MACHINE LEARNING-BASED SOLUTIONS FOR SECURE AND ENERGY-EFFICIENT COMPUTER SYSTEMS

by

Hossein Sayadi
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Electrical and Computer Engineering

Committee:

____________________	Dr. Houman Homayoun, Dissertation Director
____________________	Dr. Setareh Rafatirad, Co-director
____________________	Dr. Jim Jones, Committee Member
____________________	Dr. Avesta Sasan, Committee Member
____________________	Dr. Monson H. Hayes, Department Chair
____________________	Dr. Kenneth S. Ball, Dean, The Volgenau School of Engineering
Date: ________________	Summer Semester 2019 George Mason University Fairfax, VA

Machine Learning-Based Solutions for Secure and Energy-Efficient Computer Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

By

Hossein Sayadi
Master of Science
Sharif University of Technology, 2014
Bachelor of Science
K. N. Toosi University of Technology, 2012

Director: Dr. Houman Homayoun, Associate Professor
Co-director: Dr. Setareh Rafatirad, Associate Professor
Department of Electrical and Computer Engineering

Summer Semester 2019
George Mason University
Fairfax, VA

Dedication

This thesis is dedicated to my wife, Tahereh, who has been a constant source of support and encouragement during the challenges of graduate school and life. This work is further dedicated to my parents, who have always loved me unconditionally. I am enormously grateful and indebted to them for their continuous love and support.

Acknowledgments

First and foremost, I would like to take this opportunity to express my deepest gratitude from my supervisor Dr. Houman Homayoun for all his extensive guidance, advice, and continuous support during my doctoral studies. In particular, I am truly grateful to Houman for providing me valuable and insightful research opportunities, for his patience and understanding throughout the thesis that has set an example of excellence as a professional researcher and mentor for me. In addition, I would like to thank my co-advisor Dr. Setareh Rafatirad for all her support and valuable advice and feedbacks towards the completion of my thesis. I cannot thank Houman and Setareh enough for all influential insights and valuable experiences that I have learnt from them, for their attention to details and in-depth knowledge that has helped me grow as a researcher, and for their encouragement, concern, and interest towards my success.

I would also like to thank my other committee members, Dr. Avesta Sasan and Dr. Jim Jones for their insightful comments and feedbacks which have helped me address the research questions in a much broader aspect. Furthermore, I want to express my gratitude from Dr. Paolo Costa for his time and support giving me valuable advice and insights about my path towards the academic career.

My special gratitude to my family, particularly my Mom and Dad for their unconditional love and constant support. The holidays would have been so lonely without them. I am enormously grateful and indebted to my lovely parents for their kindness, dedication, and the education they offered me, and for having always supported me, believed in me and encouraged me to pursue my life-long dreams and goals.

Last but not least, a heartfelt gratitude and appreciation go out to my lovely wife, Tahereh, for all her love, constant support, understanding, and patience during the challenges of graduate school and life. You have always been my constant source of support and encouragement during all days and nights of research and hard work in my academic education. You were always there caring for me making all the stressful days and nights full of joy and memory. Tahereh, with out your unconditional love, encouragement, and support my academic achievements and completion of this research would not have been possible.

Table of Contents

List of Tables

List of Figures

Abstract

MACHINE LEARNING-BASED SOLUTIONS FOR SECURE AND ENERGY-EFFICIENT COMPUTER SYSTEMS

Hossein Sayadi, PhD

George Mason University, 2019

Dissertation Director: Dr. Houman Homayoun

Dissertation Co-director: Dr. Setareh Rafatirad

The ever-increasing complexity of modern computing systems results in the growth of security vulnerabilities, making such systems appealing targets for increasingly sophisticated cyber attacks. The recent proliferation of computing devices in embedded systems and Internet-of-Things domains has further exacerbated the impact of cyber attacks calling for effective detection techniques. In this work, we attempt to describe how Machine Learning (ML) techniques and applications run-time information at the hardware-level can be effectively used to address major challenges of detecting emerging attacks. In response to the latency and inefficiencies of software-based malware detection techniques, Hardware-assisted Malware Detection (HMD) has emerged as a promising solution to enhance the security of computing systems. HMD techniques rely on ML classifiers to detect patterns of malicious applications based on low-level microarchitectural features captured by processors Hardware Performance Counters (HPCs) during execution.

In this work, we propose effective machine learning-based approaches using low-level HPC information to address the security and energy-efficiency challenges of the modern computer systems. For the purpose of security enhancement, four key challenges to realize an effective run-time hardware-assisted malware detection are identified and addressed. These challenges include: 1) the type of key microarchitectural events to capture at run-time which varies across various malware classes; 2) no unique ML classifier achieves high malware detection rate across various types of malware; 3) the number of available HPC registers that can be monitored simultaneously is very limited in modern microprocessors; and 4) traditional ML-based solutions fail to detect the malware accurately when the attack is embedded in a benign application, as the microarchitectural data is polluted by both malware and benign applications data. Our comprehensive analysis shows that all of these influencing parameters highly depend on the class of malware and change across various malware classes (Virus, Rootkit, Backdoor, and Trojan), i.e. the ML classifier and the type of events to collect at run-time out of many microarchitectural events that deliver the highest detection rate and performance, highly depend on the class of malware. For each of these challenges, effective machine learning-based solutions are proposed to accurately detect malware at run-time. The experimental results for the proposed run-time HMD techniques show that the malware can be detected with 98.9% detection rate at run-time with limited available HPC resources, matching to almost what can be achieved offline having access to all microarchitectural data.

Furthermore, for the last part of this research, in order to address the energy-efficiency challenges, we focus on the suitability of deploying effective machine learning techniques on run-time HPC-based information for addressing the performance vs. power consumption trade-offs and enhancing the energy-efficiency of modern heterogeneous multicore architectures. In overall, this research is primarily focused on developing highly accurate and complexity-aware machine learning-based solutions for

security and energy-efficiency enhancement of modern computer architectures based on the application's microarchitectural events captured at run-time.

As a result, the outcome of this research opens a path for computer architects and embedded systems designers in making appropriate and efficient architectural decisions for implementing future generation of computer systems, to most effectively improve the performance of machine learning algorithms for different optimization goals such as security and energy-efficiency of computer systems for emerging applications.

Chapter 1: Introduction

Electronic system security, trust and reliability has become an increasingly critical area of concern for modern society [1–5]. Secure hardware systems, platforms, as well as supply chains are critical to industry and government sectors such as national defense, healthcare, transportation, and financial [6, 7]. Traditionally, authenticity and integrity of data has been protected with various security protocol at the software level with the underlying hardware assumed to be secure, and reliable. This assumption however is no longer true with an increasing number of attacks reported on the hardware. The ever-increasing complexity of modern computing systems has led into the considerable evolution of security vulnerabilities, making such systems appealing targets for sophisticated cyber attacks. With the advent of hardware vulnerabilities and evolution of sophisticated cyber attacks, the attention of computer systems and security researchers have shifted towards working on the *Hardware Cybersecurity* research domain and improving the security of computer systems using the hardware features and characteristics [1, 2, 8–12].

Methods of Machine Learning (ML), which build predictive models that generalize training data, have proven to be useful for detecting the behavior of applications [1,13,14]. Machine learning techniques are used in a wide variety of applications, such as computer vision, robotics, design space exploration, and performance evaluation of computer systems where it is infeasible to develop an algorithm of specific instructions for performing the task [15, 16]. Machine learning is basically closely related to computational statistics, which focuses on making predictions using computers.

In this thesis, we primarily attempt to describe how machine learning techniques and applications run-time information at the microarchitectural and hardware level can be effectively deployed to address major challenges of security and energy-efficiency improvement of modern computer systems. In particular, in this research we make use of various machine learning techniques and applications run-time signatures left on the underlying hardware for the purpose of secure and energy-efficient computer architecture. To this aim, we propose effective machine learning-based solutions which rely on the run-time traces collected from hardware events registers that are built in modern microprocessors. In order to clearly highlight the scope and different categories of research performed in this thesis, in following we briefly describe the general overview of performed researches in this work that all will be presented in the upcoming sections of this thesis.

1.1 Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection

Malware is a piece of program developed by cyber-attackers to perform various malicious activities, such as destroying the data, stealing information, running destructive or intrusive programs on devices to perform Denial-of-Service (DoS) attack, and gaining root access without the consent of user [1,2,10,13,17–19]. Malware detection can be simplified as a binary classification problem regardless of what detection method is being used. Traditional malware detection approaches such as signature-based detection and semantics-based anomaly detection are considered as software-based solutions and incur significant computational overheads.

On the other hand, recent studies have demonstrated that malicious software behavior can be differentiated from benign applications by classifying anomalies in the low-level feature spaces such as microarchitectural events collected by Hardware Performance Counter (HPC) registers [1, 2, 10, 13, 20–24]. As a result, malware detection using HPCs microarchitectural events has emerged as a promising alternative to traditional malware detection methods. As learning the underlying patterns of these microarchitectural events can aid in detecting malware, machine learning (ML) techniques are widely deployed for malware detection.

Recently, there has been a number of work on Hardware-based Malware Detection (HMD) using HPCs information. However, these works performed a limited study on malware classification accounting for the availability of a large number (e.g. 16 or 32) and diverse type of HPCs. While, modern processors in the high-performance domain have a small number of HPCs (2 to 8), due to several reasons including the design complexity and cost of concurrent monitoring of microarchitectural events [1, 25, 26]. Due to deep pipelines, complex prefetchers, branch predictors, modern cache design etc., HPCs implementation becomes a great challenge in terms of counting multiple events and maintaining counter accuracy at the same time under speculative execution [26]. Better accuracy requires better and more complex hardware design hence increasing the number of counters with limited accuracy doesn't appear to be a good trade-off.

Therefore, collecting a variety of microarchitectural events, more than the number of available HPCs, to achieve high accuracy using the general ML models presented in prior work, requires running the application multiple times, since the hardware can only count a small subset of events concurrently. This approach is not practical for run-time detection of malware.

As the performance of malware detection depends on the type of ML classifier applied and the number and type of HPC events used, in this work, we first illustrate the impact of ML classifier type on malware detection accuracy and performance and the effect of number of HPC events for malware detection. To achieve a high accuracy across all studied general ML classifiers, of more than 80%, at least 16 hardware performance counters are required, which as discussed is not available in modern processors, even in the high-performance domain, making run-time detection of malware impractical using these methods. Therefore, a key challenge in making the hardware-based malware detection a practical run-time solution is how to use a limited number of HPCs available in a microprocessor (for instance 2 or 4) and match the accuracy and performance of malware detection with the ones that can be achieved by a larger number of HPC events (for instance 16 or 32).

In this work, we address this challenge by proposing effective ensemble learning techniques to improve the accuracy and performance of the hardware-based malware detectors and break the trade-off between accuracy/performance with respect to the number of HPCs. We explore the effectiveness of ensemble learning models in 1) reducing the number of required performance counters for implementing effective ML classifiers for run-time malware detection and 2) improving the performance of weak but low-cost classifiers in malware detection with a small number of HPCs.

1.2 A Two-Stage Machine Learning-Based Approach for Run-Time Specialized Malware Detection

The ever-increasing complexity of modern computing systems results in the growth of security vulnerabilities, making such systems appealing targets for increasingly sophisticated attacks [2, 11, 12, 27]. The attackers take advantage of vulnerabilities to compromise systems and deploy malicious activities. According to a 2018 McAfee threats report [28], nearly 63 million new malware samples have been recorded in the third quarter of 2018, an all-time highest number with an increase of more than 53%

from the second quarter of 2018. The recent proliferation of computing devices in mobile and Internet-of-Things (IoT) domains further exacerbates the malware threats calling for effective malware detection solutions.

While previous studies on hardware-assisted malware detection focus on one or few general ML classifiers and limited classes of malware [20–23, 29–32], it is not clear which of the ML models deliver the best results across various metrics including the detection rate, performance, hardware design overheads as well as detection delay across various classes of malware. In addition to the drawbacks of existing approaches and non-portability concerns, in this section, we will identify and address major challenges to realize an effective run-time hardware-assisted malware detection including 1) determining the key microarchitectural events for effective malware detection, 2) the right machine learning deployed for HMD, and 3) malware detection with limited available HPC resources. For each of these challenges, effective machine learning-based solutions are proposed to accurately detect malware at run-time.

The objective of this research is to improve the detection rate and performance of malware detection for different malware classes using a limited number of microarchitectural events equal to the available number of HPCs that can be captured at run-time. We propose a two-stage machine learning-based hardware-assisted malware detection approach, referred as *2SMaRT*, to not only effectively distinguish the malware from benign applications, but also to identify the class of malware at run-time using proper low-level features and determine right machine learning model for capturing the malware behavior.

1.3 Stealthy Malware Detection using Microarchitectural Features

Alongside with the advancement in the hardware-based malware detection techniques, malicious software attacks have continued to evolve in quantity and sophistication during the past decade. Due to increasing complexity of malware attacks and financial motivations of attackers, malware trends are recently shifting towards a more dangerous attacks namely as *stealthy* attacks [33, 34]. Stealthy attack is a type of cybersecurity attack in which the malicious code is hidden inside the benign application for performing harmful purposes.

The main purpose of stealthy attacks is to remain undetected for a longer period of time in the computing system. The longer the threat remains undiscovered the more opportunity it has to compromise computers and/or steal information before suitable detection mechanism can be deployed to protect against it. Stolfo et al. discovered a new type of stealthy threat referred as *embedded malware* [33]. Under this threat, the attacker embeds the malicious code or file inside a benign file on the target host such that the benign and malicious applications are executed as a single thread on the target system. It has been shown that traditional signature-based antivirus applications are unable to detect embedded malware even when the exact signature of malware is available in the detector database. Embedded malware is potentially a serious security threat and accurate anomaly detection techniques must be developed to mitigate it.

The existing studies on hardware-based malware detection have primarily assumed that the malware is spawned as a separate thread while executing on the target host. However, in real-world scenarios malicious programs attempt to hide themselves within a benign application to bypass the detection mechanisms. Embedded malware is a category of stealthy security threats that allows malicious code to be hidden inside a benign application on the target host [35].

In response to the aforementioned challenges, in this research, we propose an effective time series machine learning-based approach, referred as *CHASE*, to accurately detect the embedded malicious patterns inside the benign programs using only one HPC feature (branch instruction). The main objective of this work is to accurately detect the malicious application embedded inside the benign program using least number of microarchitectural events (only one HPCs) in which the traditional ML-based solutions are unable to detect them with even 8/16 features. Using an effective feature reduction technique, we first identify the most prominent low-level feature for embedded malware detection. Next, we propose a lightweight scalable time series-based Fully Convolutional Neural Network (FCN) model that automatically identifies potentially contaminated samples in HPC-based time series to distinguish the stealthy malware at run-time using only branch instructions as the most significant HPC event.

1.4 Machine Learning-Based Approaches for Energy-Efficiency Prediction in Heterogeneous Architectures

In this section, we examine the suitability of applying effective machine learning techniques on captured run-time low-level information for addressing the performance vs. power consumption trade-offs and enhancing the energy-efficiency of heterogeneous multicore architectures. In particular, we show that hardware performance counter information can be also effectively used for energy-efficiency prediction and scheduling of multithreaded applications running on multicore heterogeneous architectures.

Heterogeneous Multicore Processors (HMPs) are primarily comprised of multiple core types (small vs. big core architectures) with various performance and power characteristics which offer the flexibility to assign each thread to a core that provides the maximum energy-efficiency. Although this architecture provides more flexibility for the running application to determine the optimal run-time settings that maximize energy-efficiency, due to the interdependence of various tuning parameters such as the type of core, run-time voltage and frequency, and the number of threads, the scheduling becomes more challenging. More importantly, the impact of Power Conversion Efficiency (PCE) of the On-Chip Voltage Regulators (OCVRs) is another important parameter that makes it more challenging to schedule multithreaded applications on HMPs [36–38].

In this research, first we investigate the scheduling challenges of multithreaded applications on dynamic heterogeneous architectures. To this aim, we describe a systematic machine learning-based approach to predict the right configurations for running multithreaded workloads on the composite cores architecture. It achieves this by developing a machine learning-based approach to predict core type, voltage and frequency to maximize the energy-efficiency. Our predictor learns offline from an extensive set of training multithreaded workloads. It is then applied to predict the optimal processor configuration at run-time by considering of the multithreaded applications characteristics and the optimization objective.

For this purpose, five well-known machine learning models are implemented for energy-efficiency optimization and precisely compared in terms of accuracy and hardware overhead to guide the scheduling decisions. The results show that while complex machine learning models such as MultiLayerPerceptron are achieving higher accuracy, after evaluating their implementation overheads, they perform worst in terms of power, accuracy/area and latency as compared to simpler but slightly less accurate regression-based and tree-based classifiers.

Secondly, the importance of concurrent optimization and fine-tuning of the circuit and architectural parameters for energy-efficient scheduling on HMPs is addressed to harness the power of heterogeneity. In addition, the scheduling challenges for multithreaded applications are investigated for HMP architectures that account for the impact of power conversion efficiency. To this aim, a highly accurate learning-based model is developed for energy-efficiency prediction to guide the scheduling decision. Using the predictive model, we further develop a PCE-aware scheduling scheme for effective mapping of multithreaded applications onto an HMP.

Chapter 2: Background and Related Work

In this section, we comprehensively explore the background and state-of-the-art works on hardware performance counter and machine learning-based solutions for enhancing the security and energy-efficiency of the modern computer computer systems.

2.1 Hardware Performance Counter Registers

Hardware Performance Counters (HPCs) are special purpose registers available in modern microprocessors which keep track of different microarchitectural events [1, 2, 9]. A variety of todays processor platforms such as Intel, ARM, and AMD include HPC built-in registers on their processors. The main purpose of HPCs is to analyze and tune architectural level performance of running applications [36, 37, 39–41]. HPC registers are easily programmable across all platforms capable enough to keep track various number of microarchitectural features such as cache memories access misses, TLB hits misses, branch mispredictions, and core stalls of the chip [10, 42, 43].

The performance counters are often programmed to indicate an interrupt when a counter overflows or even be set to start the counter from the desired value [44, 45]. The software handles these interrupts allowing programmers to analyze the hardware resource utilization by their applications at run-time. While hardware performance counters are finding their ways in various processor platforms from high-performance to embedded, they are limited in the number of microarchitectural events that can be captured simultaneously [1, 2, 13, 22, 29]. This is mainly due to limited number of physical registers on the processor chip which are expensive to implement.

Recently, application areas of HPC are grown from mere performance analysis to detecting firmware modification in Embedded Systems [23, 46], estimating computer system performance, power and energy-efficiency [36–39, 42, 45, 47–51], and even detection of malware [1, 2, 8, 10, 20, 27, 29, 31, 52]. The operating systems can program the HPCs using control registers, called Performance Monitoring Counters (PMCs) found in the Performance Monitoring Unit (PMU). These registers are known as Model Specific Registers (MSRs) on Intel processors. User-space applications can access the HPCs through software interfaces to PMUs and configure the HPCs using the PMCs.

In this research, in order to improve the security of processors, we use HPCs information to construct a vector of microarchitectural events by profiling malware and benign applications and feed the vector into various machine learning classifiers to detect the behavior of application. We utilize the HPC registers to collect execution traces for all available microarchitectural events by executing collected malware and benign applications in an isolated environment. Our goal is to learn malware behavior with collected HPC of various applications (including malware and normal) using supervised machine learning methods. Detailed performance counters collection procedure is discussed in the next sections.

2.2 Machine Learning Techniques used for Malware Detection

Methods of machine learning, which build predictive models that generalize training data, have proven to be useful for detecting malware. In this work, we propose effective machine learning-based solutions for improving the security of computer systems in the context of hardware-supported malware detection which rely on the run-time traces collected from HPCs. Demme et al. [20] showed the offline machine learning effectiveness in classifying malware by learning from hardware performance counter events. They indicated a high detection accuracy result for Android malware by applying complex ML algorithms like Artificial Neural Network (ANN). Although they discussed implementing classifiers on hardware, they did not present any hardware overhead results.

Supervised learning, in the context of machine learning, is a type of system in which both input and desired output data are provided. Input and output data are labeled for classification to provide a learning basis for inferring the data [13]. The majority of practical machine learning uses supervised learning. Supervised learning is basically applications in which the training data comprises examples of the input vectors along with their corresponding target vectors. It is where we have input variables and an output variable and an algorithm is used to learn the mapping function from the input to the output. The goal is to approximate the mapping function appropriately that when we have new input data, the output variables can be estimated promptly for that data.

In this work, we have deployed various supervised machine learning classifiers for malware detection. These machine learning classifiers consist of eight general classifiers including Bayesian Network (BayesNet), an artificial neural network model (MLP: MultiLayerPerceptron), two different rule-based algorithms (JRIP, OneR), a tree-based learning technique namely J48 and REPTree, a Support Vector Machine called SMO, and a Gradient Decent Optimization technique called SGD. We selected these eleven classifiers for two primary reasons. First, they are from different branches of machine learning methods; Bayesian network, neural network, decision tree, and rule-based covering a diverse range of learning algorithms which are inclusive to model both linear and non-linear problems. Second, the prediction model produced by these learning algorithms is a binary classification model which is compatible with our malware analysis and detection problem. We precisely compared and characterized these ML classifiers in terms of detection performance (accuracy and area under the curve), and hardware overhead. A brief description of each ML classifiers used in this research is presented below:

Bayesian Network (BN): A class of probabilistic and statistic graphical model that that aims to model conditional dependence and causation, by representing a set of variables and conditional dependencies by edges in a directed graph.

Artificial Neural Network (ANN): Consists of units (neurons), arranged in layers, which convert an input vector into some output. Each unit takes an input, applies a (often nonlinear) function to it and then passes the output on to the next layer. The original goal of the ANN approach was to solve problems in the same way that a human brain would [14, 48].

Decision Tree (DT): Sequential models, known as "*divide and conquer*" algorithms, which logically combine a sequence of simple tests where a numerical attribute is compared against a threshold value or against a set of possible values. It is essentially a flow chart like structure where each internal node denotes a test on an attribute with each branch representing an outcome of the test and each leaf holding a class label [14, 48].

Rule-Based Classification: Machine learning models that identify, learn, and evolve a set of relational rules that collectively represent the knowledge captured by the system.

Support Vector Machine (SVM): In machine learning domain, a Support Vector Machine (SVM) is a subset of supervised learning models which are discriminative classifiers formally defined by a separating hyperplane that examine the data used for classification and regression analysis. In other words, given labeled training data, the algorithm outputs an optimal hyperplane which categorizes new examples. In two dimentional space this hyperplane is a line dividing a plane in two parts where in each class lay in either side. SVMs can be efficiently deployed for both linear classification a non-linear classification. In addition, Stochastic Gradient Descent (SGD) is a simple yet very efficient approach to discriminative learning of linear classifiers under convex loss functions such as (linear) Support Vector Machines and Logistic Regression.

Recently, SGD has received a significant consideration by the researchers in the context of large-scale learning. SGD has been successfully applied to large-scale and sparse machine learning problems often encountered in text classification and natural language processing. The advantages of Stochastic Gradient Descent are efficiency and ease of implementation, while it has disadvantages including being sensitive to feature scaling and also requiring a number of hyperparameters such as the regularization parameter and the number of iterations.

2.2.1 Performance Evaluation Metrics

Evaluating the performance of machine learning classifiers is an important step in implementing effective ML-based optimization solutions for security and energy-efficiency analysis (e.g. malware detection techniques). In the field of machine learning and statistics, there exists variety of measures that can be deployed to evaluate the performance of a ML-based detection method in order to show its detection accuracy. The standard evaluation metrics used for performance analysis of malware detection and classification are summarized in Table 2.1. In this section, we briefly describe each evaluation metric.

In malware detection rate analysis, malicious applications samples are often considered as positive instances. As a result, the True Positive Rate (TPR) metric or the hit rate, represents sensitivity that stands for the proportion of correctly identified positives. It is basically the rate of malware samples (i.e., positive instances) correctly classified by the classification model. The True Negative Rate (TNR) also represents specificity that measures the proportion of correctly identified negatives. In addition, the False Positive Rate (FPR) is the rate of benign files (i.e., negative instances) wrongly classified (i.e., misclassified as malware samples) [1, 2].

The F measure (F score) in machine learning is interpreted as a weighted average of the precision (p) and recall (r). The precision is the proportion of the sum of true positives versus the sum of positive instances and the recall is the proportion of instances that are predicted positive of all the instances that are positive. F measure is a more comprehensive evaluation metric over accuracy (percentage of correctly classified samples) since it takes both the precision and the recall into consideration. More importantly, F measure is also resilient to class imbalance in the dataset which is the case in our experiments. The Detection Accuracy (ACC) measures the rate of the correctly classified positive and negative samples [1, 2, 10, 13].

Precision and recall are not adequate for showing the performance of detection even contradictory to each other, because they do not include all the results and samples in their formula. F-score (i.e., F-measure) is then calculated based on precision and recall in order to compensate this disadvantage. Receiver Operating Characteristic (ROC) is a statistical plot that depicts a binary detection performance while its discrimination threshold setting is changeable. The ROC space is evaluated by considering FPR and TPR as x and y axes, respectively. It primarily assists in determining trade-offs between TP and FP. Since TPR and FPR are equivalent to sensitivity and (1-specificity) respectively, each prediction result represents one point in the ROC space. The point in the upper left corner or coordinate (0, 1) of the ROC curve stands for the best detection result, representing 100% sensitivity and 100%

Table (2.1) Evaluation metrics for performance of malware detection techniques

Evaluation Metric	**Description**
True Positive	Correct positive prediction
False Positive	Incorrect positive prediction
True Negative	Correct negative prediction
False Negative	Incorrect negative prediction
Specificity: True Negative Rate (TNR)	$TNR = TN/(TN + FP)$
False Positive Rate (FPR)	$FPR = FP/(FP + TN)$
Precision	$P = TP/(FP + TN)$
Recall: True Positive Rate (TPR)	$TPR = TP/(TP + FN)$
F measure (F score)	$Fmeasure = 2 \times (Precision \times Recall)/(Precision + Recall)$
Detection Accuracy	$ACC = (TP + TN)/(TP + FP + TN + FN)$
Error Rate	$ERR = (FP + FN)/(P + N)$
Area Under the Curve (AUC)	$AUC = \int_0^1 TPR(x)dx = \int_0^1 P(A > \tau(x))dx$

specificity. An Area Under the Curve (AUC) is usually between 0.5-1.0, the bigger, the better the detection is [1, 13]. Due to the fact that it is difficult to concurrently meet with high precision and recall, we need to make a trade-off to balance both metrics. As a result, F-measure is often used to indicate detection performance [2].

Confusion Matrix for ML Classifiers: In statistical machine learning domain, a confusion matrix, is a specific table that represents the prediction performance of a machine learning classifier. It comprised of two dimensions namely as ”actual” and ”predicted”, and identical sets of ”classes” in both dimensions. Each row of the confusion matrix represents the instances in a predicted class while each column represents the instances in an actual class (or vice versa).

