *Proceedings of the fifth international conference on language resources and evaluation*, vol. 17991802, 2006, p. 21.

[132] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with bowtie 2," *Nature methods*, vol. 9, no. 4, p. 357, 2012.

[133] H. Li, "Minimap2: pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 34, no. 18, pp. 3094–3100, 2018.

[134] A. Rahimi, S. Datta, D. Kleyko, E. P. Frady, B. Olshausen, P. Kanerva, and J. M. Rabaey, "High-dimensional computing as a nanoscalable paradigm," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2017.

[135] M. Imani, Y. Kim, S. Riazi, J. Messerly, P. Liu, F. Koushanfar, and T. Rosing, "A framework for collaborative learning in secure high-dimensional space," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 435–446.

[136] M. Imani, J. Morris, J. Messerly, H. Shu, Y. Deng, and T. Rosing, "Bric: Locality-based encoding for energy-efficient brain-inspired hyperdimensional computing," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[137] M. Imani, T. Nassar, A. Rahimi, and T. Rosing, "Hdna: Energy-efficient dna sequencing using hyperdimensional computing," in *IEEE BHI*. IEEE, 2018, pp. 271–274.

[138] M. Imani, S. Salamat, S. Gupta, J. Huang, and T. Rosing, "Fach: Fpga-based acceleration of hyperdimensional computing by reducing computational complexity," in *ASP-DAC*. IEEE, 2019.

[139] M. Imani, S. Salamat, B. Khaleghi, M. Samragh, F. Koushanfar, and T. Rosing, "Sparsehd: Algorithm-hardware co-optimization for efficient high-dimensional computing," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 190–198.

[140] S. Salamat, M. Imani, B. Khaleghi, and T. Rosing, "F5-hd: Fast flexible fpga-based framework for refreshing hyperdimensional computing," in *FPGA*. ACM, 2019, pp. 53–62.

[141] M. Imani, J. Messerly, F. Wu, W. Pi, and T. Rosing, "A binary learning framework for hyperdimensional computing," in *DATE*. IEEE/ACM, 2019.

[142] S. Memeti and S. Pllana, "Analyzing large-scale dna sequences on multi-core architectures," in *2015 IEEE 18th International Conference on Computational Science and Engineering*. IEEE, 2015, pp. 208–215.

[143] E. B. Fernandez, W. A. Najjar, S. Lonardi, and J. Villarreal, "Multithreaded fpga acceleration of dna sequence mapping," in *2012 IEEE Conference on High Performance Extreme Computing*. IEEE, 2012, pp. 1–6.

[144] "National center for biotechnology information support center," https://www.ncbi.nlm.nih.gov, 2019.

[145] M. Harris, "Optimizing parallel reduction in cuda," *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.