As shown in 2.1 a confusion matrix is formed from four outcomes produced as a result of binary classification. A binary classifier predicts all data instances of a test dataset as either positive or negative. This classification (or prediction) produces four outcomes listed as follows:

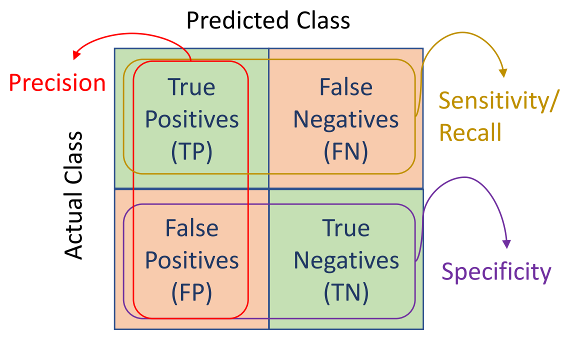


Figure (2.1) Confusion matrix layout for a machine learning classifier

- True Positive (TP): Correct positive prediction; e.g. malicious applications that are correctly classified as malware
- False Positive (FP): Incorrect positive prediction; e.g. benign applications that are incorrectly classified as malware
- True Negative (TN): Correct negative prediction; e.g. benign applications that are correctly classified as benign
- False Negative (FN): Incorrect negative prediction; e.g. malicious applications that are incorrectly classified as benign

2.3 Comprehensive Analysis of State-of-the-Art

2.3.1 Hardware-Based Malware Detection

In this section, we discuss the latest studies on malware detection techniques using low-level hardware features. The work in [20] was the first study that proposed to use HPC data for malware detection and demonstrated the effectiveness of using machine learning models. Although they have discussed implementing classifiers on hardware, they did not present any hardware overhead analysis results which are important as they decide which ML classifier responds in real-time and performs most cost-efficient. The hardware implementation overhead, particularly area and latency are important as they decide which ML classifier responds in real-time, and performs most cost-efficient. Also, the work has no discussion on main challenges to realize run-time HMD highlighted in this work including limited number of HPCs, and high variance in ML classifiers accuracy and performance across various classes of malware as well as embedded HMD.

The works in [53] and [54] discussed the feasibility of unsupervised learning method on low-level features to detect Return-Oriented Programming (ROP) and buffer overflow attacks by finding an anomaly in HPCs information. Although unsupervised algorithms can be more effective in detecting new malware and attacker evolution, they are complex in nature requiring more sophisticated analysis, resulting in complex hardware implementations. Also, their software implementation is not an effective solution to detect malware at run-time, due to large latency to compute the complex algorithms.

In [21, 55], the authors used sub-semantic features to detect malware. Moreover, they suggested changes in microprocessor pipeline to detect malware in truly real-time nature. They discussed estimated latency and area utilization of Logistic and

ANN algorithm implementation for their architecture. However, our work is different as it does not require any change in the processor pipeline. In [29], a single-stage ML-based hardware-assisted malware detection is proposed, but requires 8 or more HPC features to achieve higher accuracy and performance, which makes it less suitable for online malware detection. The work in [31] collected HPC information to construct support vector machine (SVM) detectors to identify malicious programs. However, they did not discuss the per-class analysis as well as hardware overhead analysis of deployed ML classifiers.

The works in [30] used logistic regression to classify malware into different types and trained a specialized classifier for detecting each type. They further used ensemble learning to improve the accuracy of logistic regression. In their ensemble learning implementation, they limited their experiments on just combining classifiers. In addition, they have ignored to account for the impact of reducing the number of HPCs on the performance of detectors and also have examined limited ML classifiers.

Collectively, prior works on hardware-assisted malware detection mostly focus on a particular machine learning classifier and ignoring per-class analysis of malware. Our work is different, given that for each class of malware, a unique ML classifier gives the best results (across detection rate, performance, delay, and area metrics), we thoroughly examined various general and ensemble learning-based malware detectors using varying number of microarchitectural features in terms of detection rate, robustness, performance, and hardware overheads for effective run-time harwdare-based mawlare detection [1].

2.3.2 Energy-Efficient Heterogeneous Architectures

Heterogeneous Architectures. Heterogeneous multicore architectures refers to computer systems that include more than one type of processors. Such systems gain performance or energy efficiency not just by adding the same type of processors, but by adding dissimilar coprocessors, usually incorporating specialized processing capabilities to handle particular tasks [56–58]. The static heterogeneous architecture in [59] enables efficient thread-to-core mapping and permits a change in the mapping across phases of execution through thread migration. Prior research has shown that the potential benefit of a static heterogeneous architecture is greater with fine-grained thread migration than with coarse-grain migration [60].

In research [61], an Intel Xeon is integrated with an Atom processor. Code instrumentation is used at the function or loop level to schedule different phases of the application on each processor. However, the separate core and memory subsystems in static heterogeneous architectures incur power and performance overheads for application migration, which makes dynamic mapping ineffective for fine-grained migration.

Unlike static heterogeneous architecture where the number and type of cores are fixed at run-time, dynamic heterogeneous architectures can be configured at run-time [62]. This provides more opportunity to map an application to a core which matches its resource needs more closely. Some of the first efforts to provide this kind of heterogeneity include Core Fusion [63] and TFlex [60]. Composite cores proposed a dynamic heterogeneous architecture where a big core can dynamically be decomposed into a smaller core [64].

The work in [62] and [65] extended the concept of composite cores into 3D stacking which enables fine-grain sharing of resources between cores on a stacked chip multiprocessor architecture. Their proposed architecture permits multiple smaller cores to be composed together making a larger core, given the performance and energy requirements of the running application. Previous work on dynamic heterogeneous architecture and specifically on composite core, has mainly studied mapping of single threaded applications. However, our work is different as it mainly focuses on multi-threaded applications and how they would benefit from such architecture to maximize the energy-efficiency.

Scheduling Challenges in Heterogeneous Architectures. As mentioned before, a main challenge for heterogeneous architectures is the mapping and scheduling decision, which finds the most efficient application-to-core match at run-time. The work in [59] and [66] address the problem of dynamic thread mapping in static heterogeneous many-core systems. Prior research aimed to maximize performance under power constraints [56, 59, 66, 67]. Our work is different as it first targets dynamic heterogeneous architectures where core size can be adapted at run-time, and second it aims to maximize the energy-efficiency by reducing the EDP. It is important to note that the power and performance of an application on different cores at various frequencies must be known for proper mapping. Traditional designs suggest selecting the best core based on a small sampling of applications on each core [68]. Other techniques [56,64,67,69], estimate core performance and adapt the resources without running applications on a particular core type using learning models.

The work in [70] and [64] provide a model for performance estimation on two core types (i.e., big and little cores). The complexity of application mapping on a heterogeneous architecture increases exponentially by increasing number of core types and applications. While previous studies have mainly examined the advantages of using single threaded applications, there is limited study on effectively mapping multithreaded applications onto heterogeneous composite cores architecture. There have been several studies on mapping multithreaded applications on homogeneous architectures.

The work in [71] suggested a framework called Thread Reinforcer to determine the appropriate number of threads for a multithreaded application on a homogeneous architecture. It examines the mapping between number of threads and number of cores to find the optimal or near optimal number of threads to minimize the execution time.

The research in [70] proposed a scheduling method to predict application to core mappings that enhances performance. Using profiling parameters, it estimates performance and examines whether the workload needs to run on different core type. The work in [59] proposed a mapping strategy for multithreaded applications on static heterogeneous multicore architecture by initializing a maximum throughput mapping and iteratively performing a thread swap on adjacent types of cores until the power constraint is met.

The work in [56] took a closer look at joint optimization of voltage and frequency as well as the microarchitecture. It proposed a platform, which is capable of scaling resources, i.e., bandwidth, capacity, voltage, and frequency, based on single-threaded application performance requirements at run-time while reducing EDP.

Chapter 3: Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection

Malware (Malicious Software) is a piece of code designed to perform various malicious activities, such as destroying the data, stealing information, running destructive or intrusive programs on devices to perform Denial-of-Service (DoS) attack, and gaining root access without the consent of user [1,2,8,17]. Malware detection can be simplified as a binary classification problem regardless of what detection method is being used [1, 13]. It is basically envisioned as distinguishing whether the running application has malicious intent or not.

Traditional malware detection approaches such as signature-based detection and semantics-based anomaly detection are considered as software-based solutions and incur significant computational overheads [1, 2, 13, 29]. In particular, these software-based malware detection methods such as AV software primarily pose several drawbacks. First, they rely on static signature-based detection in order to detect malicious pattern of infected application. Such detection mechanism search for suspicious byte patterns in the program, whereas an attacker can deceive AV software by programming and crafting malware in such a way that its signature appears as a benign software. Second, AV software are prone to exploits like any other software which can ultimately compromise protection if exploited. Third, AV software tools are slow and resource hungry. Conditions become even worse for metamorphic viruses, as defective detection of such attacks is an NP-complete problem [72].

Recent studies on malware detection approaches have demonstrated that malware behavior can be differentiated from benign applications by classifying anomalies in the low-level microarchitectural features spaces such as microarchitectural events collected by Hardware Performance Counter (HPC) registers[1, 2, 9, 13]. HPCs are CPU hardware registers that count hardware events such as instructions executed, cache-misses suffered, or branches mispredicted.

Performance counters data have been extensively used to predict the power, performance, and energy-efficiency of computing systems [10, 36–39], and recently drew attentions to be used for detecting the malicious pattern of running applications to improve the security of systems. Thus, malware detection using HPCs microarchitectural events has emerged as a promising alternative to traditional malware detection methods. As learning the underlying patterns of these microarchitectural events can aid in detecting malware, machine learning techniques are widely deployed for malware detection. The HPC microarchitectural features are used to train ML-based classifiers. In addition, such ML-based malware detection methods can be implemented in microprocessor hardware with significantly low overhead as compared to the software-based methods, as detection of anomaly inside the hardware is very fast (few clock cycles) [1, 13].

Recently, there has been a number of work on hardware-based malware detection using HPCs information [20–23, 30, 31, 46, 55]. However, these works performed a limited study on malware classification accounting for the availability of a large number (e.g. 16 or 32) and diverse type of HPCs. While, modern processors in the high-performance domain have a small number of HPCs (2 to 8), due to several reasons including the design complexity and cost of concurrent monitoring of microarchitectural events [1, 25, 26].

Due to deep pipelines, complex prefetchers, branch predictors, modern cache design etc., HPCs implementation becomes a great challenge in terms of counting multiple events and maintaining counter accuracy at the same time under speculative execution [1, 25, 26]. Better accuracy requires better and more complex hardware design hence increasing the number of counters with limited accuracy doesn't appear to be a good trade-off. Even modern Intel Xeon architectures houses only 4-6 performance counters, compare to 2 in Pentium 4 and server class Intel Atom processor, for the very same reason. For embedded mobile and IoT domains, the number of HPCs that can be accessed simultaneously is even smaller.

Therefore, collecting a variety of microarchitectural events, more than the number of available HPCs, to achieve high accuracy using the general ML models presented in prior work, requires running the application multiple times, since the hardware can only count a small subset of events concurrently. This approach is not practical for run-time detection of malware. In addition, previous studies, mostly focus on specific learning classifiers and limited types of malware. A quantitative comparison of these studies indicates that there is no unique classifier that delivers the best results across various metrics including performance (accuracy and robustness) and area overhead as well as detection delay and various classes of malware.

As the performance of malware detection depends on the type of ML classifier applied and the number and type of HPC events used, in this work, we first illustrate the impact of ML classifier type on malware detection accuracy and performance and the effect of number of HPC events for malware detection [1]. To achieve a high accuracy across all studied general ML classifiers, of more than 80%, at least 16 hardware performance counters are required, which as discussed is not available in modern processors, even in the high-performance domain, making run-time detection of malware impractical using these methods. Therefore, a key challenge in making the hardware-based malware detection a practical run-time solution is how to use a limited number of HPCs available in a microprocessor (for instance 2 or 4) and match the accuracy and performance of malware detection with the ones that can be achieved by a larger number of HPC events (for instance 16 or 32).

In this work, we address this challenge by proposing ensemble learning techniques to improve the accuracy and performance of the hardware-based malware detectors and break the trade-off between accuracy/performance with respect to the number of HPCs. We explore the effectiveness of ensemble learning models in 1) reducing the number of required performance counters for implementing effective ML classifiers for run-time malware detection and 2) improving the performance of weak but low-cost classifiers in malware detection with a small number of HPCs.

This work proposes a machine learning-based solution to break this trade-off to realize effective run-time detection of malware. We propose ensemble learning techniques to improve the performance of the hardware-based malware detectors despite using a very small number of microarchitectural events that are captured at run-time by existing HPCs, eliminating the need to run an application several times. For this purpose, eight robust machine learning models and two well-known ensemble learning classifiers applied on all studied ML models (sixteen in total) are implemented for malware detection and precisely compared and characterized in terms of detection accuracy, robustness, performance (accuracyrobustness), and hardware overheads. The experimental results show that the proposed ensemble learning-based malware detection with just 2 HPCs using ensemble technique outperforms standard classifiers with 8 HPCs by up to 17%. In addition, it can match the robustness and performance of standard ML-based detectors with 16 HPCs while using only 4 HPCs allowing effective run-time detection of malware.

3.1 Ensemble Learning

Ensemble learning is a branch of machine learning which is used to improve the accuracy and performance of general ML classifiers by generating a set of base learners and combining their outputs for final decision [1, 2, 13, 73]. It fully exploits complementary information of different classifiers to improve the decision accuracy and performance. The ensemble learning and joint decision procedure are widely used to devise learning methods to achieve more accurate predictions and stronger generalization performance. In this work, we deploy and analyze the effectiveness of two ensemble learning methods for efficient malware detection even with less number of HPCs. These ensemble methods are briefly described in below:

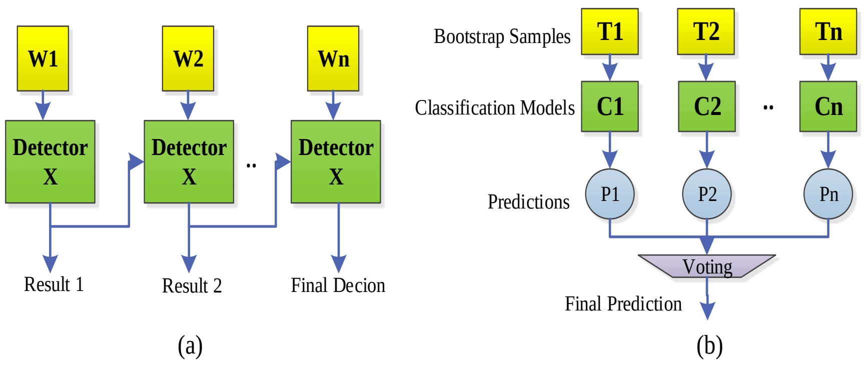


Figure (3.1) Ensemble learning block diagrams a) AdaBoost, b) Bagging

3.1.1 Boosting

Boosting is one of the most commonly used ensemble learning methods for enhancing the performance of ML algorithms. Adaptive Boosting, or in short AdaBoost [74], is the first proposed implementation of this type of ensemble learners. Figure 3.1-a illustrates the AdaBoost methodology. As shown, each base classifier is trained on a weighted form of the training dataset in which the weights depend on the performance of the previous base ML classifier.

Once all the base classifiers are trained, they will be combined to produce the final classifier. Each training instance in the dataset is weighted and the weights are updated based on the overall accuracy of the model and whether an instance was classified correctly or not. Subsequent models are trained and added until a minimum accuracy is achieved or no further improvement is possible. In this work, we applied AdaBoost as a boosting learning technique on all studied general ML classifiers to analyze its impact on the accuracy and performance improvement of hardware-based malware detection [1,13].

3.1.2 Bagging

Bagging, or Bootstrap Aggregation is an ensemble learning model that is used for classification and regression problems. It is a statistical prediction technique where a statistical value like a mean is estimated from multiple random samples of training data which are drawn with replacement and used to train different ML models. Each model is then exploited to make a prediction and the results are averaged to give a more robust and generalized prediction.

Figure 3.1-b illustrates the overview of bagging model. Bagging is a technique that is best used with models with low bias and high variance, in which the predictions of base learners are highly dependent on the data from which they were trained. Therefore, it is most suited for our purpose, given the wide variation in ML classifier performance as we will show later in this work. The most used algorithm for bagging that fits the requirement of high variance are decision trees [1,74].

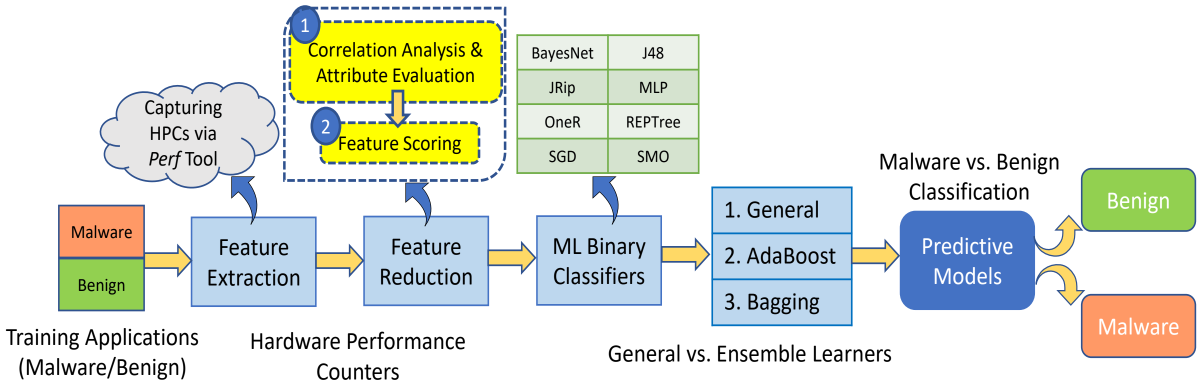
3.2 Proposed Run-Time Malware Detection

In this section, we present the details of our proposed run-time hardware-based malware detection approach.

3.2.1 Experimental Setup and Data Collection

This section provides the details of the experimental setup and data collection procedure. We execute all applications on an Intel Xeon X5550 machine running Ubuntu 14.04 with Linux 4.4 Kernel and collect various HPCs data. This processor is based on Intels Nehalem design, providing four performance counter registers. In order to extract the HPC information, we use *Perf* tool available under Linux. *Perf* provides rich generalized abstractions over hardware specific capabilities. It exploits *perf-event-open* function call in the background which can measure multiple events simultaneously. We have executed more than 100 benign and malware applications for HPC data collection. Benign applications include MiBench benchmark suite, Linux system programs, browsers, text editors, and word processor. For malware applications, Linux malware is collected from virustotal.com. Malware applications include Linux ELFs, python scripts, perl scripts, and bash scripts, which are created to perform malicious activities. After collecting microarchitectural events using *Perf*, we use WEKA tool [75] for evaluating the accuracy and performance of various machine learning classifiers.

Figure 3.2 depicts the overview of the proposed hardware-based malware detection approach and training the ML classifiers for predicting the malicious behavior of applications. It is primarily composed of various stages including feature extraction, feature reduction, and ML classifiers (general and ensemble) implementation for



Correlation Analysis & Attribute Evaluation
Feature Scoring
Capturing HPCs via *Perf* Tool
BayesNet
J48
JRip
MLP
OneR
REPTree
SGD
SMO
Malware
Benign
Feature Extraction
Feature Reduction
ML Binary Classifiers
1. General
2. AdaBoost
3. Bagging
Malware vs. Benign Classification
Predictive Models
Benign
Malware
Training Applications (Malware/Benign)
Hardware Performance Counters
General vs. Ensemble Learners

Figure (3.2) The overview of the proposed run-time hardware-based malware detection framework

malware detection which will be discussed in more details in next sections. HPC information is collected by executing all applications in Linux Containers (LXC) which is an isolated environment [76]. LXC is an operating system level virtualization method that shares the same kernel with the host operating system. In this work, LXC is chosen over other commonly available virtual platforms such as VMWare or VirtualBox since it provides access to actual performance counters data instead of emulating the HPC registers behavior.

We extracted 44 CPU events available under *Perf* tool. Since Intel Xeon has only 4 counter registers available [77], we can only measure 4 events at a time. As a result, multiple runs are required to fully capture all events. We divide 44 events into 11 batches of 4 events and run each application 11 times at sampling time of 10ms to gather all microarchitectural events. Running malware inside the container can contaminate the environment which may affect subsequent data collection. To ensure that there is no contamination in collected data due to the previous run, the container is destroyed after each run.

3.2.2 Feature Selection

Feature selection can be found in many areas of data mining such as classification, clustering, association rules mining, and regression. Feature selection is primarily considered as a process of selecting the most important subset of features from original features space [1,2,10,78,79]. As the dimensionality of a data expands, the number of features increases. Finding an optimal feature subset is typically intractable and many problems related to feature selection have been shown to be NP-hard. Redundancy of attributes is also an important point that needs to be accounted for. An attribute may be considered as a redundant feature if it can be derived from another attribute or set of attributes [39, 78].

As mentioned earlier, detecting malware using machine learning models requires representing programs at low microarchitectural level. This process produces very high dimensional dataset. Running ML algorithms with large HPCs would be complex and slow. Besides, incorporating irrelevant features would result in lower accuracy for the classifier [37,39,80]. Therefore, instead of accounting for all captured features, irrelevant data is identified and removed using a feature reduction algorithm and a subset of HPCs is selected that includes the most important features for classification. The features are supplied to each learning algorithm and the learning algorithm attempts to find a correlation between the feature values and the application behavior to predict the malware or benign type.

As discussed, the key aspect of building an accurate detector is finding the right features to characterize the input data. We started from 44 performance counters. As shown in Figure 3.2, after feature extraction, the feature reduction process reduces the number of low-level features. We first use Correlation Attribute Evaluation on our training set under WEKA to monitor the most vital microarchitectural parameters to capture application characteristics. Next, the features are scored based on their importance and relevance to the target variable through the feature scoring process.

By applying the feature reduction method, the sixteen most related hardware performance counters are determined and numbered in order of importance for malware detection. These HPCs and their descriptions are listed in Table 1. They are included in the general and ensemble learning prediction models as input parameters to classify the behavior of application as shown in Figure 3.3. The selected features include HPCs representing pipeline front-end, pipeline back-end, cache subsystem, and main memory behaviors influential in the performance of standard applications.

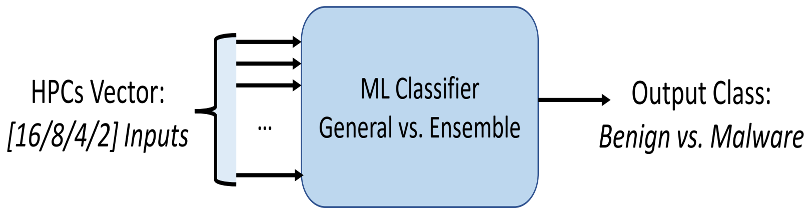


Figure (3.3) The functionality of ML-based malware detectors

3.2.3 Training and Testing the Malware Detectors

In this section, we describe the details of training and testing the ML classifiers for malware detection. Training involves profiling the incoming application with Perf tool under Linux and extracting low-level feature values for each training program, reducing the extracted features to the most vital performance counters, and developing a learning model from the training data. It is important to note that the input variables in our classifiers are the HPCs extracted every 10ms interval from the running applications, and the output variable is the class (malware vs. benign) of an application.

Table (3.1) Hardware performance counters in order of importance

Rank	HPC Feature	Description
1	branch instructions	Number of branch instructions that are retired
2	branch loads	Number of successful branches (Taken)
3	iTLB load misses	Number of misses occurred in instruction TLB during load operations
4	dTLB load misses	Number of misses occurred in data TLB during load operations
5	dTLB store misses	Number of misses in data TLB during store operations
6	L1 dcache stores	Number of stores instructions executed into L1 cache
7	cache misses	Number of misses occurred in Last Level Cache (LLC)
8	node loads	Number of successful load operations occurred into DRAM
9	dTLB stores	Number of store operations occurred in data TLB
10	iTLB loads	Number of load operations occurred in instruction TLB
11	L1 icache load misses	Number of instruction misses occurred in L1 instructions cache
12	branch load misses	Number of load branches that are mispredicted
13	branch misses	Number of branches that are mispredicted
14	LLC store misses	Number of misses occurred in L3 cache during store operation
15	node stores	Number of successful store operations to DRAM
16	L1 dcache load-misses	Number of misses occurred in L1 data cache during load operation

For each ML classifier, we construct the general and ensemble models (AdaBoost and Bagging) to detect the malware. In order to validate each of the utilized ML classifiers, a standard 70%-30% dataset split for training and testing is followed. To ensure a non-biased splitting, 70% benign- 70% malware application for training (known applications) and 30% benign-30% malware applications for testing (unknown applications) are used.

3.3 Experimental Results and Evaluation

In this section, we present the evaluation results for different machine learning classifiers. We thoroughly compare these learning techniques in terms of the prediction accuracy, robustness, performance, and the hardware implementation costs.

Table (3.2) AUC values for all general and ensemble detectors with varying number of HPCs

Classifier	16HPC	8HPC	4HPC	4HPC-Boosted	4HPC-Bagging	2HPC	2HPC-Boosted	2HPC-Bagging
BayesNet	0.922	0.922	0.926	0.916	0.937	0.916	0.875	0.927
J48	0.876	0.876	0.81	0.936	0.852	0.81	0.925	0.817
JRip	0.857	0.857	0.81	0.88	0.932	0.81	0.932	0.883
MultiLperc.	0.902	0.902	0.888	0.923	0.865	0.901	0.931	0.872
OneR	0.81	0.81	0.81	0.897	0.869	0.81	0.898	0.869
REPTree	0.85	0.85	0.81	0.85	0.885	0.81	0.924	0.91
SGD	0.741	0.741	0.714	0.889	0.736	0.714	0.714	0.714
SMO	0.651	0.651	0.67	0.883	0.851	0.68	0.886	0.827

3.3.1 Detection Accuracy

To evaluate the detection accuracy of our malware classifiers, we calculate the percentage value of samples that are correctly classified. Figure 3.4 shows a comprehensive accuracy comparison of various ML classifiers (general and ensemble) used for malware detection. We have implemented 8 general ML classifiers and two ensemble learning techniques and calculated their accuracy in classifying malware and benign applications. The accuracy of malware detection with different number of hardware performance counters (16, 8, 4 and 2) are reported. Before feature reduction (16 HPCs), most ML classifiers perform well, mostly providing above 80% accuracy. Feature reduction has noticeable impact on the accuracy of several classifiers. However, OneR classifiers perform well even after feature reduction. The reason that OneR classifier is not affected by feature reduction and shows almost constant accuracy results is that it only selects one performance counter (branch-instructions) to predict the malware behavior.

As can be seen in Figure 3.4, in some classifiers like BayesNet, JRip, OneR, REPTree, and SMO by reducing the number of hardware performance counters to 2 or 4 and applying ensemble learning techniques, a higher or similar accuracy level to 8/16 HPC models is achieved. This interesting observation confirms the effectiveness of using ensemble learning to boost the accuracy of classifiers. For instance, as

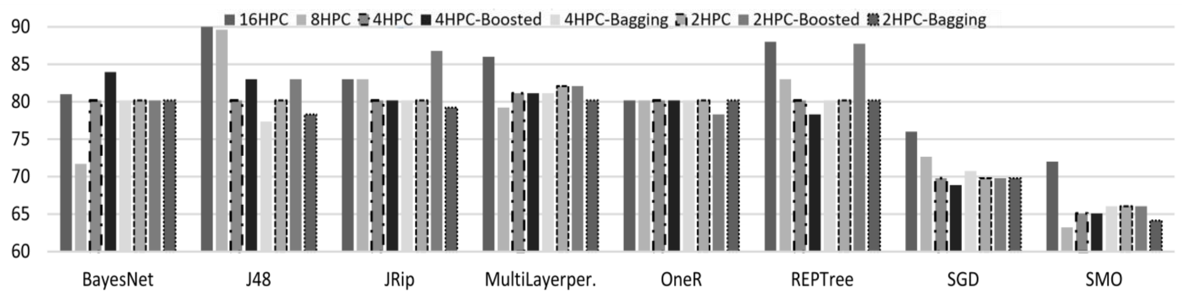


Figure (3.4) Detection accuracy results (ACC) for various ML classifiers with varying number of HPCs

shown, REPTree achieves close to 88% accuracy with 16 HPCs. However, we observe that reducing the number of vital performance counters to 2 and applying AdaBoost ensemble technique result in achieving almost the same 88% accuracy, as with 16 HPCs.

3.3.2 Classification Robustness

To evaluate the accuracy and robustness of ML classifiers in detecting malware, Receiver Operating Characteristics (ROC) graphs are used. As described earlier, the ROC curve is produced by plotting the fraction of true positives versus the fraction of false positives for a binary classifier as the threshold changes. The best possible classifier would thus yield a point in the upper left corner or coordinate (0,1) of the ROC space, representing 0% false positives and 100% true positives.

We use the Area Under the Curve (AUC) measure for ROC curve in the evaluation process to examine the robustness of each ML classifier. The area under the ROC curve corresponds to the probability of correctly identifying which application is malware and which is benign. In other words, the AUC measure is more related

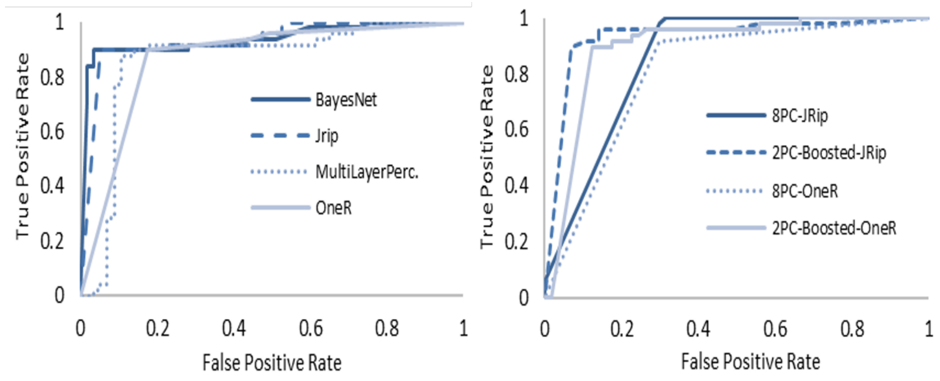


Figure (3.5) The ROC curves for ML-based malware detectors

to the robustness of the classifier. In this work, robustness term is referred to how well the classifier distinguishes between binary malware and benign classes, for all possible threshold values. The AUC value of the best possible classifier is equal to 1, which means that we can find a discrimination threshold under which the classifier obtains 0% false positives and 100% true positives.

Table 3.2 presents the list of the area under the ROC graphs values for each ML general and ensemble classifier with varying number of HPCs. It primarily presents the values for the ROC curves resulted from all comparisons between the general and ensemble-based detectors. A higher AUC value means that the ROC graph is closer to the optimal threshold and the classifier is performing better in terms of classification of malware and benign applications. Area under the curve analysis provides valuable insights to select possibly optimal ML classifiers suitable for malware detection and to discard the suboptimal detectors.

Figure 3.5 depicts the ROC curves for two different ensemble learning models and different number of performance counters. Due to space limitation, here we present the ROC graphs for selected ML classifiers and show the impact of ensemble learning techniques on AUC robustness. In Figure 3.5-a, the ROC graphs for 4 ML classifiers improved by Bagging ensemble learner are shown which were developed with 4 performance counters. As can be seen in this figure as well as Table 3.2, the BayesNet and JRip classifiers have maximum AUC of 0.937 and 0.932, respectively, delivering best robustness with only 4 performance counters. Figure 3.5-b represents the AdaBoost technique effectiveness on two different detectors when reducing the number of HPCs from 8 to 2. As shown, for each classifier boosting model significantly improve the AUC of ROC curve making the ML classifier more effective in terms of classification robustness.

3.3.3 Performance of Malware Detection

In order to evaluate and compare the performance of malware detectors, we consider the product of accuracy and area under the ROC graph (ACC*AUC) as a performance metric. This metric combines the impact of accuracy and robustness in malware classification and concurrently accounts for both measures. We accounted for performance as a final comparison metric across various ML classifiers since it is a more comprehensive measure by considering both impacts of the detection accuracy and AUC values. Figure 3.6 illustrates the ACC*AUC results of various ML classifiers under a varying number of hardware performance counters.

As can be seen in the results, most of the classifiers such as JRip, J48, Multi-Layer Perceptron (MLP), and SMO deliver higher performance when they are supplied with 16 and 8 performance counters and by decreasing the number of performance counters, the performance of general ML classifiers decreases showing the potential for applying ensemble learning techniques to boost the accuracy and performance with fewer performance counters.

For instance, in SMO classifier by reducing the number of performance counters to 4 and 2 and applying AdaBoost ensemble technique, we achieve 16% and 17%

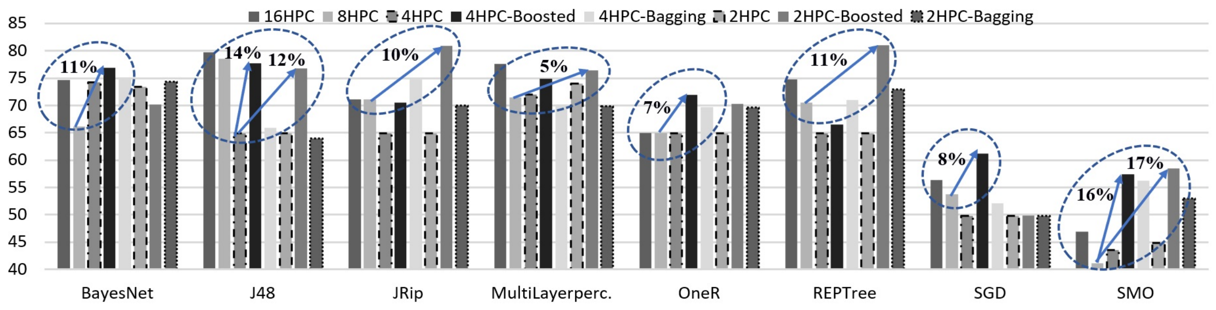


Figure (3.6) Performance results (ACC*AUC) for various ML classifiers with varying number of HPCs

performance improvement, respectively. In REPtree classifier, 2HPC-Boosted detector is achieving 11% improvement in ACC*AUC measure as compared to the general ML classifier with 8 performance counters. JRip classifier achieves 10% performance improvement by applying boosting method and 7% improvement with Bagging technique with the use of only 4 performance counters compared to using 8 HPCs.

The results clearly confirm the effectiveness of using ensemble techniques for performance improvement of ML classifiers with a lower number of HPCs for malware detection. The key point here is that rather than extracting 16 or 8 hardware performance counter which definitely impose significant implementation cost overhead to the systems in terms of resource utilization and power consumption, it is more effective to alternatively collect lower number of HPCs (four or two), depending on the classifier type, and boost the performance of the ML classifier with one of the ensemble learning approaches to improve the accuracy as well as the robustness of malware detectors.

3.3.4 Hardware Implementation Results

The software implementation of ML classifiers for malware detection is slow in the range of tens of milliseconds which is an order of magnitude higher than the latency needed to capture the malware at run-time [29]. Therefore, in this work, we develop a hardware implementation of the general and ensemble learning detectors. We use Vivado HLS compiler to develop the HDL implementation of the classifiers and deploy on Xilinx Virtex 7 FPGA. FPGA is a target in our study, as few modern microprocessors have on-chip FPGAs available for programmable logic implementation. Such arrangement makes it feasible to implement reprogrammable low-level malware detection logic (ML model) which can detect malware by reading the CPU hardware performance counters through shared memory bus [81].

When it comes to choosing the ML classifiers for hardware implementation, the accuracy of an algorithm is not the only parameter in decision-making. Design area and response time (latency) overhead of ML classifiers also plays a key role in selecting the cost-efficient hardware solution [82–86]. While complex algorithms such as Neural Networks can deliver high accuracy, they will also add significant overhead in terms of hardware implementation cost. Also given their complexity, they can be slow in detecting malware.

In order to compare hardware implementation costs, in Table 4.4, we report the results for general classifiers using 8 HPCs and boosting ensemble method (AdaBoost) applied on each classifier using 4 and 2 most important HPCs. Latency unit is represented in terms of the number of clock cycles (cycles @10 ns) required to classify input vector. In order to compare the area overhead of the implemented hardware-based ML classifiers, we consider the OpenSPARC (FPGA) implementation as reference and calculate the area overhead relative the core size. The area is the total number of utilized LUTs, FFs, and DSP units inside Virtex 7 FPGA. As can be seen from

Table 4.4, the Multi-Layer Perceptron algorithm, as expected, results in a significant area and latency overhead, as compared to other learning methods.

The ensemble learning introduces area overhead for some classifiers. However, the introduced overhead is less than 3% compared to the general ML classifiers using 8 HPCs for malware detection. In addition, in some other classifiers we observe that by using ensemble learning with a lower number of performance counters, the area overhead is significantly reduced, compared to the general classifiers using 8 HPCs. For instance, as reported in the previous section, the Boosted-MLP with 2 HPCs gains 5% performance improvement, while as shown in Table 4.4, it interestingly shows close to 19% area reduction in 2 HPC case and only 0.6% increase in 4 HPCs which is negligible, as compared to the general detectors with 8 HPCs. Ensemble learning algorithms generate models according to the data sets given and configuration of the algorithm. We observe that some algorithms do not see reduction in area from 4 HPCs to 2 HPCs. This is because such algorithms generate same number and equally complex models due to their nature. For instance, JRip-Boosted generates 10 models with 4 HPCs and 10 models with 2 HPCs, hence it is not guaranteed that the area of the 2 HPCs will be less than 4 HPCs. Because JRIP is a rule-based learning algorithm and the area highly depends on how many rules are generated for each model and the 2 HPCs case may have more rules per model.

To the best of our knowledge, there has been no prior work available that discusses the area costs for implementing ML classifiers as a function of HPCs. It can be argued that the number of HPCs can be increased during design time. However, there are several studies available such as [25,26,87] that discuss and justify the limited number of HPCs due to complex microarchitecture of modern microprocessors. Because of deeper pipelines, modern complex cache design and etc., implementing the hardware performance counter registers becomes a challenge issue in terms of counting multiple

Table (3.3) Hardware implementation results of various ML-based malware detectors

Model Used	**8HPC**		**4HPC-Boosted**		**2HPC-Boosted**	
Classifier	Latency @10ns	Area (%)	Latency @10ns	Area (%)	Latency @10ns	Area (%)
BayesNet	14	11.5	56	13.6	32	10.9
J48	9	3	67	4.3	35	4.1
SGD	34	4.3	87	6.3	51	5.1
JRip	4	2.5	56	5.3	37	8.2
MLP	302	61.1	591	61.7	201	42.2
OneR	1	2.	70	5.1	38	5
REPTree	39	2.9	60	3.9	30	3.7
SMO	34	4.3	87	6.3	51	5.1

microarchitectural low-level features and at the same time maintaining the accuracy, while achieving higher accuracy requires better and more complex hardware design. As a result, increasing the number of HPCs with limited accuracy doesn't appear to be a good trade-off. Compare to that, ensemble learning algorithm such as AdaBoost can be easily implemented on the programmable logic present in modern heterogeneous microprocessors. Clearly, the results show some trade-offs between the accuracy, latency, and area overhead. Therefore, it is important to compare classifiers by taking all of these parameters into consideration.

Concluding Remarks. Hardware-based detectors rely on machine learning classifiers and use HPCs information at run-time. A comparison of recent works on ML-based malware detectors shows that there is no unique general classifier that delivers the best results in terms of performance (accuracy and robustness), area overhead as well as detection delay across various types of malwares. In addition, prior studies mostly relied on a large number of HPCs to gain high accuracy making them less practical for run-time detection using very limited number of HPCs available in modern processors. In this paper, we showed a clear trade-off between the type and count of HPCs and malware classifier performance. To achieve a high accuracy and performance of more than 80% across all studied general ML classifiers, at least 16 HPCs are required, far beyond 2-8 HPCs available in modern architectures. In response to this challenge, this paper proposed using ensemble learning classifiers to boost the performance of general ML classifiers such that by only using 2-4 HPCs they can match the performance of 8-16 HPCs. The proposed ensemble classifiers are applied on 8 general ML classifiers and the results are comprehensively evaluated in terms of accuracy, robustness, performance, and hardware design overhead. The experimental results show that in all studied cases, boosting techniques improves the performance of malware detection classification by up to 17% while using a significantly lower number of performance counters. Given the implementation cost of on-chip HPCs and their limited availability and accuracy, the results of this research will help in making an architectural decision on the number and types of HPCs needed to implement in future architectures, to most effectively improve the performance of ML classifiers for detecting the malicious software.

Chapter 4: 2SMaRT Malware Detection Approach

The ever-increasing complexity of modern computing systems results in the growth of security vulnerabilities, making such systems appealing targets for increasingly sophisticated attacks [2, 12, 86]. As discussed, one of the most common cyber attacks are malicious software (malware) programs that are typically designed by cyber attackers to compromise the security of the computing systems by infecting the systems without the user authorization for serving harmful tasks such as stealing confidential information, unauthorized data access, and disrupting essential services to carry out financial fraud. According to a 2018 McAfee threats report [28], nearly 63 million new malware samples have been recorded in the third quarter of 2018, an all-time highest number with an increase of more than 53% from the second quarter of 2018. The current evolving proliferation of computing sytsems in mobile and Internet-of-Things (IoT) domains further exacerbates the security threats of malware attacks implying the necessity of protecting legitimate users from these attacks and calling for effective and low-cost malware detection approaches.

Existing traditional malware detection methods such as signature-based detection [88–92] and semantics-based anomaly detection techniques [93–96] are considered as software-based solutions that often incur significant computational overhead to the computer system [18, 29], making them unfit for devices with limited available computing and memory resources. Furthermore, the emergence of new malware threats requires patching or updating the software-based malware detection solutions (such as off-the-shelf anti-virus) that needs a vast amount of memory footprint and hardware resources which is not feasible for emerging computing systems specially in embedded mobile and Internet of Things (IoT) devices [3, 8, 10, 97, 98]. The emerging embedded systems and IoT devices used in various computing domains ranging from small mobile computing platforms to large scale IoT networks, which account for a wide range of applications are often highly resource-constrained challenging the conventional software-based methods traditionally deployed for detecting and containing malware in general purpose computing systems. In addition to the complexity and cost (computing and storage), the software-based malware detection methods mostly rely on the static signature analysis of the running programs, requiring continuous software update in the field to remain accurate in capturing emerging malware, which is not affordable for embedded systems with limited computing and communication bandwidth [1, 2, 10, 22]. Moreover, most of these advanced analysis techniques are architecture dependent i.e., dependent on the underlying hardware. Hence, this makes the existing traditional software-based malware detection techniques hard to import onto emerging embedded computing devices.

In response, Hardware-assisted Malware Detection (HMD) techniques by employing the underlying hardware-related information, have shown promising results reducing the latency of malware detection process by order of magnitude with small hardware cost [20, 31, 99]. Recent studies have shown that malware can be differentiated from normal programs by classifying anomalies using Machine Learning (ML) techniques in low-level microarchitectural feature spaces captured by Hardware Performance Counters (HPCs) [20, 29, 46, 53, 54, 99] to appropriately represent the application behavior. Machine learning-based malware detectors can be implemented in microprocessor hardware with significantly low overhead as compared to the software-based methods, as detection inside the hardware is very fast (few clock cycles) [29]. The HPCs are a set of special-purpose registers built into modern microprocessors to capture the trace of hardwareevents such as executed instructions, suffered cache-misses, or mispredicted branches for a running program [24,32,46]. While HPCs have been typically used for performance tuning of applications, in this work we leverage HPCs information for security enhancement of the computer systems.

While previous studies on hardware-assisted malware detection focus on one or few general ML classifiers and limited classes of malware [20–23, 29–32], it is not clear which of the ML models deliver the best results across various metrics including the detection rate, performance, hardware design overheads as well as detection delay across various classes of malware. In addition to the drawbacks of existing approaches and non-portability concerns, in this paper, we identified and addressed four major challenges to realize an effective run-time hardware-assisted malware detection which are described as follows:

Challenge 1 (C1): Determining Key Microarchitectural Features. Determining the most significant low-level microarchitectural features is an important step for hardware-based malware detection due to three main reasons: a) As there exists numerous microarchitectural events (for instance more than 100 in Intel Xeon) with each of them representing a different functionality, collecting all of them leads to data with high dimensionality [80,100]. b) Processing raw dataset involves computational complexity and induces delay [101]. In addition, as different microarchitectural events are employed for various purposes, it is important to determine the most suitable events for malware detection. c) Due to different functionality of malware classes which results in different impact on the underlying hardware events, it is crucial to know which hardware events are more relevant to a given class of malware. The challenge in specifying the non-trivial microarchitectural events is that these events differ with respect to the class of application.

Challenge 2 (C2): The Right Machine Learning Model. We studied several general ML classifiers to detect malware representing a diverse range of machine learning algorithms such as neural networks, bayesian network, rule-based, tree-based, and regression techniques. As observed in our experiments (see Table 4.1), no single ML classifier model is the winner; i.e. achieves higher malware detection rate across all studied malware classes. As we show in this work, this disparity of best ML solutions is also observed across other evaluation metrics (robustness, detection performance, and hardware implementation cost) making the detection and classification of malware classes highly dependent on the type of the ML classifier deployed. In addition, not knowing the malware type ahead of time, makes it a challenge to deploy the right ML model for effective detection of malware at run-time.

Challenge 3 (C3): HMD with Limited Available HPCs. Prior works on HMD have limited their studies on malware detection while accounting for the availability of a large number (e.g. 16 or 32) and diverse type of events accessed at a time [20–22, 29,30]. However, modern microprocessors even in the high-performance domain have limited number of HPC registers (2 to 8), due to the design complexity and cost of concurrent monitoring of microarchitectural events [25,26,87]. Due to deep pipelines, complex prefetchers, branch predictors, modern cache design etc., adding HPCs is a challenge in terms of counting multiple events and maintaining counter accuracy simultaneously under speculative execution [87]. Better detection rate requires better and more complex hardware design, hence increasing the number of counters with limited detection rate does not appear to be an efficient trade-off. For embedded mobile and IoT domains, the number of HPCs that can be accessed concurrently is even smaller. Therefore, utilizing a variety of microarchitectural events, more than the number of available HPCs, to achieve high detection rate using the general ML models presented in prior works, requires executing the application multiple times, since the hardware can only count a small subset of events at once. This approach is not practical for run-time detection of malware. Our experimental results show that the malware detection rate is highly dependent on the number of microarchitectural events deployed (see Figure 4.2) which provides unique opportunity to propose ensemble learning-based HMD to enhance the performance of HMD and break the trade-off between detection performance and the number of microarchitectural events.

The objective of this work is to improve the detection rate and performance of malware detection for different malware classes using a limited number of microarchitectural events equal to the available number of HPCs that can be captured at run-time. We propose a two-stage machine learning-based hardware-assisted malware detection approach, referred as *2SMaRT*, to not only effectively distinguish the malware from benign applications, but also to identify the class of malware at run-time using proper low-level features and determine right machine learning model for capturing the malware behavior. To the best of our knowledge, this is the first work that considers major challenges involved in run-time hardware-assisted malware detection and proposes a unified solution to address all of them. The key contributions of this work are summarized as follows:

- To facilitate an efficient run-time malware detection by choosing an optimal set of events captured from available HPCs, an effective feature reduction method is introduced by exploiting the correlation between HPC features to determine the most prominent microarchitectural events across various classes of malware that can be employed for effective detection of each malware class.
- Based on the comprehensive analysis of different classes of malware, we propose 2SMaRT, a two-level specialized run-time malware detection framework which comprised of a multiclass classification method to predict the malware classes or benign applications in the first level, followed by effective specialized ML

classifiers for an efficient and highly accurate per-class malware detection in the second level. The specialized malware detector is selected during run-time based on the detection performance of HMD with respect to the malware class and ML classifier.

- We further propose using an ensemble learning technique for HMD in the second level of 2SMaRT to improve the detection rate and performance of malware detection when using a limited number of features captured at run-time using limited existing HPCs, eliminating the need to run an application multiple times and matching to the detection rate that can be achieved offline.
- To quantify the effectiveness of each specialized malware detector, we precisely compare malware detectors in terms of detection accuracy, robustness, performance (accuracy$\times$ robustness), and hardware implementation overhead to determine the most suited ML classifier per malware class for effective run-time malware detection using 2SMaRT.

4.1 Motivation

In this section, we discuss the motivations for proposing a run-time malware detection using microarchitectural features.

4.1.1 Malware Detection using HPCs Data

In this work, we deploy HPCs information to construct a vector of microarchitectural features by profiling malware and benign applications. We take advantage of the HPC registers to collect execution traces for various microarchitectural events by executing malware and benign applications in an isolated environment (details will be discussed

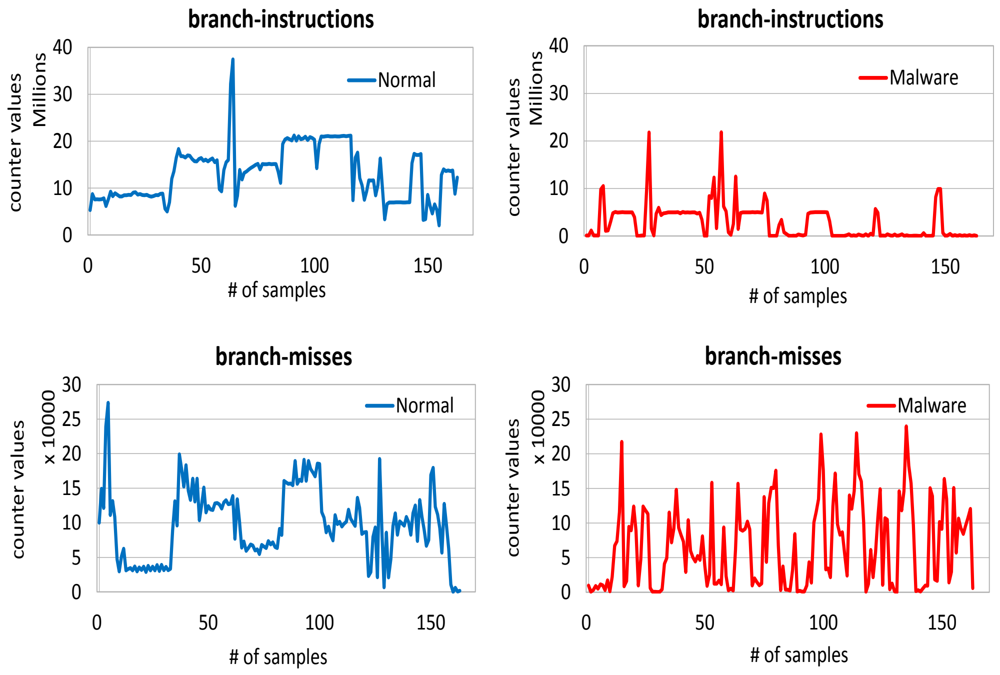


Figure (4.1) HPC traces of two different features (branch-instructions and branch misses) for sample malware and normal programs

in Section 4.2). The profiling process shows that if two different programs are executed on a processor, they generate different performance counter traces providing unique opportunity to detect the behavior of the running application. As an example, Figure 5.2 illustrates the trace of *branch instructions* and *branch misses* features for normal and malware applications. As seen, the malware traces are significantly different from benign applications for both features. Using this observation, malware can be distinguished from normal applications by its different HPC values. Our goal in this work is to learn malware behavior with the aid of supervised machine learning methods based on microarchitectural features captured by limited number of available HPCs from various applications. Unlike prior studies, the analysis given in this work is focused on run-time HMD, as such we limit the number of microarchitectural events to 4, which is equal to the maximum number of HPCs that can be simultaneously read at run-time.

4.1.2 The Need for Specialized Malware Detectors and Per-class Analysis

Specialized malware detectors are trained using the characteristics of a particular malware class, whereas generalized detectors are trained with generic characteristics obtained from malware applications belonging to different classes. As a case study to highlight the importance of per-class analysis of malware and specialized malware detection, Table 4.1 presents the ML classifiers that achieve the highest malware detection rate in our experiments across different malware classes using various number of HPCs (will be discussed in details in Section 4). Given the results, the following observations can be made: a) depending on the class of malware (Trojan, Virus, Rootkit, Backdoor), the type of ML classifier that performs best varies and there exists no specific ML classifier with highest rate for all malware classes; and b) not only does the ML classifier that performs best varies with the malware class, but also varies with the number of HPCs used. For instance, with 16 HPCs, BayesNet classifier achieves highest detection rate for Backdoor detection, but by reducing the number of HPCs to 4, OneR outperforms BayesNet. The disparity of optimal ML solutions across various classes of malware and varying number of HPC features implies the necessity of per-class malware analysis and developing effective specialized HMD for different classes of malware. Different behavior of each malware class allows a specialized detector to more effectively perform the classification. In this work, we implemented specialized detectors for four classes of malware including Backdoor, Virus, Rootkit, and Trojan.

Table (4.1) ML solutions with highest detection rate

Malware Class	16HPCs	8HPCs	4HPCs
Trojan	JRip	JRip	MLP
Virus	OneR	J48	MLP
Rootkit	J48	J48	MLP
Backdoor	BayesNet	OneR	OneR

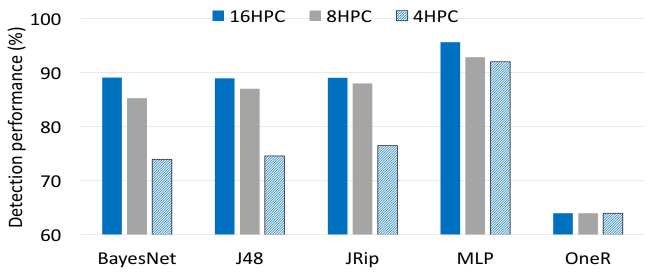


Figure (4.2) Case study for impact of #HPCs on HMD

4.1.3 Impact of Number of HPCs on Malware Detection

Figure 4.2 depicts a case study to verify the impact of number of HPCs on malware detection performance. As shown, a wide range of ML classifiers are employed and performance of malware detectors are evaluated using various number of HPC features. As the results show, with the increase in number of HPCs, the performance of malware detectors improves. As mentioned earlier, as the number of HPCs are limited in modern microprocessors, to perform an effective run-time malware detection, the number of microarchitectural events to capture should be equal to or less than the number of HPCs that are available (can be read simultaneously). As such, it is important to utilize fewer HPC events specially in IoT and embedded devices in order to realize a run-time malware detection along with ensuring a high malware detection performance. This eliminates the need to run the application multiple times to capture the required HPC features to detect the malicious patterns.

4.2 Proposed Malware Detection Framework

This section presents the details of the proposed run-time hardware-assisted malware detection approach.

4.2.1 Experimental Setup and Data Collection

The applications (both malware and benign) are executed on an Intel Xeon X5550 machine running Ubuntu 14.04 with Linux 4.4 Kernel. In order to extract the HPC information, we deployed *Perf* tool available under Linux. *Perf* provides rich generalized abstractions over hardware specific capabilitiesexploiting *perf-event-open* function call in the background which can measure multiple events simultaneously. We executed more than 3000 benign and malware applications for HPC data collection. Benign applications include MiBench [99] and SPEC2006 [102], Linux system programs, browsers, and text editors. For malware applications, Linux malware is collected from virustotal.com [103] and virusshare.com [104]. Malware applications include Linux ELFs, python scripts, perl scripts, and bash scripts which are created to perform malicious activities consisting of four classes of malware including 452 Backdoors, 350 Rootkits, 650 Viruses, and 1169 Trojans. The functionality of the selected malware applications are as follows: Backdoor applications attempt to gain remote access for the attacker and facilitates information leakage; Rootkits provide the attackers with privilege even to modify the registers and authorized programs; Trojans perform confidential information and passwords phishing; and Viruses that are able to duplicate themselves to launch DoS attacks.

Figure 4.3 depicts the overview of the data collection process and proposed run-time HMD framework. It is primarily composed of various stages including feature

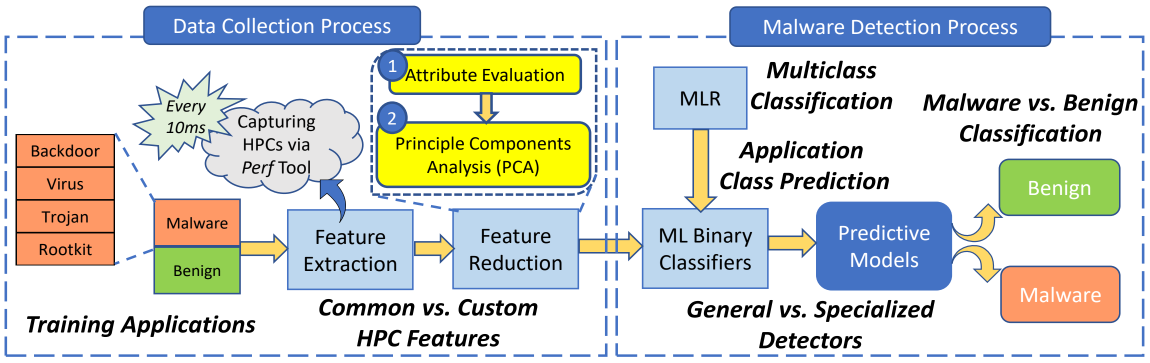

Data Collection Process
Malware Detection Process
Every 10ms
Capturing HPCs via Perf Tool
1 Attribute Evaluation
2 Principle Components Analysis (PCA)
Backdoor
Virus
Trojan
Rootkit
Malware
Benign
Feature Extraction
Feature Reduction
Training Applications
Common vs. Custom HPC Features
MLR
Multiclass Classification
Application Class Prediction
Malware vs. Benign Classification
ML Binary Classifiers
Predictive Models
Benign
Malware
General vs. Specialized Detectors

Figure (4.3) The overview of the proposed hardware-assisted malware detection framework

extraction, feature reduction, and ML classifiers (general and ensemble) implementation which will be discussed in more details in this section. HPC information is captured by executing all applications in Linux Containers (LXC) which is an isolated environment [76]. LXC is chosen over other commonly available virtual platforms such as VMWare or VirtualBox since it provides access to actual performance counters data instead of emulating HPCs. We extracted 44 CPU events available under *Perf* tool. Since Intel Xeon has only 4 counter registers available [77], we can only measure 4 events at a time. As a result, multiple runs are required to fully capture all events. We divide 44 events into 11 batches of 4 events and run each application 11 times at sampling time of 10ms to gather all microarchitectural events. Running malware inside the container can contaminate the environment which may negatively impact subsequent data collection. As result, to ensure that there is no contamination in the collected data due to previous run, the container is destroyed after each run. After collecting microarchitectural events using *Perf*, we deploy WEKA tool [75] for evaluating the detection rate and performance of various ML classifiers. In order to validate each of the the utilized ML classifiers, a standard 60%-40% dataset split for training and testing is followed.

Solution for C1: Key Microarchitectural Features

As discussed, detecting malware using ML models require representing programs as low-level microarchitectural level which leads to a high-dimensional data processing and involves high computational overheads and complexity [80, 105]. Furthermore, incorporating irrelevant features would result in lower accuracy for the ML classifiers[100, 101]. On the other hand, as we show in this work, microarchitectural events representing behavior of malware (and can be used to distinguish them from benign applications) are varied across different classes of malware. This poses two research questions. First, which low-level features are relevant to be employed to detect and classify a class of malware? Second, how to perform feature reduction of collected data to alleviate unnecessary computational overheads? As shown in Figure 4.2, by reducing the number of HPCs used for training the ML classifiers, the performance of malware detection greatly varies. In addition, as the number of HPCs that can be concurrently captured at run-time are limited, an optimal set of features among numerous possible HPC events needs to be determined and supplied to ML classifiers. The ML-based detector attempts to find a correlation between the feature values and the application behavior to predict the benign application or malware type.

4.2.2 Feature Reduction

In order to perform run-time malware detection with minimal overhead while reducing the learning time and avoiding multiple runs, we intend to identify a minimal set of critical HPCs that can effectively represent the malware class behavior and are feasible to collect in a single run even on low-end processors with few HPCs. To precisely examine the critical microarchitectural events, we propose a two-level feature reduction approach. At the first level of reduction, we first manually and next algorithmically remove less significant features. This reduces the number of events to 16, as shown in Figure 4.4. At the second level, we propose using *Principal Component Analysis (PCA)* to reduce the remaining features to the most vital microarchitectural parameters to capture application characteristics. By applying the feature reduction methods, 8 most related microarchitectural events are determined and numbered in order of importance for malware detection. They are included in the malware detector models as input parameters. Next, we describe the details of the proposed feature reduction method.

First Level: Attribute Evaluation

a) Manual Feature Reduction. We first analyze the events manually and exclude events obviously not related to the target variable (malware behavior). Out of all 44 events, there are certain features provided by Linux kernel that are included as software events under *Perf.* We exclude a total of 12 events from the final selected features list. These events are alignment-faults, context-switches, cpu clock, cpu migrations, emulation-faults, major-faults, dummy, minor-faults, page-faults, and task-clock. In addition, events like cpu-cycles, and ref-cycles do not represent uniqueness in terms of program phase. Thus, they are also excluded from the final list.

b) Algorithmic Feature Reduction. Following the manual approach, we deploy *Correlation Attribute Evaluation* to rank 32 remaining features under WEKA. Correlation evaluation algorithm calculates Pearson correlation between each attribute and class as follows:

$$\rho(i) = \frac{cov(X_i, C)}{\sqrt{(var(X_i)var(C))}} \tag{4.1}$$

where ρ is the Pearson correlation coefficient. Xi is an input dataset of any performance counter event i. C is an output dataset contains different classes, "Malware" or "No Malware" in our case. Value of i represents any feature out of 32 features and cov(Xi; C) measures covariance between input dataset and output dataset. The var(Xi) and var(C) also measure variance of both input and output dataset, respectively. Equation 4.1 can be elaborated further as shown below:

$$\rho(i) = \sum_{k=1}^{n} \frac{(x_{k,i} - \bar{x}_i)(c_k - \bar{c})}{\sqrt{\sum_{k=1}^{n}(x_{k,i} - \bar{x}_i)^2 \sum_{k=1}^{n}(c_k - \bar{c})^2}} \tag{4.2}$$

where k is the number of input values; $x_{(k,i)}$ is k^{th} value in input dataset for feature i; and c_k is k^{th} value in output dataset. The mean of input for feature i is denoted by $\bar{x}_i$, and that for the output data by $\bar{c}$. Based on the ranking of ρ, top 16 features are selected for analysis. This algorithm finds correlation co-efficient for all 32 features as per above equation. We list top 16 features with the highest correlation coefficient value. These reduced features are shown in Figure 4.4. in which the branch-instruction has highest value of ρ than other features.

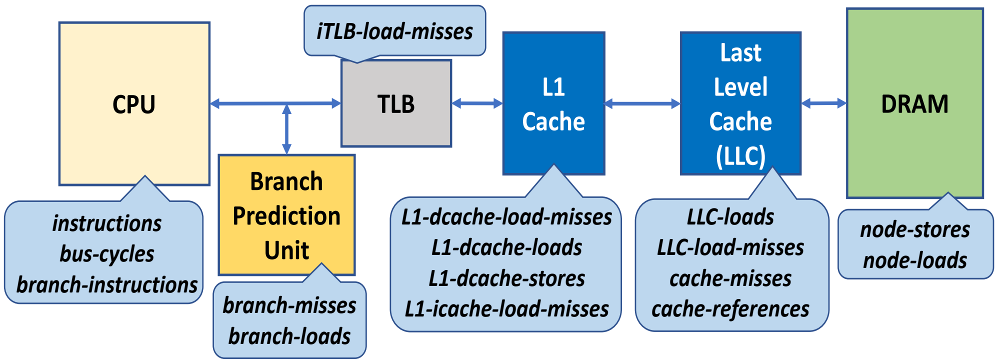


Figure (4.4) Reduced HPC features after first level of reduction

Second Level: PCA Analysis

For the second level of feature reduction, we apply PCA technique on the selected

Table (4.2) Prominent top eight HPC features for each class of malware

Rank	**Backdoor**	**Trojan**	**Virus**	**Rootkit**
1	*branch-inst*	*branch-inst*	*branch-inst*	*branch-inst*
2	*cache-ref*	*cache-ref*	*cache-ref*	*cache-ref*
3	*branch-miss*	*branch-miss*	*branch-miss*	*branch-miss*
4	*node-st*	*node-st*	*node-st*	*node-st*
5	branch-lds	cache-miss	LLC-lds	cache-miss
6	L1-icache-ld-miss	L1-icache-ld-miss	L1-dcache-lds	branch-lds
7	LLC-ld-miss	LLC-ld-miss	L1-dcache-st	LLC-ld-miss
8	iTLB-ld-miss	iTLB-ld-miss	iTLB-ld-miss	L1-dcache-st

16 HPC features from correlation analysis to select the top 8 features for detecting each class of malware in a single run. PCA analysis allows us to monitor and determine the most vital and distinct microarchitectural parameters to extract application characteristics [106]. PCA is a class of dimensionally reduction techniques that captures most of the data variation by rotating the original data to a new variable in a new dimension, commonly known as the Principal Components (PC) [78, 79, 106] that are uncorrelated to each other and are a linear combination of the original data. We employ PCA to project our 16 original gathered features into a new dimensional space to determine the most important features along different PC dimensions. As a result, for each malware class, we reduced the features to 8 most significant HPCs shown in Table 4.2 to capture the behavior of specific type of malware. The selected features are then supplied to each ML-based malware detector as input parameters for each malware class. These features include performance counters representing pipeline front-end, pipeline back-end, cache subsystem, and main memory behaviors and are influencing the performance of standard applications.

Solution for C2: Specialized HMD

4.2.3 Overview of the Proposed Two-stage HMD

In this section, we describe the details of 2SMaRT, the proposed two-stage run-time malware detection approach. As shown in Figure 4.5, in the first stage, an effective multiclass classification model is developed to estimate the type of the running application. Next, depending on the predicted application type, specialized ML classifiers are chosen for HMD with highest detection performance using limited number of events equal to available HPCs (only 4).

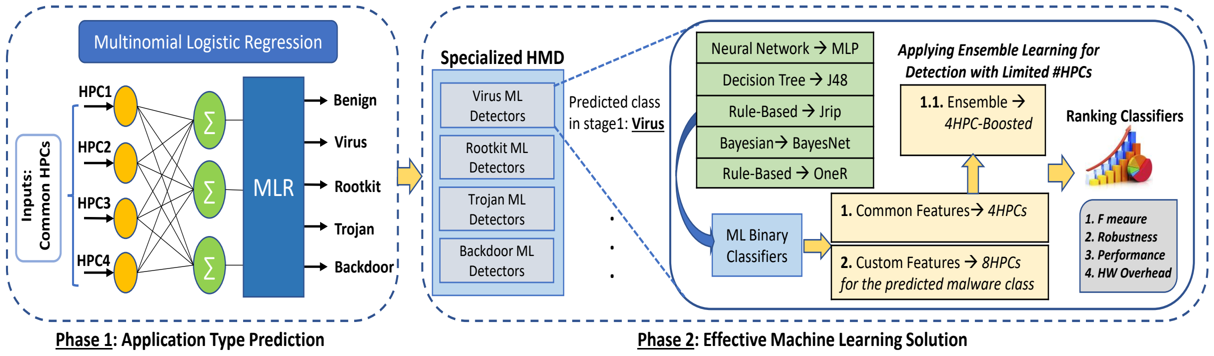


Figure (4.5) 2SMaRT overview, the proposed two-stage run-time hardware-assisted malware detection approach

Stage 1: Application Type Prediction. As each of the specialized detectors is trained to classify a different class of malware, they are each answering a different classification question. Initially, the system is unaware of existence of malware in the application, as such the use of specialized detectors cannot be effective. By analyzing the ML classifiers for malware detection across various malware classes, we observe (details are presented in Section 5.4) that the detection rate and performance of malware detectors are highly correlated to the class of malware (Virus, Trojan, etc.) infecting the system. To address this challenge, we primarily convert the basic binary malware classification into a multiclass problem, i.e. with more than two possible discrete outcomes. For this purpose, we propose to predict the behavior of the application (benign or a particular malware class) using a Multinomial Logistic Regression (MLR) technique shown in Figure 4.5.

The MLR classifier is a generalized linear model that predicts the probability of a discrete set of outcomes of categorically distributed dependent variable, given a set of independent variables which makes it a suitable classifier for predicting the class of applications. The MLR model is basically an extension of binary logistic regression that allows for more than two categories of the outcome variable. In this work, the output of MLR is corresponding to the set of feasible classes of applications including 5 individual classes, one for "benign" program and 4 malware classes namely "Virus", "Trojan", "Rootkit", and "Backdoor" classifying the four analyzed malware types. The MLR model is trained using extensive set of HPCs data captured by running various benign and malware programs. The inputs to the MLR consists of the top 4 low-level features shown in Table 4.2.

Let y denote the output of the MLR classifier, x denote the vector of input values, and $\varphi(x)$ denote a fixed nonlinear vector valued function of x. The probability of a particular output c under the MLR model is presented in below:

$$Pr(y = c|X, W) = \frac{exp(W_c^T \varphi(x))}{\sum_{c'}^{c} exp(W_c^T \varphi(x))} \tag{4.3}$$

The variable W_c contains the weights associated with output c, and $W$is a vector concatenating the weights across all outputs. Equation (4.3) maps a continuous real-valued argument $W_c^T \varphi(x)$ to a probability $y = c$ such that the probabilities across all of the possible outputs $\{1, \ldots, C\}$ sum to 1. A positive weight $W_{c_k} \in W_c$ implies that a positive value on input $\varphi_k(x) \in \varphi(x)$ increases the probability that $y = c$, and likewise a negative weight implies a decrease in probability. The logistic weights are estimated using training data, which in this case is an extensive set of HPCs information gathered by running various benign and different type of malware programs. During run-time, the probability of each class of application being executed is calculated and the MLR then selects the class that achieves the highest probability. The evaluation results of the proposed MLR model show that while the detection rate for the MLR using all captured 16HPC features is shown to be 83%, lowering the number of events to the 4 Common features does not reduce the detection rate of the multiclass classifier noticeably and results in detection rate of close to 81%. As a result, by using only 4HPCs the MLR model can predict the type of running application.

Common vs. Custom Features: As can be seen in Table 4.2, after the feature reduction, 4 out of the 8 identified microarchitectural events are remained the same across various classes of malware. These microarchitectural events are referred as Common features. These features include branch instructions, cache references, branch misses, and node-stores which provide a unique opportunity to identify the class of malware ahead of time for unknown malware. Along with the Common features, within each class of malware, we increase the number of HPCs from 4 to 8, referred as Custom features, tuning the ML classifiers individually for the corresponding malware class. To evaluate the effectiveness of the 2SMaRT at the second stage, we implemented all ML classifiers using the 4 Common and the 8 Custom HPCs listed in Table 4.2 for each malware class.

Stage 2: The Right Machine Learning Model. As observed previously, using simple MLR classifier does not provide a high run-time malware detection rate when using small number of microarchitectural events (equal to the number of available HPCs). To address the challenge of run-time malware detection with high detection rate and performance, we cascade a second stage of detection which is using various type of ML techniques. These ML classifiers are chosen based on the predicted class of malware by the MLR. As discussed earlier, the specialized ML classifiers are the ML models trained with the dataset of a specific malware class rather than with the data which comprises of patterns of all classes of malware. As seen from Table 4.1, no unique classifier is the winner in detecting all classes of malware. Thus, employing a specialized classifier which is the winner for the particular class of malware enhances the malware detection performance. These ML classifiers are shown in the second stage of Figure 4.5. The rationale for selecting these machine learning models are: First, they are from different branches of ML; regression, neural network, decision tree, rule-based, and ensemble learning covering a diverse range of learning models which are inclusive to model both linear and non-linear problems. Second, the prediction model produced by these learning algorithms can be a binary classification model which is compatible with the malware detection problem. As a result in 2SMaRT, when an application is running, the optimal set of HPC events are obtained first. This is followed by prediction of the class of application (benign or one of the classes of malware), based on which the customized ML classifier is utilized for detecting and classifying the malware class.

Solution for C3: Improved Malware Detection with Limited Number of Microarchitectural Events

Boosted-ML using Common HPCs. As discussed, today's microprocessors have limited number of HPCs that can be accessed simultaneously. To address the problem of utilizing only available HPCs i.e., even if it is less than what is deployed in previous solutions, we propose to use ensemble learning on top of the two-stage HMD framework. As depicted in Figure 4.5, in the second stage of 2SMaRT, we employ ensemble learning on-top of the traditional ML classifiers to improve the detection rate and performance of specialized classifiers using only 4HPCs to match the performance of classifiers using Custom features (8HPCs). Ensemble learning is a branch of machine learning used to improve the accuracy of general ML classifiers by generating a set of base learners and combining their outputs for final decision [107]. In Boosted 2SMaRT, we deploy Adaptive Boosting (AdaBoost) to construct the final classifier for run-time HMD. AdaBoost is one of the most commonly used ensemble learning for enhancing the performance of ML algorithms. In AdaBoost, each base classifier is trained on a weighted form of the training set in which the weights depend on the performance of the previous base classifier [107]. Subsequent models are trained and added until a minimum accuracy is achieved, or no further improvement is possible. Once all the base classifiers are trained, they are combined to produce the final classifier.

4.3 Experimental Results

In this section, we evaluate 2SMaRT across different malware classes in terms of detection rate, classification robustness, performance, and hardware overhead.

4.3.1 Evaluation Metrics

To examine the effectiveness of 2SMaRT, we consider four important metrics including detection rate (F measure), classification robustness, detection performance, and hardware overhead. For evaluating the detection rate we calculate the F measure (F score). F measure in machine learning is interpreted as a weighted average of the precision (p) and recall (r) where an F measure reaches its best value at 1 and worst at 0 which is formulated as follows:

$$FMeasure = \frac{2 \times (precision \times recall)}{precision + recall} \tag{4.4}$$

As shown, F measure accounts for both the precision and the recall of the classification to compute the score. The precision is the proportion of the sum of true positives versus the sum of positive instances and the recall is the proportion of instances that are predicted positive of all the instances that are positive. F measure is a more comprehensive evaluation metric compared with accuracy (percentage of correctly classified samples), since it takes both the precision and the recall into consideration. In malware detection applications, due to the level of desired security, it is crucial to minimize the number of false positives and detect as many as possible malware programs (positive cases). Hence, we weigh recall more. In other cases, we may want to be really precise not considering any sample to be positive unless we are definitely certain in which we weigh precision more. In addition to considering both precision and recall, F measure is also resilient to class imbalance in the dataset which is the case in our experiments.

F measure is not the sole factor to determine the efficacy of the malware detection and classification. For a comprehensive evaluation, we further calculate the robustness of classifiers with the aid of Area Under the ROC Curve (AUC). The AUC corresponds to the probability of correctly recognizing malware and benign applications. In this work, robustness is referred to how well the classifier distinguishes between binary malware and benign classes, for all possible threshold values. The AUC value of the best possible classifier is equal to 1, which means that we can find a discrimination threshold under which the classifier obtains 0% false positives and 100% true positives. Using the calculated AUC values, we define the product of F measure and robustness

(F$\times$AUC) as the detection performance metric to combine the impact of F measure detection rate and robustness (AUC) of malware detection and classification and account for both evaluation metrics. The hardware overhead metrics will also be discussed in next section. Due to space limitation, here we will show the F score, detection performance (F$\times$AUC), and hardware overhead analysis results.

4.3.2 Evaluation of Per-class Malware Detection

F Measure Evaluation: Table 4.3 presents the F score results of 2SMaRT across different classes of malware and number of HPCs (16, 8, and 4 HPCs). The results show that with the reduction in the number of HPC features used by ML classifiers, the F measure for majority of cases also drops. Also, it can be validated that no unique ML classifier achieves highest F score across all studied malware classes. Before feature reduction (16HPCs), most ML classifiers perform well, mostly providing above 85% detection rate. Feature reduction has noticeable impact on the several classifiers. However, OneR classifier performs well even after feature reduction. The reason that OneR classifier is not affected by feature reduction and shows almost constant F measure is that it only selects one HPC feature (branch instructions) to predict the malware.

Malware Detection Performance: Figure 4.6 illustrates the performance results of various ML classifiers in 2SMaRT using different number of HPC features. As can be seen in the results, most of the classifiers such as JRip, J48, and BayesNet deliver higher performance when supplied with 16 and 8 performance counters. For Multi-Layer Perceptron (MLP), due to overfitting the performance is degraded in some cases with the increase in the number of HPCs. However, techniques such as dropout can be employed, but at the cost of additional overheads. Similar to detection rate and robustness, the performance of OneR remains nearly invariant with number

Table (4.3) F measure of 2SMaRT detectors with and without ensemble learning

Class	**Backdoor**				**Rootkit**			
#HPC	16	8	4	4-Boosted	16	8	4	4-Boosted
BayesNet	96.1	86.5	85.55	86.6	85.6	78.37	72.49	74.22
J48	86.7	79.6	80.4	85.5	94.6	87.7	85.75	91.2
JRip	90.5	90	87.8	87.6	84.1	82.5	80.8	91.5
MLP	94.4	92.4	89.5	90	82.9	82.35	93.8	79.8
OneR	94	94	94	93.8	73.2	73.2	73.18	85.99
Class	**Virus**				**Trojan**			
#HPC	16	8	4	4-Boosted	16	8	4	4-Boosted
BayesNet	92.4	90.2	88	91.7	92.8	92.7	92.44	97.2
J48	94.7	94.5	93.2	96.5	98.8	98	93.2	97.3
JRip	93.6	93.1	93	93.9	98.9	98.2	93.3	94
MLP	68.1	67.6	94.7	95.4	98.6	96.7	98.9	98.9
OneR	97.1	90.2	89	94.8	92.7	92.7	92.7	92.7

of HPCs deployed. The proposed methodology achieves a performance of 77.3% on average when employing 16HPCs but drops to 71.3% when using only 4HPCs (with out ensemble learning) across all malware classes. The reduction is mostly due to the impact of reduced robustness when utilizing less HPCs.

Hardware Overhead Analysis: When it comes to choosing the right ML classifiers for hardware implementation, the detection rate is not the only factor for decision making. Design overhead (e.g. implementation logic area) and response time (latency) of ML classifiers also play a key role in selecting the cost-efficient HMD solution. While complex algorithms such as neural networks can deliver high accuracy and detection rate, they will also add significant overhead in terms of hardware implementation cost. Furthermore, given their complexity, they can be slow in detecting malware. As the software implementation of malware detection is slow (range of tens of milliseconds which is an order of magnitude higher than the latency needed to capture malware at run-time [20, 29]), we develop a hardware implementation of the 2SMaRT and analyze the associated overheads. We use Vivado HLS compiler

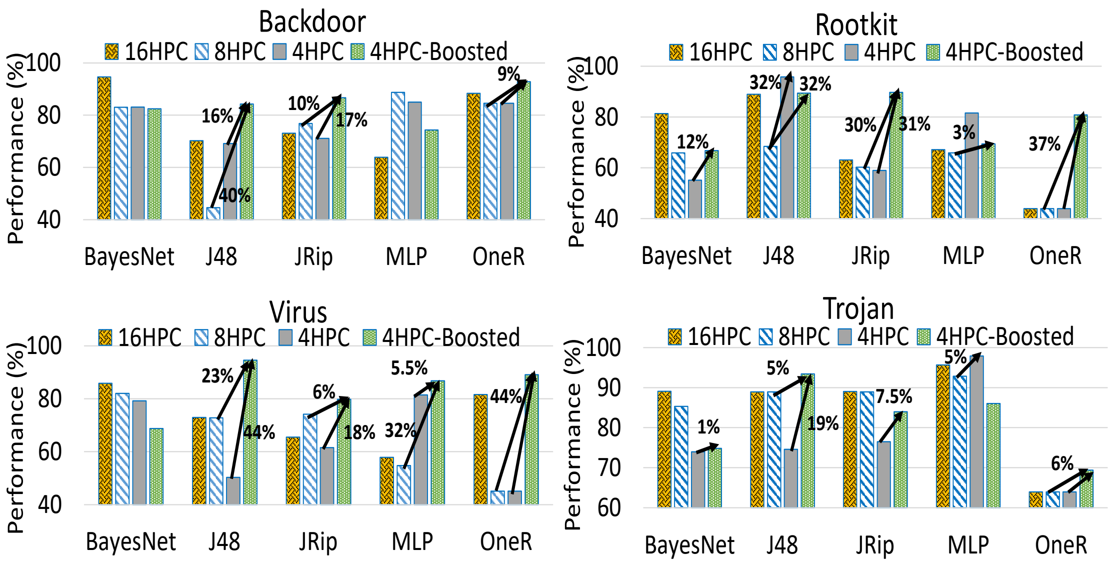


Figure (4.6) Malware detection performance of 2SMaRT for various ML classifiers across different malware classes

to develop the HDL implementation of the ML classifiers for Xilinx Virtex 7 FPGA. This analysis not only helps us to have an in-depth understanding of the complexity of the proposed solutions in terms of number of logic gates (which can be similarly proportional to an ASIC implementation), but also assists in assessing the design implementation overhead and cost in a heterogeneous FPGA+CPU architecture. As heterogeneous architectures have emerged in Cloud or even mobile environments, FPGA facilitates implementing reprogrammable low-level malware detection logic (ML model) that can detect malware by reading the CPU HPCs through a shared memory bus.

In order to compare hardware implementation costs, in Table 4.4, we report the results for general classifiers that use 8HPCs and 4HPCs for malware detection. Latency unit is represented in terms of the number of clock cycles (cycles @10 ns) required to classify the input vector. In order to compare the area overhead of the implemented hardware-based ML classifiers, we consider the OpenSPARC (FPGA) implementation as reference and calculate the area overhead relative to the core size. The area is the total number of utilized LUTs, FFs, and DSP units inside Virtex 7 FPGA. As can be seen from Table 4.4, the MLP algorithm, as expected, results in a significant area and latency overhead, as compared to other learning methods. Though the MLP achieves higher malware detection rate in many cases, the additional area overhead is around 30x compared to lightweight classifiers (OneR and J48). However, with the reduction in the number of HPCs used for classification from 8 to 4, the area decreases accordingly, as presented in Table 4.4.

4.3.3 Evaluation of Malware Detection with Limited Available HPCs

F Measure Evaluation: As observed in Table 4.3, for majority of the classifiers like BayesNet, JRip, and OneR by reducing the number of HPCs to 4 and applying ensemble learning techniques, a higher or similar detection rate to 8/16 HPCs-based HMD is achieved. This confirms the effectiveness of using ensemble learning to boost the F measure of ML classifiers. For instance, by applying AdaBoost on top of the proposed HMD, we achieve up to 98.9% detection rate for MLP in Trojan and almost 92% F score on an average across all other ML classifiers and malware classes compensating the accuracy loss resulted from feature reduction from 16HPCs and eliminating the need to run the application multiple times to capture the required HPCs.

Malware Detection Performance: The per-class results in Figure 4.6 clearly confirm the effectiveness of using ensemble techniques for performance improvement of

ML models with low number of HPCs. The key point here is that rather than extracting 16 or 8 HPC events which imposes significant implementation cost overhead to the system in terms of resource utilization, delay, and power consumption, in addition to requiring running application multiple times (as in each run only 4HPCs can be captured), it is more effective to collect fewer HPCs (only 4), depending on the ML classifier type, and boost the performance of the ML model with AdaBoost ensemble learning to improve the detection rate as well as robustness of malware detectors. It should be noted that the ensemble learning has a negative impact on the strong and complex classifiers such as MLP and BayesNet, as ensemble learning results in overfitting and non-convergence for such classifiers. For lightweight rule-based and tree-based classifiers (OneR, JRip, and J48) ensemble learning provides large improvement in performance. For instance, as shown, J48 in Backdoor class achieves 70% and 45% performance with 16HPCs and 8HPCs, respectively. However, we observe that reducing the number of vital HPC features to 4 and applying AdaBoost technique improves the detection performance to 84.64%.

The comprehensive analysis of the results across various ML models indicates the effectiveness of 2SMaRT by deploying ensemble learning combined with the lightweight classifiers such as JRip and J48 instead of non-boosted but costly ML solutions (MLP and BayesNet). For instance, as seen from Figure 4.6-Virus, the boosted tree-based (J48) and rule-based (OneR) ML classifiers when 4HPCs are used result in 44% performance improvement compared to the same non-boosted ML solutions. More importantly they outperform the performance of heavyweight MLP and BayesNet algorithms both in boosted and non-boosted cases.

Hardware Overhead Analysis: As shown in Table 4.4, ensemble learning with 4HPCs introduces area overhead for some classifiers. However, the introduced overhead is less than 3% compared to the general ML classifiers using 8HPCs. In addition, Table (4.4) Hardware implementation results of 2SMaRT

Model Used	**8HPC**		**4HPC**		**4HPC-Boosted**	
Classifier	Latency @10ns	Area (%)	Latency @10ns	Area (%)	Latency @10ns	Area (%)
BayesNet	14	11.5	6	7.7	56	13.6
J48	9	3	3	0.93	67	4.3
JRip	4	2.5	2	0.26	56	5.3
MLP	302	61.1	102	43.2	591	61.7
OneR	1	2.1	1	0.49	70	5.1

in some other classifiers we observe that by using ensemble learning with 4HPCs, the area overhead is significantly reduced, compared to the general classifiers using 8HPCs. Generally, ensemble learning generate models according to the data sets given and configuration of the algorithm. For instance, as reported in the previous section, the Boosted-MLP with 4HPCs gains 5.5% performance improvement for Virus, while as shown in Table 4.4, it interestingly shows close to 18% area increase compared to 4HPC-based ML classifier.

The key observation considering all different aspects of evaluation is that the AdaBoost ensemble learning aids to improve the performance of malware detection despite employing limited number of HPCs, which makes it an ideal fit for mobile and IoT-based systems. However, for large-scale systems which can simultaneously read large number of HPCs, and silicon area is not a constraint, the proposed 2SMaRT(non-boosted) with more HPCs can be employed for malware detection. To the best of our knowledge, there has been no prior work available that discusses the area cost for implementing ML classifiers as a function of HPCs. It can be argued that the number of HPCs can be increased during design time. However, there are several studies available such as [22, 25, 26, 87] that discuss and justify the limited number of HPCs due to complex microarchitecture of modern microprocessors. Due to deeper pipelines, complex prefetchers, modern cache design etc., implementing the

HPCs becomes a challenge in terms of counting multiple microarchitectural events and concurrently maintaining counter accuracy under speculative execution. Higher detection rate requires better and more complex hardware design. Thus, increasing the number of counters with limited detection rate does not appear to be a good trade-off. Compare to that, ensemble learning algorithm can be easily implemented on the programmable logic present in modern heterogeneous microprocessors to address the low accuracy and detection rate associated with limited number of available HPCs.

Table (4.5) Average performance improvement of 2SMaRT with AdaBoost across all four malware classes

ML Classifier	**8HPC→4HPC-Boosted**	**4HPC→4HPC-Boosted**
BayesNet	-6.25%	1.51%
J48	31.25%	18.2%
JRip	10.1%	18.75%
MLP	3.75%	-6.75%
OneR	24%	24%

Discussion: To qualitatively validate the efficacy of the 2SMaRT with and without AdaBoost, we first present the average performance improvement results for all studied malware classes in Table 4.5. The column '8HPC→4HPC-Boosted' in Table 4.5 denotes the average performance improvement when employing 4HPCs associated with AdaBoost compared to general 8HPCs in the proposed 2SMaRT. It can be observed that the detection performance with ensemble learning-based malware detectors using only the 4 Common HPCs outperforms the performance achieved when employing 8 or 4 HPCs without AdaBoost ensemble learning. However, a negative improvement is seen in the case of Neural Networks-based detector (MLP) when AdaBoost is employed due to overfitting. As seen, 1.51%-31.25% performance improvement is achieved with 4HPC-boosted malware detection compared to 8HPCs in which the tree-based JRip is achieving the highest improvement.

The results highlight the impact of deploying ensemble technique for performance improvement of ML classifiers with a lower number of HPC events eliminating the need for multiple runs of the same application to capture the required HPC data for effective malware detection. Thus, it is more effective to alternatively collect lower number of features (four), at lower power and performance cost to the system, and boost the performance of the ML classifiers with AdaBoost while facilitating run-time malware detection, given the availability of 4HPCs in the system.

2SMaRT vs. Ideal HMD: To evaluate the effectiveness of proposed run-time HMD employing only few features, we compare the ML models in 2SMaRT with an offline ideal HMD technique that uses all the 44 features for malware detection. Figure 4.7 shows the F measure and the robustness (AUC) results for the offline and proposed run-time malware detection along with the best ML solutions in each case. As shown, only an F measure degradation of 1% is observed in 2SMaRT as compared to the offline approach and nearly similar robustness is achieved for both offline and run-time solutions. This confirms the effectiveness of the proposed HMD using only 4 available HPCs which has a performance close to an ideal offline detector that accesses to all the 44 features.

2SMaRT vs. Single-Stage HMDs: Here, we present a comparison of detection rate of 2SMaRT against traditional single-stage HMD. Figure 4.8-(a) depicts the F measure results of HMD when utilized only first stage (represented as Stage1-MLR) against using the proposed two-stage HMD that accurately detects the type of malware ahead of time (referred as malware_4HPC_Boosted). The number of HPCs used for malware detection in Figure 4.8-(a) is the 4 Common features. As seen, using only the first stage (MLR) has the lowest F score of 80%. However, in 2SMaRT by using two levels of detection, the malware class is predicted ahead, and the proper

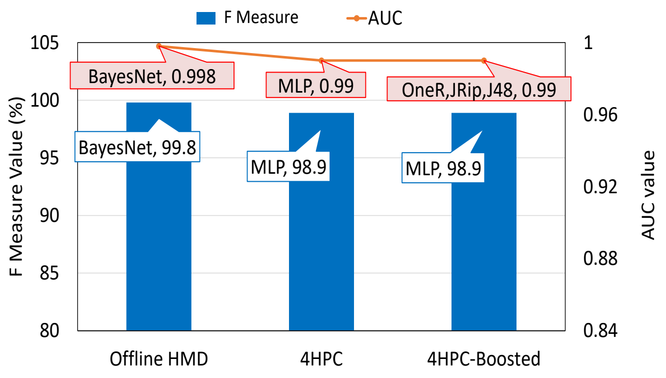


Figure (4.7) Optimal F measure and AUC of run-time 2SMaRT compared with offline HMD using all 44 HPCs

ML classifier trained for corresponding class is employed which improves the F score by up to 19%. This also signifies the advantage of 2SMaRT over prior single-stage HMDs such as [29, 30].

Furthermore, in Figure 4.8-(b) we compare 2SMaRT with a state-of-the-art single-stage HMD proposed in a recent work [29]. We compare 2SMaRT with [29] since it also employs different ML techniques using various number of HPCs to detect the malicious pattern of applications. As seen, the 2SMaRT with only 4HPCs achieves higher detection rate compared to [29] employing 4 and even 8 HPCs, due to the effectiveness of the two-stage run-time 2SMaRT methodology. Given the results, on an average close to 10%, and 9% improvement in detection rate is achieved with 2SMaRT using 4HPCs with and without ensemble learning compared to [29] using the same number of HPCs. In addition, interestingly the 2SMaRT with and without AdaBoost technique using only the Common 4HPCs outperforms the malware detectors in [29] with higher number of performance counters (8HPCs) by 9% and 8%, respectively.

Concluding Remarks. The proposed work outlines different challenges of run-time HMD that have been ignored in the existing works. These challenges include: 1)

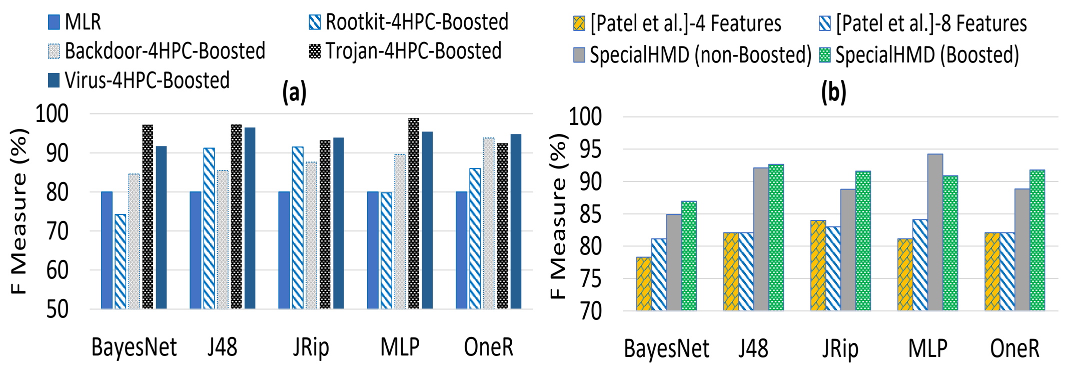


Figure (4.8) Comparison of 2-stage 2SMaRT with single-stage HMDs: a) comparison with 1 stage MLR, b) comparison of 2SMaRT using only 4 features with a recent HMD work

the type of key microarchitectural events to capture at run-time which varies across various malware classes; 2) no unique ML classifiers achieves high malware detection rate and performance across various types of malware; and 3) the number of microarchitectural events used by existing works are large, however, in reality the number of HPCs that can be monitored simultaneously is very limited. In response, we propose a multi-step solution of feature reduction by exploiting correlation between different microarchitectural events. Further, to address per-class run-time malware detection, 2SMaRT is proposed that first predicts the malware behavior, followed by employing a per-class specialized malware detector for improved detection performance. In addition, AdaBoost ensemble learning is cascaded as the last stage to address the challenge of employing small number of microarchitectural events equal to the limited number of available HPCs. 2SMaRT using only 4HPCs with a lightweight tree-based classifier (J48) boosted by AdaBoost improves the malware detection performance on an average by 31.25% compared to the HMD with 8HPCs, at the cost of only 1.3% additional area overhead.

Chapter 5: Stealthy Malware Detection using Low-Level HPC Features

The increasing complexity of modern computing systems leads into the growth of security vulnerabilities, making such systems a unique target for sophisticated attacks. Malware, a broad term for any type of malicious software, is a piece of code designed by cyber attackers to infect the computing systems without the user consent serving for harmful purposes such as stealing sensitive information, unauthorized data access, and running intrusive programs on devices to perform Denial-of-Service (DoS) attack [20, 91]. The rapid development of information technology has made malware a serious threat to computer systems.Given the exceedingly challenging detection of new variants of malicious applications, malware detection has become more crucial in modern computing systems.

Traditional software-based malware detection techniques such as signature-based and semantic-based methods [91, 95] mostly impose significant computational overhead to the system and more importantly do not scale well. Furthermore, they are unable to detect unknown threats making them unsuitable for devices with limited available computing and memory resources. The emergence of new malware threats requires patching or updating the software-based malware detection solutions (such as off-the-shelf anti-virus) that needs a vast amount of memory and hardware resources which is not feasible for emerging computing systems specially in embedded mobile and IoT devices.

In order to address the traditional malware detection shortcomings, Hardware-based Malware Detection (HMD), by employing low-level features captured by Hardware Performance Counters (HPCs), have emerged as a promising solution [20, 21, 30, 54]. HMD methods reduce the latency of detection process by order of magnitude with small hardware overhead [54]. The HPCs are basically special-purpose registers implemented into modern microprocessors to capture the trace of hardware-related events such as executed instructions, suffered cache-misses, or mispredicted branches for a running program [21, 29]. Recent studies on HMD have demonstrated that malware can be differentiated from normal programs by classifying anomalies using Machine Learning (ML) techniques applied on HPC features. ML-based malware classifiers can be implemented in microprocessor hardware with significantly low overhead as compared to the software-based methods, as detection inside the hardware is very fast (few clock cycles).

Malicious software attacks have continued to evolve in quantity and sophistication during the past decade. Due to ever-increasing complexity of malware attacks and financial motivations of attackers, malware trends are recently shifting towards *stealthy* attacks [33,34]. Stealthy attack is a type of cyber security attack in which the malicious code is hidden inside the benign application for performing harmful purposes [35]. An example of deploying stealthy malware is in document files in which the malware is capable of indirectly invoking other applications or libraries on the host as part of document rendering or editing.

The main purpose of stealthy attacks is to remain undetected for a longer period of time in the computing system. The longer the threat remains undiscovered the more opportunity it has to compromise computers and/or steal information before suitable detection mechanism can be deployed to protect against it. Stolfo et al. discovered a new type of stealthy threat referred as *embedded malware* [33,108]. Under this threat, the attacker embeds the malicious code or file inside a benign file on the target host such that the benign and malicious applications are executed as a single thread on the target system. The traditional signature-based antivirus application have shown to be ineffective in detecting embedded malware even when the exact signature of malware is available in the detector database. Embedded malware is potentially a serious security threat and accurate anomaly detection techniques must be developed to mitigate it. In this paper, we primarily focus on detecting stealthy attacks where malicious code is hidden inside the benign program, both executed as a single thread, making the detection more challenging.

The existing studies on HMD have primarily assumed that the malware is spawned as a separate thread while executing on the target host. However, in real-world scenarios malicious programs attempt to hide themselves within a benign application to bypass the detection mechanisms. In HMD methods the HPC data is directly fed to a detector, therefore, for embedded malicious code hidden inside the benign application, HPC data become contaminated, as the collected events include the combined benign and malware microarchitectural events.

In addition to the challenge of detecting embedded malware, prior studies on hardware-based malware detection performed limited study on malware classification accounting for the availability of a large number (e.g. 8/16) and diverse type of HPCs accessed at a time. However, today's microprocessors, even in the high-performance domain have limited number of HPC registers (2 to 8), due to several reasons including the design complexity and cost of concurrent monitoring of microarchitectural events. For embedded mobile and IoT domains, the number of HPCs that can be accessed simultaneously is even smaller.

In response to the aforementioned challenges, in this work, we propose an effective time series machine learning-based approach, referred as *CHASE*, to accurately detect the embedded malicious patterns inside the benign programs using only one HPC feature (branch instruction). To the best of our knowledge, this is the first work that addresses the challenge of detecting stealthy/embedded malware using hardware performance counters features at run-time.

The main objective of this work is to accurately detect the malicious application embedded inside the benign program using least number of microarchitectural events (only one HPCs) in which the traditional ML-based solutions are unable to detect them with even 8/16 features. Using an effective feature reduction technique, we first identify the most prominent low-level feature for embedded malware detection. Next, we propose a lightweight scalable time series-based Fully Convolutional Neural Network (FCN) model that automatically identifies potentially contaminated samples in HPC-based time series to distinguish the stealthy malware at run-time using only branch instructions as the most significant HPC event.

5.1 Background on Stealthy Malware Detection

5.1.1 Embedded Malware Detection

Stolfo et al. [33] proposed a new type of stealthy malicious attack referred as embedded malware. They introduced a method referred as file-print analysis, in which they calculated 1-gram byte distribution of a file and compared it to various models to identify the file type among PDF and DOC files. However, their approach is not capable of identifying the exact location of the embedded malware making it unfeasible for effective stealthy malware detection.

The work in [108] proposed static, and run-time dynamic methods for detecting malware embedded in Word documents. In static analysis, they deployed an open source application to decompose files and produced a similarity score for final classification. In dynamic approach, they employed sandbox-based tests to check OS crashes and unexpected changes to the underlying environment. However, it is acknowledged by the authors that their approach is not practical to be used as an independent malware detection scheme. The research in [109] used conditional markov n-grams to detect embedded malware.They deployed entropy rate, to quantify changes in n-gram distributions of a file and showed that the entropy rate gets significantly disturbed at malware embedding locations indicating its robustness for embedded malware detction.

5.1.2 Time Series Classification

Time series classification approaches can be divided into two different types, shapelet-based [110] and bag-of-pattern-based [111]. Shapelet-based approach attempts to find the subsequences that are the most discriminating of classes and deploys them to generate features for classification. They are comprehensible time series subsequences which are in some sense maximally representative of a class [110]. The work in [112] introduces a scalable algorithm to detect shapelets for classification. On the other hand, bag-of-pattern-based approaches attempt to discretize time series into a bag of symbols and deploy the distribution information for classification. The work in [113] proposes an approach called Bag-of-SFA-Symbols in Vector Space (BOSS VS) to efficiently classify time series. Li et al. [114] introduces a method named Bag-of-Pattern Feature (BOPF) to classify time series in linear time complexity. It has been shown that this approach can outperform the prior scalable time series classification approaches such as [112, 113].

Recently, several deep learning-based time series classification approaches are proposed [115–118]. These approaches often utilized ML techniques such as convolution neuron network and LSTM neuron network [119, 120] to extract the features from time series. However, compared with scalable time series classification approaches, these models often consist of large number of parameters incurring significant overhead and computational complexity to the computer system. As a result, in order to better evaluate and highlight the effectiveness of our proposed approach for embedded malware detection, we compare it with state-of-the-art ML-based HMD solutions as well as the BOPF approach as the most recent scalable time series classification method.

5.2 Motivational Case Studies

This section describes the motivations and challenges of hardware-based embedded malware detection using ML algorithms.

5.2.1 Challenge of Detecting Embedded Malware

Figure 5.1 illustrates the challenge of detecting embedded malware. Figure 5.1-(a) visualizes the complete benign and malware HPC data (described in details in Section 4), when the malware spawned as a separate thread, via t-distributed Stochastic Neighbor Embedding (t-SNE) algorithm [121], a widely used algorithm for visualizing high dimensional data. As seen, the marginal area between malware and benign program is large when malware spawned as a separate thread indicating that by using traditional ML models (prior works) the malware can be easily detected. However, the converted points of embedded malware data are mixed with each other in Figure 5.1-(b) depicting the impact of embedding malcode inside benign applications. The figure highlights the challenge of embedded malware detection indicating that due to the dense distribution of malware, traditional classification approaches cannot achieve

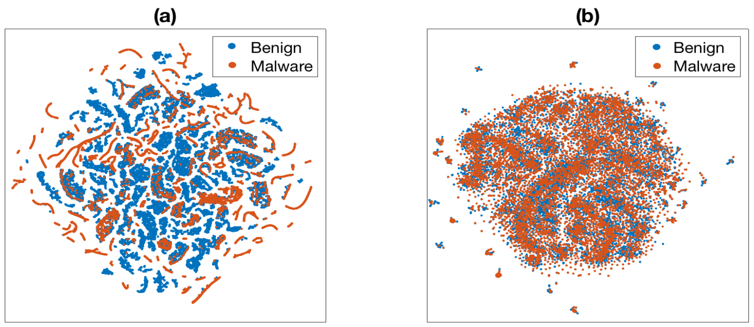


Figure (5.1) Visualizing the complete benign and malware dataset using t-SNE algorithm: a) malware spawned as separate thread b) malware embedded inside benign applications

a high accuracy in detecting embedded malware. Indeed, by performing nearest neighbor classifier in both complete and embedded malware dataset, the classifier can get accuracy of 90% in detecting the malware as a separate thread. However, the classifier only can achieve around 60% accuracy in embedded malware detection task when the malicious code is hidden inside the normal program.

5.2.2 Machine Learning for Hardware-Based Embedded Malware Detection

As discussed, in this work we intend to employ HPCs information to identify the behavior of running applications To verify the suitability of using HPCs for ML-based malware detection, we executed malware and benign applications on an Intel Nehalem architecture-based system as a case study to observe the behavioral patterns of HPCs. The benign application is selected from MiBench benchmark suite and the malware is a Backdoor application that can bypass the authentication process. The observed HPC traces of *branch instructions* for malware and benign applications are presented in Figure 5.2. The X-axis represents the time at which the HPC is monitored and the Y-axis represents the branch instruction HPC values. The profiling process shows that

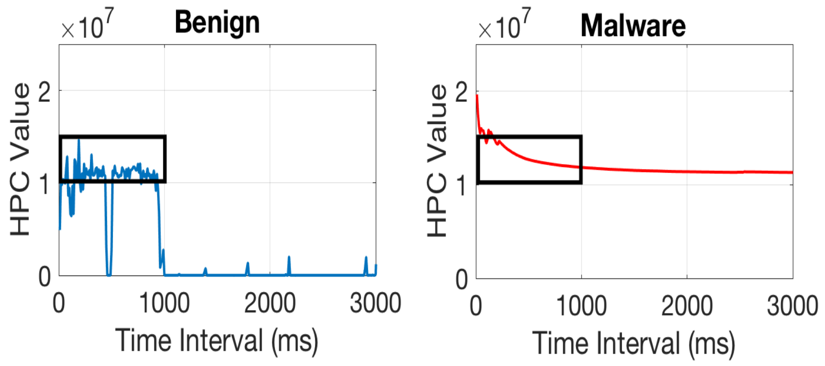


Figure (5.2) HPC traces of benign and malware for branch-inst.

if two different programs are executed on a processor, they generate relatively different HPC traces, providing a unique opportunity to detect the behavior of application. However, there exists an interesting observation in which if the malware is embedded inside benign program from 0ms to 1000ms time intervals, there is a significantly high possibility that the value of branch instructions for both benign and malware become equal which can mislead the traditional ML-based detectors in distinguishing the malicious behavior from benign applications highlighting the importance and necessity of developing an effective approach to accurately detect embedded malware.

5.3 Proposed Embedded Malware Detection Approach

In this section, we describe the proposed machine learning-based approach for effective hardware-based embedded malware detection. Figure 5.3 illustrates an overview of different steps for the proposed HMD framework. As shown, it comprised of different steps inclduing data collection and feature extraction, feature reduction, and the proposed ML-based embedded malware detector (CHASE) each described in details

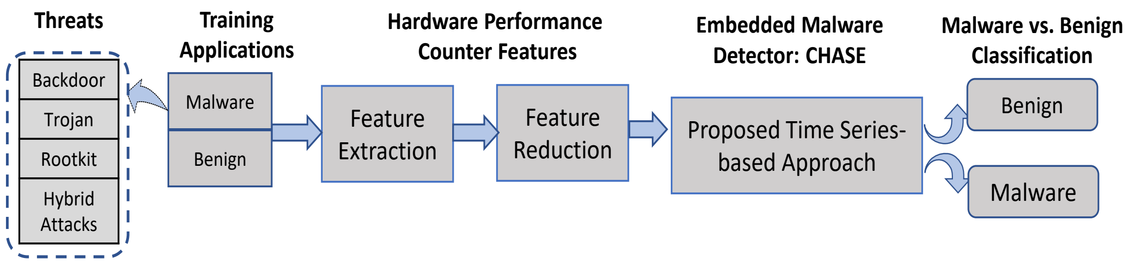

Threats
Backdoor
Trojan
Rootkit
Hybrid Attacks
Training Applications
Malware
Benign
Hardware Performance Counter Features
Feature Extraction
Feature Reduction
Embedded Malware Detector: CHASE
Proposed Time Series-based Approach
Malware vs. Benign Classification
Benign
Malware

Figure (5.3) An overview of different steps of proposed malware detection framework

in following subsections.

5.3.1 Data Collection

In our experiments, the benign and malware applications are executed on an Intel Xeon X5550 machine (4 HPC registers available) running Ubuntu 14.04 with Linux 4.4 Kernel and HPC features are captured using *Perf* tool available under Linux at sampling time of 10ms. *Perf* provides rich generalized abstractions over hardware specific capabilities. HPC-based profilers are currently built into almost every popular operating systems. *Linux perf* is a new implementation of performance counter support for Linux which is based on the Linux kernel subsystem *perf-event* and provides users a set of commands to analyze performance and trace data.

We executed more than 3500 benign and malware applications for data collection. Benign applications include MiBench and SPEC2006, Linux system programs, browsers, and text editors. Malware applications collected from virustotal and virusshare include Linux ELFs and scripts created to perform malicious activities and include 850 Backdoor, 640 Rootkit, and 1460 Trojan samples. HPC information is collected by running applications in an isolated environment referred as Linux Containers (LXC) which as mentioned before unlike common virtual platforms provides access to actual HPC data instead of emulating HPCs. In order to ensure that running malware inside the container does not affect the subsequent data collection by contaminating the environment the container is destroyed after each run.

5.3.2 Feature Representation

Determining the most significant low-level features is an important step for effective HMD. As there exists numerous microarchitectural events (for instance +100 in Intel Xeon), each of them representing a different functionality, collecting all features leads to a high dimensional data. Furthermore, processing raw dataset involves computational complexity and induces delay. As a result, we determine a minimal set of critical HPCs that can effectively represent the application behavior and are feasible to collect in a single run even on low-end processors with few HPCs. The top features with the highest correlation coefficient value and their descriptions are shown in Table 5.1. These events have mixture of branch related events representing core behavior and cache related events representing memory behavior. Next, we apply Principle Component Analysis (PCA) to find the best HPCs suited for training the ML-based malware detectors. We reduced the features to top 4 most significant HPCs to capture the behavior of specific class of malware. The feature reduction results indicate that the identified prominent 4 HPCs are the same across various classes of malware which include branch instructions, cache references, branch misses, and node-stores.

The proposed time series-based detection approach, CHASE, using only the most significant HPC feature, branch instructions, is able to detect the embedded malware inside benign application with high detection accuracy (will be discussed in details in Section 5.4). Branch operations are one of the non-trivial events as most of the malware [22] relies on branching operations for executing the malicious activity revealing the behavior of most malwares. Also, branch related counters can be accessed

Table (5.1) HPC features used for malware detection and their description

HPC event	**Description**
Branch instructions	# branch instructions retired
Branch-misses	# branches mispredicted
Cache misses	# last level cache misses
Cache-references	# last level cache references
L1-dcache-load-misses	# cache lines brought into L1 data cache
L1-dcache-loads	# retired memory load operations
L1-dcache-stores	# cache lines into L3 cache from DRAM
node-loads	# successful load operations to DRAM
node-stores	# successful store operations to DRAM
LLC-load-misses	# cache lines brought into L3 cache from DRAM
LLC-loads	# successful memory load operations in L3
Branch-loads	# successful branches

even in most of the low-end embedded and IoT devices, therefore making this type of microarchitectural events appealing to use for malware detection. Furthermore, it is hard to evade the branch instruction count due to the in-built exception handler that notifies the user regarding the exception and terminates the program or it can result in long stalls, eventually leading to termination of on-going executions.

5.3.3 Embedded Malware Threat Models

For modeling the embedded malware threats, we have considered persistent malicious attacks which occur once in the benign application with notable amount of duration attempting to infect the system. For the purpose of thorough analysis, we deployed various malware types for embedding the malicious code inside the benign application including Backdoor, Rootkit, Trojan, and Hybrid (Blended) attacks. For per-class embedded malware analysis, malware traces, taken from one category of malware, are randomly embedded inside the benign applications and the proposed detection approach attempts to detect the malicious pattern. Furthermore, the Hybrid threat combines the behavior of all classes of malware and hides them in the normal program. Persistent malicious codes are primarily a subset of Advanced Persistent Threat (APT) which comprised of stealthy and continuous computer hacking processes, mostly crafted to perform a specific malfunction activities. The purpose of persistent attacks is to place custom malicious code in the running benign application and remain undetected for the longest possible period. Persistent malware signifies sophisticated techniques using malware to persistently exploit vulnerabilities in the systems which usually targets either private organizations, states or both for business or political motives. The hybrid malware in our work represents a more harmful type of persistent threats in which the malicious samples are chosen from different classes of malware to achieve more powerful attack functionality seeking to exploit more than one system vulnerability.

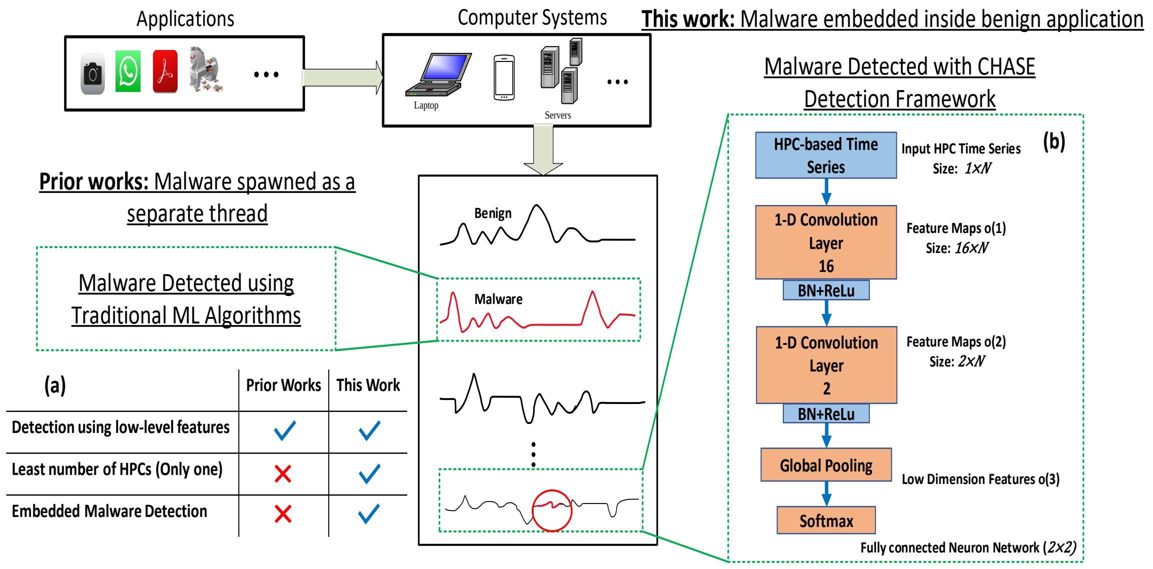
5.3.4 Embedded Malware Data Generation

With capturing interval of 10ms for HPC features, in order to model the real world applications scenario, we consider 5 sec. infected running application (benign application infected by embedded malware). For this study, 10000 test experiments were conducted in which malware appeared at a random time during run of a benign program. For this purpose, given a set of recorded benign and malware HPC time series, we randomly choose and concatenate multiple benign HPC time series to create a 3.75 sec. time series. Similarly, we then randomly choose and concatenate malware HPC time series to create a 1.25 sec. complete malware time series. This malware time series is the one that will embed into the created benign program. We randomly insert this malware time series into the benign program to form a 5 sec. contaminated application time series. In the experiment, three different sets of data including training, validation, and testing sets are created for evaluating the ML approaches. Each one contains 10000 complete benign HPC time series and 10000 embedded malware HPC time series. Since attacker can deploy unseen malware program to attack system, we create these three datasets with three groups of recorded malware HPC time series (consists of 33%,33%,33% of whole recorded data, respectively).

5.3.5 Overview of CHASE

As discussed, prior works on HMD mainly assumed that the malware is executed as a separate thread when infecting the system. This essentially means that the HPCs data captured at run-time inserted to the classifier belongs only to malware program. In real-world applications however, the malware can be embedded inside a benign application, rather than spawning as a separate thread, producing a more harmful attack. Therefore, the HPCs data captured at run-time could belong to both malware and benign applicationAs we will show in this work, this HPC data pollution could result in degradation of traditional ML classifier performance. In response to this challenge, we introduce CHASE malware detection framework which is based on a lightweight Fully Convolutional Neural Network (FCN)-based time series classification. Primarily, the proposed FCN-based approach attempts to automatically identify potentially contaminated intervals in HPC-based time series at run-time and utilizes them to distinguish the embedded malware from benign applications.

The overview of CHASE and its comparison with prior works is described in Figure 5.4. Intuitively, the network is a simplified version of neural network inspired from previous general convolutional neuron network-based time series classification models [115,118]. As shown, our proposed solution in this work is based on the least number of HPC features and targets detecting stealthy attacks that have been ignored in prior studies on hardware-based malware detection (Figure 5.4-(a)). Furthermore, as seen


Applications
Computer Systems
Laptop
Servers
This work: Malware embedded inside benign application
Malware Detected with CHASE Detection Framework
Prior works: Malware spawned as a separate thread
Malware Detected using Traditional ML Algorithms
Benign
Malware
HPC-based Time Series
Input HPC Time Series Size: 1xN
(b)
1-D Convolution Layer 16
Feature Maps o(1) Size: 16xN
BN+ReLu
1-D Convolution Layer 2
Feature Maps o(2) Size: 2xN
BN+ReLu
Global Pooling
Low Dimension Features o(3)
Softmax
Fully connected Neuron Network (2x2)

(a)	Prior Works	This Work
Detection using low-level features	✓	✓
Least number of HPCs (Only one)	✗	✓
Embedded Malware Detection	✗	✓

Figure (5.4) Overview of CHASE, the proposed embedded malware detection approach, and its contribution over prior HMD works

in Figure 5.4-(b), the network is created by stacking two 1-D convolution layers with 16 and 2 kernels, respectively. The size of kernel in these two convolution layers is 2 and 3, respectively. These convolution layers aim at selecting the subsequence of HPC time series for identifying the malware. Then a global average pooling layer is applied to converts the output of the convolution layer into low dimension features. These features are then fed into a fully connected neuron network to distinguish the embedded malware from benign applications.

Concretely, given a time series of HPC $x = x_1, x_2, \ldots, x_N$, where N is the length of the time series, in the first 1-D convolution layer, a output of k_{th} kernel can be computed by:

$$t_{i,k}^{(1)} = \sum_{j \in 1,2} w_{k,j,1} x_{i+j-1} + b_1 \tag{5.1}$$

where 2-d vector $[w_{k,1,1}, w_{k,2,1}] \in \mathbf{w}$ is the weight of k_{th} kernel and $\mathbf{w} = \{w_{k,j,1} | k = 1 \ldots, 16, j = 1, 2\}$ is a 16×2 matrix describes all weights of first layer. Given $t_k^{(1)} = [t_{1,k}^{(1)}, \ldots, t_{N,k}^{(1)}]$, a batch normalization function, $t_k^{(2)} = BN(t_k^{(1)})$, and a ReLu activation function,$o_k^{(1)} = ReLu(t_k^{(2)})$, are then applied. $BN(.)$ is a function which normalizes mean and variance of the $t_k^{(1)}$ to 0 and 1, respectively and ReLu activation function sets any negative value in $t_k^{(2)}$ to 0. $o_k^{(1)}$ is a N dimension feature map generated from the k_{th} kernel. We denote $o^{(1)} = [o_1^{(1)}, \ldots, o_{16}^{(1)}]$ as the output of the convolution layer. Intuitively, convolution layer converts original time series of length N into 16 different N dimensional feature map which capture different potential local features used to classify the input data [118]. The $o^{(1)}$ is then fed into next convolution layer with total number of kernels equal to 2. This layer summarizes $o^{(1)}$ into two different feature maps which can be computed via:

$$t_{i,k'}^{(3)} = \sum_{k=1}^{16} \sum_{j=1}^{3} w_{k',k,j,2} o_{i+j-1,k}^{(1)} + b_2 \tag{5.2}$$

where the weight of all kernels is a 3-d tensor $w_{k',k,j,2}$ of size $2 \times 16 \times 3$. For each $t_i^{(3)}$, $BN(.)$ and $ReLu(.)$ functions are further applied and four feature maps (denoted as $o^{(2)} = [o_1^{(2)}, o_2^{(2)}]$) are generated. Intuitively, stacking two convolution layers can increase the ability of model to detect complicated features. [118], which can increase the accuracy of the framework. Note that any positive value inside the $o_1^{(2)}, o_2^{(2)}$ indicates the potential HPC intervals which can be used to determine whether the input HPC time series contains an embedded malware. Similar to general convolution neuron network structure [118], we then conduct a global average pooling step to convert feature map $o^{(2)}$ into low dimension features. In particular, given a feature

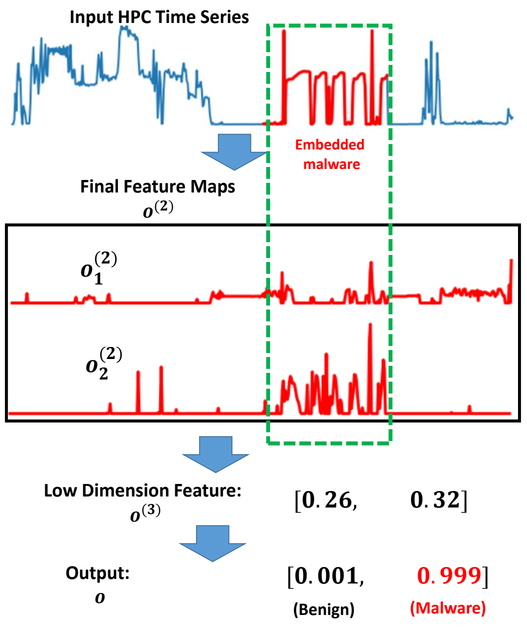


Figure (5.5) Case study for CHASE in detecting embedded malware

map of $o_k^{(2)} \in o^{(2)}$, we deploy the average value of all elements inside $o_k^{(2)}$ as the low dimension feature. As a result, this step converts $o^{(2)}$ into a 2-d vector (denoted as $o^{(3)}$).

Finally, $o^{(3)}$ is fed into a fully connected neural network with softmax activation function, a standard neuron network layer designed for classification [115, 118], for detecting embedded malware:

$$o = Softmax(W^T o^{(3)} + b_3) \tag{5.3}$$

where $Softmax(x) = \frac{e^{x_i}}{\sum_{k=1}^{2} e^{x_k}}$. Eq. (3) first converts $o^{(3)}$ into a new 2-d real value vector via linear transformation $W^T o^{(3)} + b_3$, where W is a 2×2 matrix and b_3 is a 2×1 vector. Next, all elements in the vector is mapped to [0,1] via $Softmax$ function. The final output is a 2-d vector $o = [o_1, o_2]$ which describes the possibility that the time series is benign or infected by malware (See Figure 3(c)).

Suppose we denote all the weights and the output of network as $\mathbf{\Theta}$ and $\mathbf{\Theta}(x) = [\mathbf{\Theta}_1(x), \mathbf{\Theta}_2(x)]$ respectively. Given an training dataset $\mathcal{D}$ and the network weights $\mathbf{\Theta}$, we update $\mathbf{\Theta}$ by minimizing the binary cross-entropy loss which can be computed by:

$$L = \sum_{(\mathbf{x_i}, y_i) \in \mathcal{D}} -y_i \log(\mathbf{\Theta}_1(\mathbf{x}_i)) - (1 - y_i) \log(\mathbf{\Theta}_2(\mathbf{x}_i))) \tag{5.4}$$

where $\mathbf{x}_i$ and y_i is the HPC time series and the associated ground true label of the i_{th} record in $\mathcal{D}$. And $y_i \in \{0, 1\}$ indicates whether the time series is benign or contains malware. Equation 5.4 can be minimized via standard back propagation algorithm, a widely used model for training various types of neural networks[118][115]. It primarily updates weights in neuron network by propagating the loss function value from the output layer to the input layer and iteratively minimize the loss function for each layers via gradient descent method. In this work, for each layer, the weights are optimized via Adam optimizer [122], a stochastic gradient descent method used to efficiently update weights of neuron network. In order to show the functionality of the proposed CHASE approach in identifying the malware embedded inside the benign program, a detection case study is presented in Figure 5.5. As shown, an HPC-based time series is the input to the classifier which contains an embedded rootkit malware (the embedded malware is highlighted in red). CHASE generates the two feature map $o_1^{(2)}$, $o_2^{(2)}$ via the proposed fully convolution neuron network. The $o_1^{(2)}$ and $o_2^{(2)}$ are then categorized as a 2-d feature vector $o^{(3)}$ by calculating the simple average of all the value in the feature map. In the given example, $o^{(3)}$ is equal to [0.26, 0.32]. This 2-d feature is then fed into a fully connected neuron network layer and the proposed CHASE analyze the input HPC time series and attempts to find that whether the input trace contains an embedded malware or not in which in this case it successfully identified the embedded malware with significantly high probability (0.999).

We implement the CHASE framework via Pytorch deep learning library. For evaluating CHASE framework using accuracy and F-measure (will be described in Section 5.4.1), it determines whether the input time series contains embedded malware by computing $argmax(o)$. Different from previous models, the proposed framework has small total number of kernels and layers which dramatically reduces the number of parameters and the cost for detecting malware in new HPC time series. For instance, in the latest neuron network introduced by [115], to classify a time series of length 500, the classification model needs more than 150,000 parameters, whereas the CHASE framework only contains 200 parameters. Having less parameters enhances the efficiency of the framework. Furthermore, the framework deploys less kernels compared with similar FCN-based classification model [118], which uses 3 convolution layers in which each layer has 128, 256, and 128 kernels, respectively.

5.4 Experimental Results and Analysis

In this section, we evaluate the proposed embedded malware detection approach across different attack types and evaluation metrics.

5.4.1 Performance Evaluation Criteria

In this work, CHASE framework is evaluated using precision, recall (a.k.a. sensitivity), F1-score, and detection accuracy (the overall rate of correctly classified samples). As described earlier, in binary classification techniques, true positives *(tp)* refer to the correctly classified/predicted positive samples, whereas true negatives *(tn)* are the number of the correctly classified/predicted negative samples. False positives *(fp)* indicate the incorrectly classified positive samples. Similarly, false negative *(fn)* metric is the number of incorrectly classified negative samples. On one hand, the terms positive and negative denote the classifier success, on the other hand true and false determines whether or not the prediction is matched with the actual application class (malware or benign).

The precision is the proportion of the sum of true positives versus the sum of positive instances. For instance, it is the probability for a positive sample to be classified correctly. The recall is the proportion of instances that are predicted positive and are also actually positive (i.e., tp) of all the instances that are positive. The F1-score, also known as F-measure or F-score, is the weighted harmonic mean of the precision and recall. F1-score reaches at its best value 1 and the worst score at 0. Since detection accuracy is not the only metric to determine the performance of the malware detection, we also evaluate CHASE using with the aid of Receiver Operating Characteristics (ROC) graphs. The ROC curve represents the fraction of true positives versus the fraction of false positives for a binary classifier as the threshold changes. We use the Area under the Curve (AUC) measure for ROC curves in the evaluation process. The AUC corresponds to the probability of correctly identifying malware and benign programs.

5.4.2 Evaluation of Proposed Approach

For the purpose of comprehensive evaluation, we compare our proposed approach with both the recent general time series classification approaches and recent proposed traditional machine learning-based HMD techniques. We studied two general time series classification methods including a nearest neighbour classifier (1-NN), a classical time series classification method, and Bag-of-Pattern-Features (BOPF) [114] classifier, which is the latest proposed scalable time series classification approach. Given

Table (5.2) Evaluation results for validation set

Type	Precision	Recall	F-score	Accuracy
Hybrid	0.85	0.893	0.876	0.868
Rookit	0.928	0.8844	0.9057	0.907
Trojan	0.912	0.8718	0.8917	0.894
Backdoor	0.8811	0.936	0.908	0.905
Average	0.892	0.896	0.8953	0.893

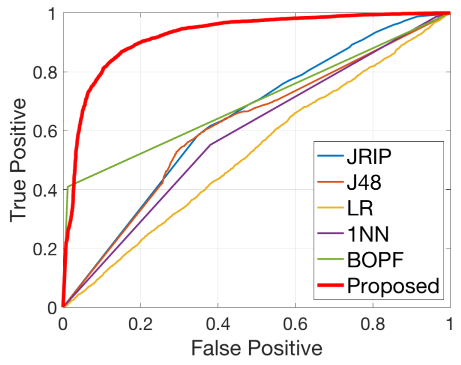


Figure (5.6) ROC Curve for Hybrid Embedded Malware detection

the input time series, 1-NN classifier will assign same class label to the input time series based on the most similar observed time series in training set. The similarity is measured by Euclidean distance. Bag-of-Pattern-Features (BOPF) based time series classification approach is one of the recent fast time series classification algorithm. BOPF has a very low time complexity compared with many existing time series classification approach and maintains a very high accuracy. Furthermore, we studied JRip, J48, and Logistic Regression-based HMD that are representing the rule-based, decision tree, and regression-based classifiers and have demonstrated high accuracy for detecting malware (spawned as separate thread) in recent prior works [20–22, 29].

Table 5.2 presents the evaluation results of malware detection for different classes of embedded malware for validation set analysis. The results show that our proposed lightweight two layer neural network-based solution can achieve average accuracy,

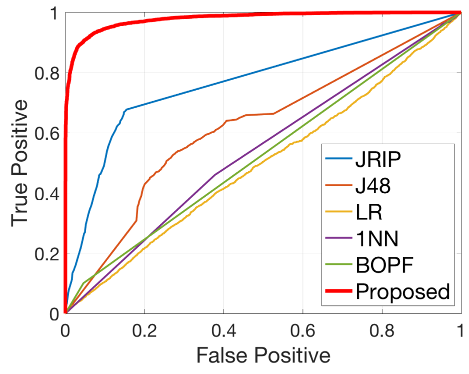


Figure (5.7) ROC Curve for Rootkit Embedded Malware detection

Table (5.3) AUC of testing set results for detecting various embedded malware

Method Type	CHASE	JRIP	J48	LR	1NN	BOPF
Hybrid	**0.922**	0.64	0.617	0.526	0.599	0.699
Rookit	**0.979**	0.77	0.62	0.497	0.541	0.526
Trojan	**0.93**	0.847	0.69	0.572	0.65	0.7859
Backdoor	**0.913**	0.731	0.537	0.507	0.603	0.676
Average	**0.936**	0.747	0.616	0.525	0.585	0.671

precision, recall and F-score of nearly 0.9 across all types of experimented embedded malware only by using the most prominent HPC feature (branch instructions). This makes the run-time detection of stealthy malware feasible which is primarily eliminating the need to execute applications multiple times to capture various low-level features suitable for malware detection.

Figure 5.6 and 5.7 illustrate the ROC graphs of the proposed approach compare to state-of-the-art HMD and time series classification techniques for two types of embedded malware, Hybrid and Rootkit. The correspondent AUC values for each embedded malware category are further presented in Table 5.3. A higher AUC value means that the ROC graph is closer to the optimal threshold and the classifier is performing better in terms of identifying the stealthy mwalre and classification of malware and benign applications. The ROC results clearly indicate the effectiveness of the proposed approach in this work as compared with prior ML-based malware detection and time series classification. As can be seen, our proposed approach, CHASE, achieves an average AUC value of nearly 0.94 across all experimented categories of embedded malware. Furthermore, CHASE significantly outperforms the traditional ML algorithms used in recent prior HMD works, JRip, J48, and LR, by up to 0.41., and further outperforms tested time series classifications approaches by up to 0.35.

For the purpose of thorough analysis and comparison, Table 5.4 presents the testing results of proposed light-weight embedded malaware detection approach in comparison with state-of-the-art works on hardware-based malware detection and time series classification. As seen, CHASE approach achieves highest accuracy and F-score for detection of all four different types of embedded malware. Overall, CHASE performance is outperforming the state-of-the-art HMD (e.g. Logistic Regression) by up to 0.4 in accuracy and 0.41 in F-score indicating the effectiveness of the proposed solution in detecting embedded malware.

Concluding Remarks. Malware detection at the hardware level has emerged recently as a promising solution to improve the security of computing systems. The existing works on hardware-based malware detection primarily assume that the malware is spawned as a separate thread. However, detecting stealthy attacks, malicious code embedded in a benign application, is a significantly more challenging problem in today's computing systems, since the malware hides itself in the normal application execution. In malware detection using low-level features, when the HPC data is directly fed to a machine learning classifier, embedding malicious code inside the benign applications leads to contamination of HPC information, as the collected features combine benign and malware microarchitectural events together. In response, in this work we proposed CHASE, a time series-based Fully Convolutional Neural Network framework to effectively detect the embedded malicious code that is hidden inside the benign applications. Our novel approach, using only the most significant HPC, branch instructions, can detect the embedded malware with 94% detection performance (Area Under the Curve) on average at run-time outperforming the detection performance of state-of-the-art hardware-based malware detection methods by up to 41%.

Table (5.4) Evaluation results for testing set for detection of various embedded malware

Embedded Hybrid Malware				
Method	Precision	Recall	F-score	Accuracy
CHASE	0.85	0.8303	**0.8579**	**0.8869**
JRip	0.627	0.583	0.604	0.618
J48	0.629	0.572	0.599	0.617
LR	0.52	0.494	0.507	0.518
BOPF	0.971	0.409	0.5761	0.6987
1NN	0.592	0.552	0.571	0.585
Embedded Rootkit Malware				
CHASE	0.955	0.898	**0.925**	**0.928**
JRip	0.814	0.677	0.74	0.76
J48	0.657	0.531	0.588	0.627
LR	0.503	0.469	0.486	0.503
BOPF	0.6885	0.1008	0.1759	0.5276
1NN	0.549	0.461	0.501	0.543
Embedded Trojan Malware				
CHASE	0.92	**0**.8164	**0.861**	**0.87**
JRip	0.844	0.784	0.813	0.819
J48	0.7	0.686	0.693	0.695
LR	0.556	0.548	0.552	0.554
BOPF	0.918	0.627	0.745	0.785
1NN	0.629	0.719	0.671	0.647
Embedded Backdoor Malware				
CHASE	0.889	0.833	**0.862**	**0.858**
JRip	0.83	0.577	0.681	0.729
J48	0.595	0.312	0.409	0.549
LR	0.501	0.432	0.464	0.509
BOPF	0.9257	0.3836	0.5424	0.6764
1NN	0.622	0.495	0.551	0.582

Chapter 6: Scheduling Challenges in Heterogeneous Architectures

In this section, we examine the suitability of applying effective machine learning techniques on captured runtime hardware-based information for addressing the performance vs. power consumption trade-offs and enhancing the energy-efficiency o heterogeneous multicore architectures. In particular, we show that hardware performance counter information can be effectively used for energy-efficiency prediction of multithreaded applications running on multicore heterogeneous architectures.

6.1 Machine Learning-based Approaches for Energy-Efficiency Prediction in CCAs

Heterogeneous multicores offer an effective solution to energy-efficient computing. To unlock the potential of the heterogeneity, software applications must adapt to the variety of different processors and make good use of the underlying hardware by executing workloads on the most appropriate core type. By running multithreaded applications on heterogeneous architectures, each thread is able to run on a core that matches its resource needs more closely than one-size-fits-all solution [62]. However, the effectiveness of heterogeneous architectures significantly depends on the scheduling policy and how efficiently we can allocate applications to the most appropriate processing core [56, 59, 65]. Applying ineffective scheduling decisions can lead to performance degradation and excess power consumption in such architecture [62–64, 123]. Commercially available heterogeneous architectures include Intel Quick IA [124], ARMs big.LITTLE [125], and Nvidia Tegra 3 that integrates a high performance big core with a low power little core on a single chip.

Composite cores architectures can provide further benefits by allowing the system to construct a right core for each running application. Several designs have been proposed that provide some level of dynamic heterogeneity. These proposals include Core Fusion [63], TFlex [60] and Composite Cores [62, 64]. In [62, 64] the concept of composite cores is proposed where a big core architecture can be dynamically decomposed into a smaller little-core architecture. The authors in [62] adapted the concept of composite cores in 3D by further enabling the core composition and decomposition at a low granularity of processor building blocks such as register file and load and store queue. Their proposed architecture allows multiple smaller cores to be composed together to build a larger core or vice versa, as needed.

While composite cores architecture provides more opportunity to construct the right core for the running applications, it is making the scheduling a difficult problem. Previous studies have mainly examined the advantages of using single threaded applications in CCAs [56, 62, 63, 65]. However, running multithreaded applications on a CCA and composing ideal processor architecture for energy-efficiency is a more challenging problem, considering the possible number of cores and threads and type of core micro-architecture. Furthermore, the challenge of how many and what type of core to compose for each multithreaded applications becomes even more complicated considering the impact of tuning parameters on energy-efficiency.

The main challenge for scheduling is to effectively tune system, architecture and application level parameters in CCA when running multithreaded applications. These parameters that are critical to performance and power include core type, voltage/frequency settings and the number of running threads. While there has been number of studies on mapping applications to heterogeneous architectures, no solution has been developed for mapping multithreaded applications into composite cores with its unique architecture. In addition, previous studies on mapping applications to multicore architectures have focused primarily on 1) homogeneous architectures, 2) static heterogeneous architectures where the number and type of cores are fixed at design time, and 3) configuring individual or a subgroup of tuning parameters at a time, such as applications thread counts [68, 71, 101, 126], voltage/frequency [56, 65], or core type [60,63,64,68,70,125] and they have ignored the interplay among all these parameters.

This study indicates that these parameters individually, while important, do not make a truly optimum configuration to achieve the best energy-efficiency on a CCA. The best configuration for a multithreaded application can be effectively found, only when these parameters are jointly optimized. Figure 6.1 illustrates the tuning parameters influencing in scheduling decision in a CCA. In this section, we focus on scheduling challenges by considering four tuning parameters including thread counts, voltage/frequency, core type, and application behavior. And the impact of power conversion efficiency on heterogeneous architectures will be discussed in details in Section 6.2.

In this research, through methodical investigation of power and performance, and comprehensive system and micro-architectural level analysis, we first characterize multithreaded applications on CCA to understand the power and performance trade-offs offered by various configuration parameters and to find how the interplay of these parameters affects the energy-efficiency. Our study is focused on a CCA where many little cores (base) can be configured into few big cores (composed) and vice versa. The experimental results support that there is no unique solution for the best configuration for different applications. Given the dispersed pattern of optimum configuration, we

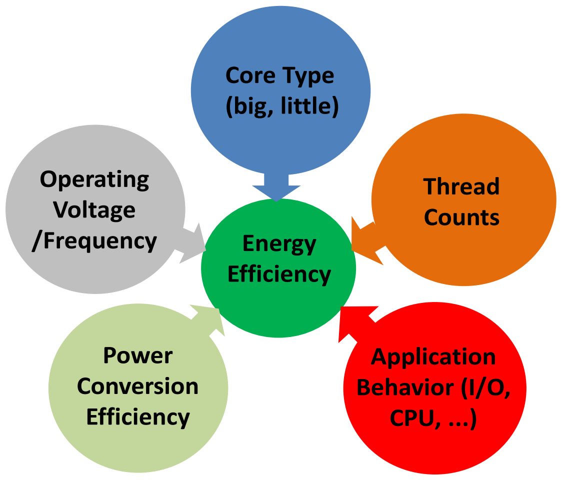

Core Type (big, little)
Operating Voltage /Frequency
Thread Counts
Energy Efficiency
Power Conversion Efficiency
Application Behavior (I/O, CPU, ...)

Figure (6.1) Tuning parameters influencing energy-efficiency in heterogeneous multicore architectures

develop various machine learning models to predict the energy-efficiency of parallel regions, and guide scheduling and fine-tuning parameters to maximize the energy-efficiency. As behavior of applications changes at run-time, we applied our prediction and tuning method at a fine-grained level of individual parallel region within an application.

We use five well-known machine learning models to build predictors based on the knowledge extracted from an extensive set of hardware performance data which are a good representative of application behavior for each parallel region within the training phase. The models are then used at run-time to predict the optimal processor configuration for each parallel region of a multithreaded workload to maximize the energy-efficiency. We analyze the proposed machine learning-based models in terms of their prediction accuracy, power overhead and implementation cost to understand their cost effectiveness [37, 40].

6.1.1 Motivation and Overview of our Approach

Multithreaded applications are comprised of number of parallel regions which are separated by serial regions. In this work, we refer to these parallel regions as Region of Interest (ROI). In a homogeneous multicore architecture and for conventional scheduling, all of these regions are processed on the same core type, same voltage and frequency, and number of threads, though not all regions may have the same preferences. Some of these ROIs may benefit from different configurations than the others to obtain the maximized energy-efficiency. As mentioned earlier, the main challenge for scheduling is to effectively tune system, architecture and application level parameters in CCA for the entire application as well as the intermediate parallel regions to achieve maximum energy-efficiency. In this work, similar to [62, 65], we are assuming that big cores (composed) are constructed by composing multiple little (base) cores.

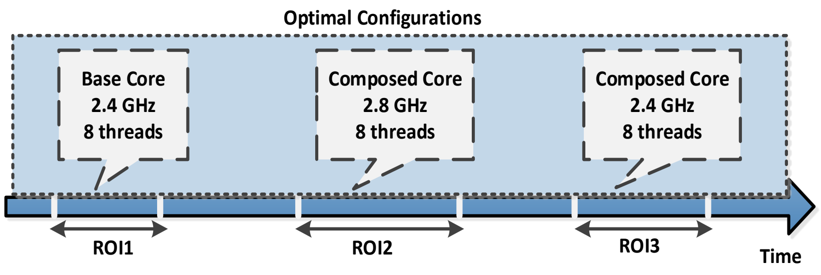


Figure (6.2) Optimal configurations in different parallel regions of an application

Figure 6.2 illustrates an overview of optimal configurations for an application selected from SPLASH-2 multithreaded benchmark suite with three parallel regions. In each ROI the best possible configuration for core type, operating frequency and thread counts that results in maximized energy-efficiency is specified. As can be seen, ROI1 needs to be run on the base core with 2.4GHz frequency and 8 threads, whereas for ROI2 in order to achieve the best energy-efficiency, we need to compose two small cores to make a big core. Moreover, the optimal operating frequency increases to 2.8GHz. The ability to accurately predict the optimal application, system and even microarchitecture parameters and adapt them to achieve the maximum energy-efficiency within different parallel regions of an application is the main motivation for this work. We develop several machine learning-based approaches which are able to predict the optimal setting of tuning parameters and change them to suit the specific requirement of each parallel region within the multithreaded application.

Figure 6.3 depicts our three-stage approach for predicting the right core type and application configuration when running a multithreaded application on composite cores architecture. Our machine learning-based approach begins from extracting microarchitectural data (referred as feature extraction), from different parallel regions of application to characterize the multithreaded workload. These data (or features) include the hardware performance counter data, which are representative of application behavior at run-time. Next, a machine learning-based predictor (that is built off-line) takes in these features and predicts the best configuration settings for a given parallel region.

For this purpose, we have implemented five well-known machine learning algorithms and compare them in terms of accuracy, power and area overhead to find to the most effective learning model which yields in optimized energy-efficiency. Finally, we configure the processor and schedule the application to run on the predicted configuration. In this work, the metric that we use to characterize energy-efficiency is the Energy Delay Product (EDP) which aims to balance performance and power consumption. We construct and compare five predictors using the machine learning algorithms described in the section V to guide the scheduling in a CCA.

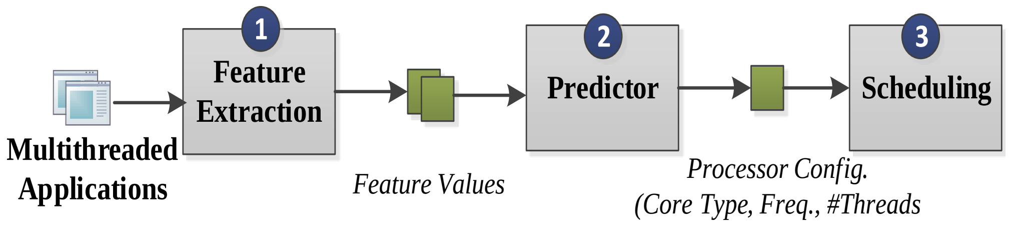


Figure (6.3) An overview of our approach for predicting the optimal configuration and scheduling the multithreaded application

6.1.2 Experimental Setup and Methodology

This section provides the details of our experimental setup. We use Sniper [127] version 6.1, a parallel, high speed and cycle-accurate x86 simulator for multicore systems. McPAT [128] is integrated with Sniper and is used to obtain power consumption results. We study SPLASH-2 [129] and PARSEC [130] multithreaded benchmark suites for simulation. For architectural simulation, we modeled a heterogeneous composite cores architecture based on the recently proposed work in [60, 63]. Our study is focusing on a CCA where two little cores (base) can be configured into one big core (composed) and vice versa. Figure 6.4 provides a conceptual overview of a four-core CCA. In this paper, we investigate two baseline heterogeneous CCAs which consist of multiple base and composed cores: 1) 8base/4comp, and 2) 4base/2comp. It is also important to note that for benchmark simulation we applied the binding (one-thread-per-core) model with threads == cores to maximize the performance of multithreaded applications [70, 71].

Table 6.1 shows the microarchitectural configuration of base (little) and composed (big) core of CCA in our experiment. We collect performance counters data on each architecture for characterization and drive the scheduling and mapping algorithms.

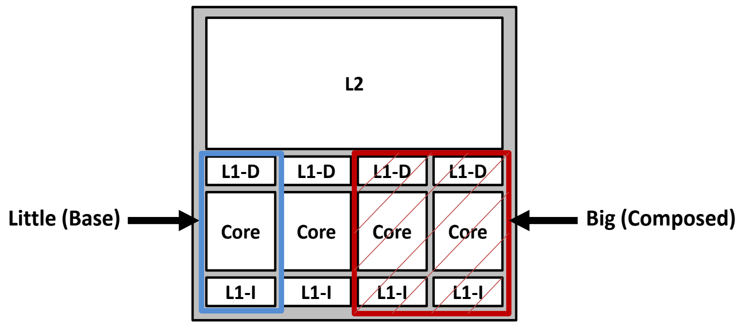


Figure (6.4) Conceptual structure of a four core CCA

Table (6.1) Architectural specification

Microarch. Parameter	Base	Composed
Issue-Commit width	2	4
INT instruction queue	16 entries	32 entries
FP instruction queue	16 entries	32 entries
Reorder buffer	32 entries	64 entries
Branch penalty	7cyc	14cyc
iL1-dL1 Cache	16KB/4-way/2cyc	32KB/4-way/2cyc
L2 Cache	4MB/8-way/32cyc	4MB/8-way/32cyc

We use these data to extract and evaluate the actual behavior of applications (I/O, CPU or memory intensive) for predicting energy-efficiency and assisting scheduling decision.

6.1.3 Characterization Results

In this section, we evaluate the applications performance and energy-efficiency sensitivity to tuning parameters of operating frequency, number of running threads, and the choice of microarchitectures (base vs. composed) in heterogeneous composite cores architecture. The studied parameters not only directly impact the power and

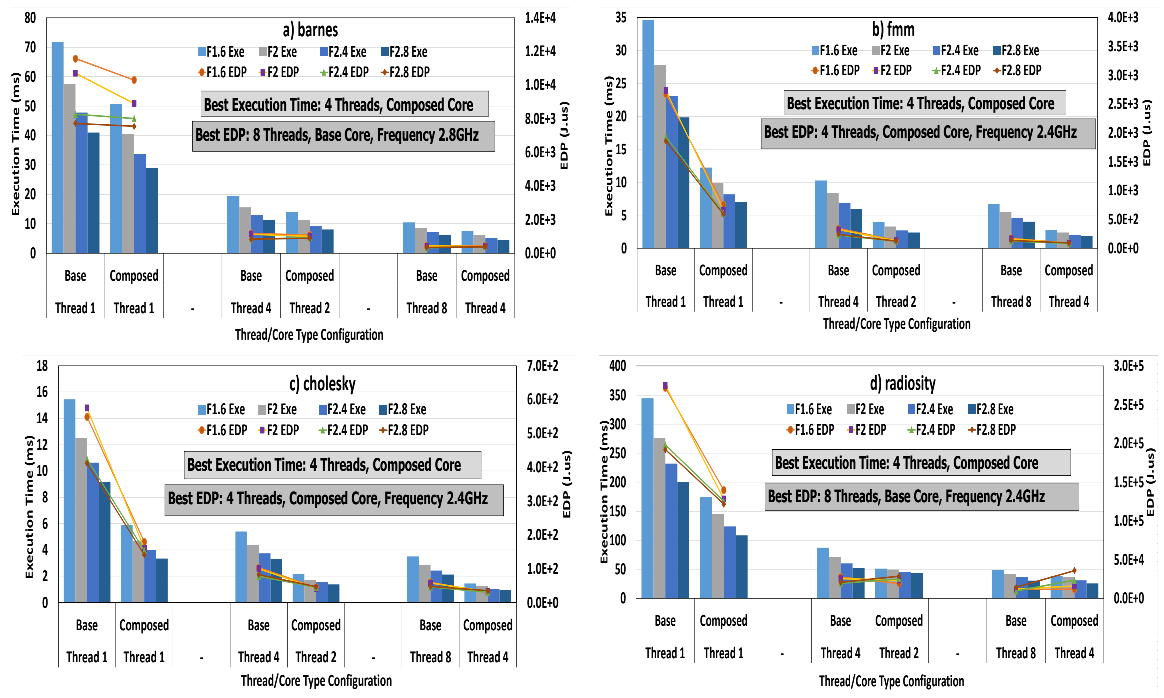


Figure (6.5) Execution Time and EDP of a) barnes, b) fmm, c) cholesky, d) radiosity with various Core Types, Threads, Frequencies

performance of the processor, but they also influence one another. The optimal system and microarchitecture configuration to maximize energy-efficiency varies based on the characteristics of the application, which all together influence the best tuning parameters. Therefore, it is essential to investigate the interplay of these parameters to guide the optimal mapping and scheduling decision in CCA. These observations form the basis for developing the prediction model presented in section V.

Note that the entire set of benchmark analysis results is quite extensive. Therefore, due to space limitations we only present the results for a limited number of representative benchmarks shown in Figure 6.5. This figure depicts the overall performance in terms of execution time (represented as a bar graph) and EDP results

Table (6.2) Optimal configurations with optimization target of EDP for different architectures

Benchmark	8Base/4Comp					4Base/2Comp				
	Best-base		Best-composed		Var. (%)	Best-base		Best-composed		Var. (%)
	Freq. (GHz)	*#Thread*	*Freq. (GHz)*	*#Thread*		*Freq. (GHz)*	*#Thread*	*Freq. (GHz)*	*#Thread*	
barnes	2.4	8	2.8	4	-444.8	2.8	4	2.8	2	-475.4
fmm	2.4	8	2.4	4	2.2	2.4	4	2.8	2	-2.9
cholesky	2.4	8	2	4	28	2.4	4	2.8	2	5.8
radix	2.8	8	2.8	4	-138.7	2.8	4	2.8	2	-236.2
radiosity	2.4	8	1.6	4	-102.8	2.4	4	1.6	2	-128.1
raytrace	2.4	5	2	4	-28.9	2.4	4	2.8	2	-152
fft	2	4	2	2	36.3	2	4	2	2	36.3
lu.cont	2.4	8	2.8	4	27.2	2	4	2	2	7.8
blackscholes	2.8	4	2.4	4	83.85	2.8	4	2.4	2	77.08
bodytrack	2	3	2	3	41.23	2	3	2	2	34.1
ferret	2	6	2	4	62.4	2	4	2	2	54.3

(represented as a line graph) for four different multithreaded benchmarks across different core types, frequencies and number of threads. In this section, first we discuss the impact of changing each parameter on energy-efficiency and next we perform a joint analysis to investigate the interplay of studied parameters and their influence on energy-efficiency in heterogeneous CCA.

Frequency Sensitivity. All benchmarks were simulated using a baseline composed core running with only a single thread. The operating frequency is swept from 1.6 GHz to 2. 8GHz with a step of 400MHz and the voltage is changed between 0.7, 0.8, 0.9, and 1V, respectively. As can be seen in Figure 6.5, some benchmarks are very sensitive to changing the frequency. For instance, in fmm and cholesky reducing the frequency almost linearly reduces the overall performance. Overall, as expected, as the frequency increases, the performance increases accordingly. The EDP results show that the higher frequency results in lower EDP.

The next observation is that increasing the number of threads interestingly reduces the sensitivity to frequency. In other words, increasing the number of running threads increases the performance gain due to parallelization. Consequently, the overall performance as the number of threads increases is more influenced by the speedup gain as a result of parallelism rather than operating at higher frequency. Moreover, the results show that the base core is more sensitive to frequency scaling than the composed cores. This is also an interesting observation as the composed core has a large pipeline, allowing it to tolerate performance cost due to alterations in cache access latency as a result of frequency scaling. Note that changing clock frequency changes the number of cycles it takes for the processor to communicate with the cache.

Core Type Sensitivity. In this section, the results are reported for a baseline configuration with a core running a single thread at the highest frequency of 2.8 GHz and operating voltage of 1V. The changing parameter is the core type, which varies between a base core and a composed core architecture. Core type demonstrates constant behavior with regards to EDP. As shown in Figure 6.5, there is a clear gap between the big composed and little base cores (in Thread1 and F2.8), with composed core having lower EDP. In these cases, the performance benefits of the composed core outweigh the energy savings of the base core.

Thread Counts Sensitivity. Finally, each benchmark is simulated with varying number of threads. In this step, each simulation was performed at the same frequency of 2.8 GHz and operating voltage of 1V, when changing the number of threads from 1 to 8. As shown in Figure 6.5 (2.8GHz cases with varying threads), increasing the thread counts leads to better performance. Moreover, there is a large gap between the EDP values of base and composed core at lower number of threads. Particularly, we observe that by increasing the thread counts, the corresponding gap between different core types diminishes and makes the base core competitive to the composed core in terms of EDP.

Joint analysis of (Core Type, Frequency, Thread Count). To understand the interplay among various tuning parameters and find the optimum configuration for maximizing the energy-efficiency, in this section all permutations of the parameters were simulated. We test four voltage/frequency settings on two core types and execute each multithreaded benchmark with 1 to 4 or 8 threads (depending on the core type), where each thread is assigned to a single core. These results are illustrated in Figure 6.5. Due to space limitations, we only demonstrate the results for 1, 4 and 8 running threads. Furthermore, the best evaluated execution time and EDP for each application is shown in each figure.

As mentioned earlier, we examine two different heterogeneous CCAs consisting of multiple base and composed cores: 1) 8base/4comp, and 2) 4base/2comp. Table 6.2 presents the optimal set of results for both architectures. This table includes benchmarks name, followed by the best core configuration parameters (Core, Freq., Thread) in terms of EDP across base and composed cores. We have also calculated the relative EDP variation for each benchmark, which indicates the relative difference between energy-efficiency of the best configuration parameters in base and composed cores. We quantify variation parameter $Var = (Base - best_{(EDP)} - Composed - best_{(EDP)})/Base - best_{(EDP)}) \times 100$. The Variation Parameter (Var) indicates whether it is justified to compose cores. For this purpose, a variation threshold is defined that decides what type of core architecture should be selected for executing the corresponding multithreaded application more energy-efficiently.

The user-defined threshold can be adjusted based on the architecture and available resources as well as the cost of core composition. Note that composing base cores to a big composed core is not free and comes with power as well as core utilization overhead. The core utilization overhead is in fact due to using additional cores to build bigger cores. When cores are composed to build a bigger core, fewer cores will be available for incoming or co-scheduled applications. In this work, we assume a 20% variation threshold. As a result, if the variation percentage between best-base and best-composed architectures is found to be lower than 20%, we use the base core for scheduling instead of composing to avoid power as well as core utilization costs.

As can be seen from Table 6.2, for most studied applications the best running thread count is equal to the maximum available cores. For instance, barnes performs with 2.4 GHz and 2.8 GHz on base and composed cores, respectively, while the best number of running threads on these two architectures are equal to 8 and 4, respectively. As shown, the variation for this application has negative value in some cases, which indicates it is more energy as well as core-utilization efficient to run the application on base core. Therefore, given that the variation value is lower than predefined threshold, rather than running the application on costly big composed core, we schedule the multithreaded application onto cost-effective little base core. From these observations, we conclude that while we can obtain significant performance gains, power and core utilization costs could be drastic when running application on big composed core. As a result, in those cases we choose the little base core as the optimal core configuration.

In order to perform a comprehensive EDP characterization of studied architectures, we classified all possible configurations (core types and number of threads) into four classes. The first two are Fully-Base and Fully-Composed configurations that are referred to cases in which the lowest EDP is achieved with full utilization of the base and composed core, respectively. In other words, the optimum number of threads is equal to the maximum number of existing base/composed cores. On the other hand, we use Partially-Base and Partially-Composed configurations when the best number of threads is lower than maximum available cores.

Parallel Region Analysis. As explained before, multithreaded applications are composed of a number of parallel sub-regions, which are separated by serial regions. Considering the application behavior, not all ROIs may have the same performance and power requirements. To illustrate the improvements offered by composite cores

Table (6.3) Optimal configurations in different parallel regions of radix and cholesky applications for EDP optimization

radix			
Region	Core Type	Freq.	#Threads
1	base	2.8	8
2	comp	2.4	8
3	base	2.8	8
4	comp	2.8	7
cholesky			
Region	Core Type	Freq.	#Threads
1	base	2.8	1
2	base	2.4	8
3	comp	2.8	8

architectures, studied multithreaded benchmarks were modified to monitor the behavior of each parallel region within the application. Simulation markers were placed at different sections of the benchmarks that would be simulated as individual parallel regions. All simulations were then run again using all permutations of the configuration parameters discussed earlier, including voltage/frequency, core type, and thread counts. Due to space limitation in our paper, we only present simulation results from parallel regions of four benchmarks which are reported in Table 6.3 and 6.4. Each table shows the optimal configuration in terms of EDP for each of the ROIs listed for a given benchmark. It can be clearly seen that every ROI within the application does not benefit from the same configuration parameters.

In order to clarify this point, here we look at an example, the radix benchmark, in more depth. Radix benchmark was instrumented with four individual sub-regions. As results show, all four sub-regions have different set of configuration parameters to achieve the best EDP. The core type varies between base and composed, and the frequency varies between 2.8GHz and 2.4GHz. Also, the thread counts changes between 7 and 8. This example demonstrates the importance of using right tuning

Table (6.4) Optimal configurations in different parallel regions of fft and lu.cont applications for EDP optimization

fft			
Region	Core Type	Freq.	#Threads
1	base	2.8	8
2	comp	2.8	8
3	comp	2.8	1
4	comp	2.4	8
5	comp	2.8	8
lu.cont			
Region	Core Type	Freq.	#Threads
1	base	2.4	8
2	comp	2.8	8
3	comp	2.4	8

parameters for best EDP, not only for the entire multithreaded application, but also even for each parallel region within the application.

Overall, the results show the importance of concurrent optimization at the application, system and microarchitecture levels at coarse-gained level of application or even fine-grained level of individual parallel regions within the application to maximize the energy-efficiency. The challenge is to develop a technique that automatically determines the best configuration for any given multithreaded applications and optimization goal and perform the tuning at a fine-grained level of individual parallel region within the application. In the next section, we will describe our proposed approach using various machine learning models.

The diversity of optimum configurations across various applications and their parallel ROIs demonstrates that when running a given multithreaded workload on a heterogeneous CCA, depending on the application and energy-efficiency optimization metric, different configuration parameters (Core Type, V/Freq., Thread) lead to the best energy-efficiency. The configuration also changes at run-time for each parallel region within the application. In other words, experimental results support that there is no unique solution as the best configuration across various parallel regions of an application. This dispersed pattern of optimum results implies the necessity of developing a prediction method to guide scheduling decision of multithreaded applications onto heterogeneous composite cores architecture in order to improve the energy-efficiency.

6.1.4 Predictive Modeling

Recent studies have proposed linear regression modeling to predict the power and performance of a processor at run-time. As mentioned earlier, in this work we implement different machine learning models to estimate the EDP. Table 6.5 shows five machine learning models that we use for predicting the best processor and application configuration to deliver the lowest EDP. These models include least square median, linear regression, Multi-layer Perceptron (an artificial neural network model), and two decision tree techniques namely REPTree and M5Tree. We selected these five classifiers for two reasons. First, they are from three different types of machine learning methods; regression, neural network, and decision tree, covering a diverse range of learning algorithms which are inclusive to model both linear and non-linear problems. Second, the prediction model produced by these learning algorithms is deterministic which is compatible with our numerical target variable, EDP.

All of ML classifiers were implemented using WEKA machine learning toolkit [75]. The inputs to our models are a set of features extracted from application profiling at run-time. While these microarchitectural features are accessible through our simulator infrastructure, in a real hardware, they are also accessible through hardware performance counters. These architectural information used for differentiating

Table (6.5) ML classifiers used for prediction

ML Classifier	Learning Type
LinearReg	Regression
LeastSqMed	Regression
MultiLayerPercep	Neural Network
M5Tree	Decision Tree
REPTree	Decision Tree

Table (6.6) HPCs data used for training the classifiers

Category	Hardware performance counter
Memory subsystem	L1 D-cache access, L1 D-cache miss, L1 I-cache access, L1I- cache miss, L2 cache access, L2 cache miss, I-TLB miss, D-TLB miss
Instructions	Integer instruction issue, Integer floating point issue
Branch	Branch instruction, Branch misprediction

workloads behavior are listed in Table 6.6. The output of our models is the optimal EDP that corresponds to the type of core to use for running the application, the clock frequencies of the core and thread counts. The goal is to use these models to find the best configuration parameters for individual parallel regions within an application and understand how these parameters should be adapted at run-time in a heterogeneous architecture. Developing and deploying these models include a 3-step process for supervised machine learning as follows: (i) generating training data, (ii) developing a predictive model, (iii) using the predictor at run-time for every parallel region of the application.

6.1.5 Training the Predictor

Figure 6.6 depicts the process of training multithreaded applications to build a machine learning classifier for EDP prediction. Training involves finding the best processor and application configuration and extracting feature values for each training workload, reducing the extracted features to the most vital performance counters, and developing a learning model from the training data. It is important to note that the input variables in our classifiers are performance counters extracted from different parallel regions of application, and the output variable is the EDP for a given set of tuning parameters.

Generating Training Data. To derive the prediction model for energy-efficiency, we need to develop a data set to train the prediction model. We applied our ML classifiers on extensive set of SPLASH-2 and PARSEC multithreaded benchmark suits. The studied multithreaded applications represent diverse compute, memory and I/O intensity behavior. We exhaustively execute each training benchmark, with different processor and application configurations and record the configuration with lowest EDP. We have also collected the hardware performance data from 20 parallel sub-regions of these applications and use them to build regression, neural network and decision tree classifiers for predicting the EDP.

As mentioned earlier, the extracted features from each of the parallel ROIs appropriately represent the application behavior during these execution phases. For each parallel region within an application, we collect twelve hardware performance counters data listed in Table 6.6 on all possible configurations of core types, voltage/frequency operating points and number of threads. We applied machine learning models on the studied benchmarks using these performance counters and profiling information for predicting energy-efficiency. In order to validate each of our classifiers, we applied the percentage split method to divide the dataset into two sets, using 70% (known applications) of the data to train the model and 30% (unknown applications) to simulate and test.

Developing Predictive Models. The features together with the processor configuration are supplied to each learning algorithm. The learning algorithm attempts

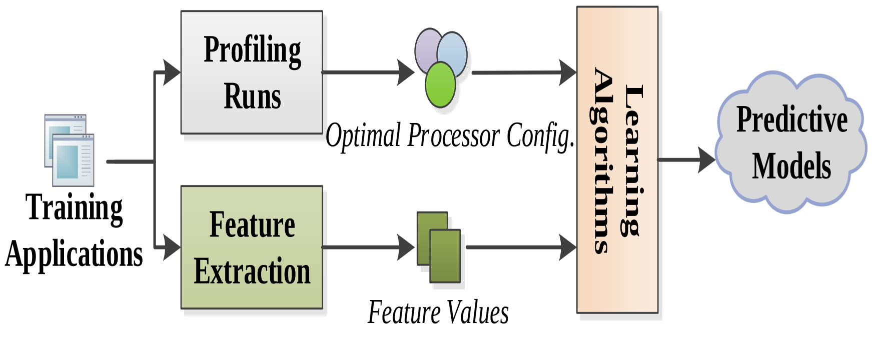


Figure (6.6) Training process for machine learning predictive models

to find a correlation from the feature values to the optimal configuration and predicts the energy-efficiency that corresponds to each configuration. As described in previous section, our predictors are based on a number of features extracted from the hardware performance counter attributes. One of the key aspects in building an accurate predictor is finding the right features to characterize the input data. We started from twelve performance counters that can be collected from parallel regions of each multithreaded application.

These features include performance counters representing pipeline front-end, pipeline back-end, cache subsystem, and main memory behaviors and are influential in the performance of standard applications. As shown in Figure 6.6, after feature extraction we use Principle Component Analysis (PCA) and correlation analysis on our training set to monitor the most vital micro-architecture parameters to capture application characteristics. By applying the attribute reduction method, we determine the four most related performance counters including L1 D-cache access, L2 cache-access, L2 cache-miss and branch misprediction. These performance counters are included in our model as input parameters.

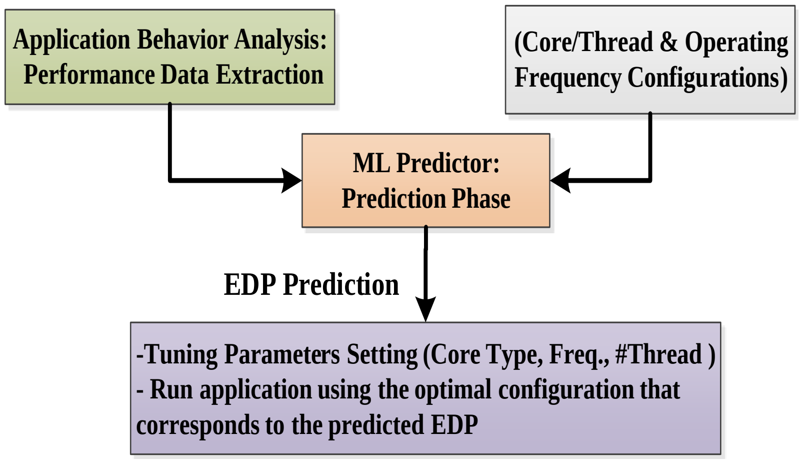

Application Behavior Analysis: Performance Data Extraction
(Core/Thread & Operating Frequency Configurations)
ML Predictor: Prediction Phase
EDP Prediction
-Tuning Parameters Setting (Core Type, Freq., #Thread)
- Run application using the optimal configuration that corresponds to the predicted EDP

Figure (6.7) Energy-efficiency prediction and tuning parameters configuration

6.1.6 Prediction Phase

Once we build our ML predictor, we deploy it for predicting the energy-efficiency of various configurations. Figure 6.7 provides an overview of EDP prediction process and tuning processor and application parameters using the trained machine learning classifiers. This prediction model predicts continuous values representing EDP as a function of performance counter inputs and tuning parameters, which is then used to make the scheduling decisions at run-time.

In particular, in this phase we run a multithreaded application with the most aggressive configuration setting where all tuning parameters are set at max (maximum thread counts, highest frequency, and for composed core). It is important to note that this would be the fastest way to collect run-time features of an application, since this is done for most aggressive configuration that corresponds to the highest performance. At run-time, we extract the hardware performance counters by profiling the application for each parallel region.

The ML classifier then takes the key performance features and configuration settings as inputs, and outputs the system energy-efficiency for each configuration. The configuration corresponding to the lowest EDP is then used to tune and schedule the application. Thus, at run-time, given an unknown application, the predictor can predict the EDP of all possible configurations based on a single run data. The configuration corresponding to the lowest estimated EDP is then selected for the run. The predictive models by observing run-time behavior of a multithreaded application running with a specific configuration, predicts the right configuration parameters to achieve the maximum energy-efficiency. It is important to note that each predictor can be simply trained for other objectives such as ED2P optimization.

6.1.7 Experimental Results

In this section, we present the evaluation results for our machine learning predictors. We compare these learning techniques in terms of EDP prediction accuracy and hardware implementation cost. As mentioned in previous section, in this work we focus on analyzing two heterogeneous CCA consisting of 8base/4comp, and 4base/2comp cores.

In order to perform a comprehensive EDP characterization of studied architectures, we classified all possible configurations (core types and number of threads) into four classes. The first two are Fully-Base and Fully-Composed configurations are referred to cases in which the best energy-efficiency is achieved with full utilization of the base and composed cores, respectively. In other words, the optimum number of threads is equal to the maximum number of existing base/composed cores. On the other hand, we use Partially-Base and Partially-Composed configurations when the best number of threads is lower than maximum available cores.

Optimal Configurations. Figure 6.8 and 6.9 show the distribution of the optimal configurations for two studied composite core architectures. It demonstrates how the distribution of optimal configurations changes across all studied parallel regions (for all studied applications). In both studied architectures, Fully-Base configuration running at a medium frequency of 2.4 GHz yields the lowest EDP for a majority of studied cases. However, composing cores yields the lowest EDP also for a noticeable number of cases; 37.5% in 8Base/4Comp and 12.5% in 4Base/2Comp.

From Figure 6.8, we observe that overall 62.5% of studied cases benefit from Fully-Base configuration which yields the lowest EDP. Moreover, as shown in Figure 6.9, for all studied cases in 4Base/2Comp architecture, no Partially-Composed or Partially-Base configurations was selected as the optimum configuration; i.e. in this architecture for maximum energy-efficiency, all cores, either base or composed, need to be allocated to the running multithreaded application. Also, the optimal clock frequency found to be lower than the maximum frequency for the majority of studied cases in both architectures (more than 87.5%). This diagram shows the need to adapt the microarchitecture and application settings to different multithreaded applications for energy-efficiency optimization.

Overall, the results confirm a large disparity in the optimum configuration across a large range of tuning parameters, highlighting the importance of developing a predictive method. Also, as the number of base cores in the studied architecture increases, the importance of composing cores to make a larger core is highlighted; more than 32% of studied cases in 8Base/4Comp vs. 12.5% in 4Base/2Comp are corresponding to composite cores.

Prediction Accuracy. To evaluate the accuracy of our prediction model, we calculate the value of relative mean absolute error defined as [(estimated value-actual value)/actual value]100. This metric indicates the relative difference between the predicted and observed maximum EDP. Figure 6.10 shows the energy-efficiency accuracy comparison of the machine learning classifiers used for predicting the EDP. As shown, M5Tree achieves close to 94.5% accuracy and outperforms all other classifiers

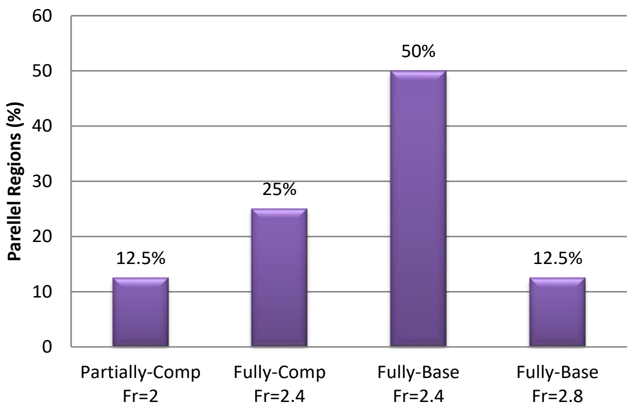


Figure (6.8) The distribution of the optimal configurations for EDP for 8Base/4Composed architecture

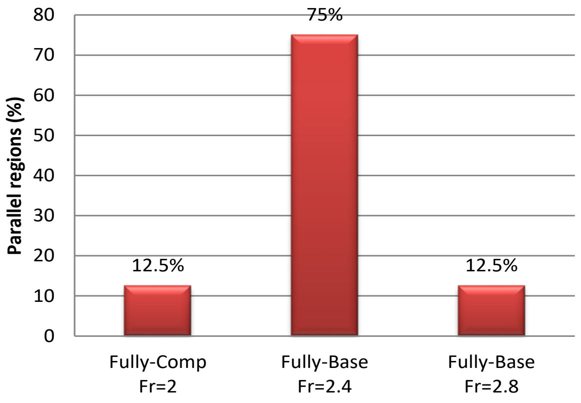


Figure (6.9) The distribution of the optimal configurations for EDP for 4Base/2Composed architecture

in predicting the energy-efficiency. This tree-based classifier generates a decision list for regression problems using separate-and-conquer process which results in highest EDP accuracy.

Next are Perceptron, LinearReg and LeastSqMed predictors, respectively. We implemented a multi-layer perceptron neural network with three layers which is capable of numerical predictions, since neurons are isolated and region approximations can be adjusted independently to each other. Finally, REPTree classifier shows the lowest accuracy as compared to other learning models. REPTree is another fast decision tree learning model, which builds a decision tree using information gain and variance. This model only sorts values for numeric attributes once and missing values are dealt with by splitting the corresponding instances into pieces which negatively impacts the accuracy of predictor in our EDP prediction problem as compared to other models.

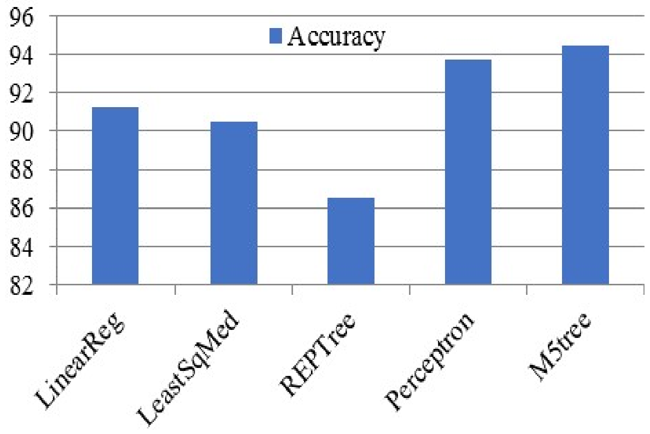


Figure (6.10) Comparison of EDP prediction accuracy of ML classifiers

Hardware Implementation. In this section, we discuss the hardware implementation of the machine learning classifiers. We use Vivado HLS compiler to develop the HDL implementation of the classifiers and estimate the area, latency and power overhead. When it comes to choosing machine learning classifiers for hardware implementation, accuracy of any algorithm is not the only parameter for decision-making. Area, power and latency overhead of ML classifiers are also key factors in selecting a cost-efficient machine learning classifier. While complex algorithms such as Neural Networks can deliver high accuracy, they will also add significant overhead in terms of hardware implementation. Also given their complexity, they might be slow in finding the right configuration for scheduling. We are interested in analyzing these overheads when implementing these machine-learning algorithms. A ML algorithm with high

Table (6.7) Hardware implementation reports of various ML classifiers

ML Algorithm	Latency (cycles@10ns)	Power (W)	Area (LUTs+FFs+DSPs)
LinearReg	36	0.253	3071
LeastSqMed	46	0.267	3127
MultiLayerPercep.	116	0.52	20955
M5Tree	51	0.287	11120
REPTree	9	0.241	2532

accuracy, low area, low power consumption, and low latency is the ideal choice for EDP prediction to guide the scheduling.

The latency, power and area results for implemented machine learning algorithms are shown in Table 6.7. As can be seen, the MultiLayerPerceptron algorithm results in significant area and latency overhead compare to other learning methods. REPtree decision tree is the fastest algorithm compared to others but comes with the lowest accuracy. M5Tree, another decision tree learning predictor, is the most accurate predictor however it comes with significant area overhead. Clearly the results show some trade-off between accuracy, latency, and area overhead.

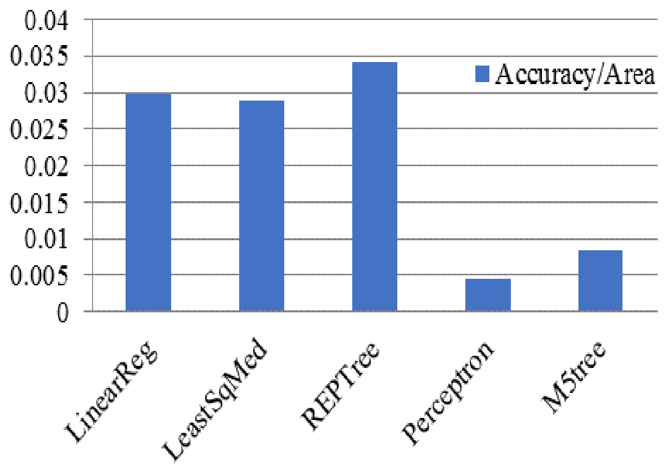


Figure (6.11) Accuracy/Area ratio comparison between ML predictors

Therefore, it is important to compare classifiers by taking these parameters into account. The metric of EDP accuracy over area is a fair ratio to compare the studied predictors. This metric essentially indicates which learning algorithm is the most accurate per unit of silicon area. We have shown results of Accuracy/Area in Figure 6.11. As can be seen in this figure, REPTree, LinearReg and LeastSqMed classifiers are performing significantly better in terms of accuracy per area compared to highly accurate but complex MultiLayerPerceptron and M5Tree.

In addition, if delay is a constraint, REPTree, LinearReg and LeastSqMed classifiers outperform the more complex MultiLayerPerceptron and M5Tree in terms of latency. This is mainly because REPTree model doesn't involve in complex floating-point operations unlike others and instead, it involves in various conditional evaluations. This helps REPTree to achieve lower power consumption as well. However, this is not the case for every tree-based classifier. M5tree which is also a tree-based classifier with higher power consumption has floating point operations. Out of all classifiers, MultiLayerPerceptron performs worst in terms of power consumption and latency mostly because of complex sigmoid function calculations. Comparing based on Accuracy/Area ratio, the results show that REPTree is outperforming all other learning algorithms.

Concluding Remarks. Scheduling multithreaded applications on composite cores architectures is a challenging problem, given various optimization parameters. In this paper, we respond to this challenge by developing a scheduling and tuning solution. The space for tuning configuration parameters in a composite core architecture is large, and our analysis indicates that there is no unique solution for the most energy-efficient configuration for different multithreaded applications, calling for developing a model to predict energy-efficiency for various tuning parameters. In response, we present a systematic approach for energy-efficiency prediction using various machine learning algorithms.

We develop five machine learning-based models for estimating energy-efficiency of multithreaded applications in composite cores architecture. Our proposed ML-based approach, takes hardware performance counters information at run-time from a multithreaded application, and it predicts the most energy-efficient configurations based on run-time analysis and sets the number of threads and operating frequency. It also decides whether to compose little cores into big cores. The results show significant EDP prediction accuracy of as high as 94% across studied applications. We also compared these algorithms in terms of accuracy, latency, power and area overhead. Our results show that although using non-linear regression or neural network models such as M5Tree or MultiLayerPerceptron provides more accurate EDP prediction compared to simple regression or decision tree-based models, they significantly increase complexity of the design in terms of power and area overhead.

6.2 Power Conversion Efficiency-Aware Mapping of Multithreaded Applications on HMPs

Heterogeneous multicore processors offer significant advantages over homogeneous designs in terms of both performance and power by executing workloads on the most appropriate core type. By running multithreaded applications on a heterogeneous architecture, each thread is able to run on a core that matches required resources more closely than a one-size-fits-all solution [37, 60]. Commercially available heterogeneous architectures include Intel Quick IA [124] and ARMs big.LITTLE [125]that integrates a high performance big core with a low power little core on a single chip. Although heterogeneous architectures take advantage of variation in the application characteristics at run-time to improve energy-efficiency, they create unique challenges in the effective mapping of threads to cores. As the core configurations in HMPs become more diverse, the task of effective programming becomes more difficult. In other words, the effectiveness of heterogeneous architectures significantly depends on the scheduling policy and how efficiently the application is assigned to the most appropriate processing core [37, 56, 62, 70, 101].

As mentioned previously, prior studies have mainly examined the advantages of using single threaded applications in HMPs. However, running multithreaded applications on HMPs and choosing the ideal processor architecture to optimize energy-efficiency is a more challenging problem, that must consider the possible number of cores and threads, type of core micro-architecture, and the potential to combine multiple core types. In addition, prior work have ignored power conversion efficiency as a critical optimization parameter. In fact, unlike a homogeneous architecture, in an HMP, the maximum load on cores varies significantly depending on the core type. For instance, in an Exynos 5, the maximum power of big A15 is five times more than little A7. Therefore, there is a difference in power conversion efficiency on big and little cores for the same application that is critical for scheduling. For instance, assume the same application executed on a big core and little core dissipates 1W and 0.9 W of power, respectively. Now consider a PCE of 90% and 70% for the big and little cores, respectively.

The execution of the application now requires 1.1 W and 1.4W of power supplied to the big and little cores, respectively, which implies a change in the most efficient core type after accounting for PCE. Since power conversion efficiency is dependent on the load (core type), it is shown in this paper that it is critical to account for PCE when making scheduling decisions. The experimental results demonstrate that PCE directly affects the choice of the right core type (big vs. little) optimize energy-efficiency. In this work, an energy-efficient scheduling approach is proposed that accounts for the interplay between various application and micro-architectural tuning parameters with respect to the impact of on-chip power delivery on the energy-efficiency of the HMP executing multithreaded programs. To the best knowledge of the authors, there has been no prior effort to concurrently fine-tune the core type, operating voltage/frequency, and application thread counts that also considers the impact of the PCE of the voltage regulators on the optimization of the energy-efficiency in an HMP.

Previous studies on mapping applications to multicore architectures have focused primarily on 1) homogeneous architectures, and 2) configuring individual or a subgroup of tuning parameters at a time, such as application thread counts,voltage/frequency, core type, and have ignored the interplay among all parameters. In addition, the recent studies in [1,36] attempt to examine the interplay among tuning parameters for an HMP but have ignored the impact of PCE of voltage regulators.

This study indicates that tuning parameters individually, while important, do not produce an optimized configuration that achieves the best energy-efficiency on an HMP. The best configuration for a multithreaded application is effectively found, only when tuning parameters are jointly optimized. Exploring the impact of on-chip voltage regulator PCE on the energy-efficiency of an HMP running multithreaded applications is an additional main contribution of this work. The key contributions of this work are summarized as follows [38]:

- The interplay of tuning parameters on performance, power, and energy-efficiency is evaluated for an HMP. The specific parameters at the micro-architecture, system and application levels that are critical to performance as well as power and energy-efficiency and are studied in this work are core type, voltage/frequency settings and the running thread counts.

- The impact of power conversion efficiency of on-chip voltage regulators on the selection of tuning parameters is investigated for maximizing the energy-efficiency. Specifically, four different settings for PCE of VRs are implemented to examine the effect of PCE on the energy-efficiency of multithreaded applications running on the HMP. The results indicate that the energy-efficiency of the multithreaded applications running on an HMP significantly depends on the power conversion efficiency of the on-chip voltage regulators. In other words, depending on the PCE of the on-chip VRs, instead of migrating to small cores, it can be more energy-efficient to keep the application on a big core.

- A system level optimization technique is developed that is aware of the PCE of the on-chip VRs. Based on conducted workload characterization and analysis of the PCE, a machine learning-based model is proposed for predicting the energy-efficiency of various configurations of the application, system, and architecture level parameters to guide scheduling of multithreaded applications.

6.2.1 Overview of PCE-Aware Scheduling

The primary objective of this paper is to analyze the interplay of various tuning parameters, (core type, voltage/frequency, number of threads) for running multithreaded applications on heterogeneous architectures and to highlight the importance of accounting for the PCE of on-chip voltage regulators for each core type to assist in scheduling decisions. An overview of the three-stage PCE-aware approach for predicting the right core type and application configuration is depicted in Figure 6.12. The machine learning-based approach begins from extracting micro-architectural data (referred as feature extraction) and the power and performance (execution time) characteristics, to characterize the multithreaded workload, prepare the dataset for PCE analysis, and train the prediction model. The extracted features include the hardware performance counter data, which represent the application behavior at run-time.

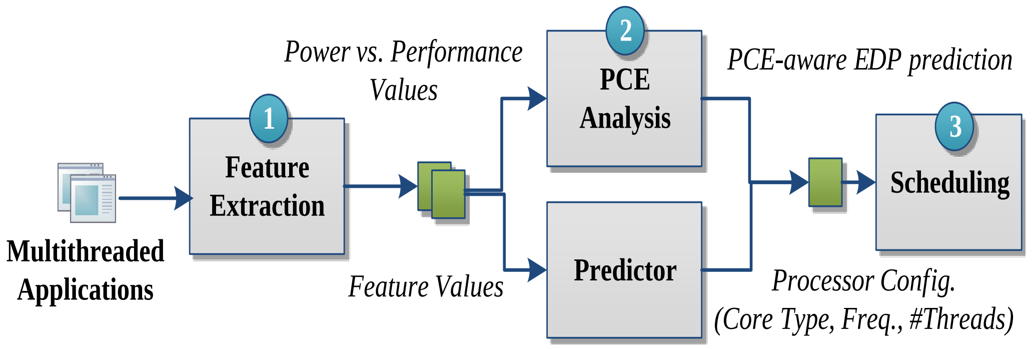

Power vs. Performance Values
2
PCE Analysis
PCE-aware EDP prediction
1
Feature Extraction
3
Scheduling
Multithreaded Applications
Feature Values
Predictor
Processor Config. (Core Type, Freq., #Threads)

Figure (6.12) An overview of the PCE-aware learning-based approach

Since PCE is dependent on voltage regulator design, the architecture of the big and little cores, and the maximum load gap between the two, in this work, no specific assumption is made regarding the PCE of the big and little cores. Instead, all possible scenarios representing various differences between the PCE of the big and little cores are explored. Next, a comprehensive PCE analysis is performed by implementing various PCE models for both big and little cores and evaluating the impact of power conversion efficiency on the energy-efficiency of multithreaded applications. Furthermore, a machine learning-based predictor (that is built off-line) accordingly takes in feature data as well as the PCE of the regulator and predicts the best system configuration for a given application. Finally, the processors are configured and the application is scheduled to run on the predicted configuration.

Table (6.8) Architectural specification

Microarch. Parameter	Little Core	Big Core
Dispatch Width	2	4
Window Size	32	128
Levels of Cache	2	3
L1 I-Cache/Acc. Time	32KB, 8-way/4-cyc	32KB, 4-way/2-cyc
L1 D-Cache/Acc. Time	24KB, 6-way/4-cyc	32KB, 4-way/2cyc
L2-Cache/Acc. Time	1024KB/16-way/12- cyc	256KB/8-way/8cyc
L2-Shared Cores	2	1
L3 Cache	-	8MB/16-way

6.2.2 Experimental Setup and Methodology

In this section details of the experimental setup are provided. Sniper [127] version 6.1, a parallel, high speed and cycle-accurate x86 simulator for multicore systems is used for simulation. McPAT is integrated with Sniper and is used to obtain power consumption of the cores. The SPLASH-2 [129] and PARSEC [130] multithreaded benchmark suites are examined through simulation. For architectural simulation, a big.LITTLE heterogeneous architecture is modeled. For the little core architecture, a core similar to the Atom Silvermont is modeled and the big core is configured with resources similar to the Xeon Gainestown. The Uncore event set of Silvermont and the Intelligent Performance Counter of Gainestown are used to collect data for characterization and drive the scheduling algorithm. The Energy Delay Product (EDP) is used to characterize energy-efficiency, that aims to balance performance and power consumption.

The micro-architectural configuration of the little and big core of the described HMP is listed in Table 6.8. The examined HMP consists of 8 little and 4 big cores. It is important to note that for benchmark simulation, the binding (one-thread-per-core) model is applied with threads = cores to maximize the performance of multithreaded applications [71].

Table (6.9) PCE scenarios for little and big core VRs

VRs PCE Models (Little Core vs. Big Core)	PCE_Little	PCE_Big
Full Efficiency	100%	100%
Low gap	60%	80%
Medium gap	40%	80%
Large gap	20%	80%

6.2.3 Power Conversion Efficenicy Analysis

In this section, the motivation to include the voltage regulator efficiency as one of the tuning parameters influencing the energy-efficiency of the HMP is described. For analyzing the efficiency of on-chip voltage regulators, per-core voltage regulation is considered as shown in Figure 6.5. In the model, each core has a dedicated OCVR, which is a flexible state of the art VR configuration that enables the system to set the voltage and frequency for each core individually to address core-to-core process variation [131, 132]. In addition, since the power management unit directly controls the OCVRs, turning them on or off, a power gating circuit is not needed. The impact of power conversion efficiency of the on-chip VRs on energy-efficiency of multithreaded applications is demonstrated by implementing the PCE scenarios listed in Table 6.9.

The first case listed in Table 6.9 represents Full Efficient VRs. This is the ideal scenario in which the power conversion efficiency of the little and big cores are assumed to be 100%. The full efficiency case is used as a baseline for comparing the other PCE models and evaluating the impact of OCVR efficiency on the EDP. For this purpose, three different PCE sets are assigned to each OCVR with low, medium and large gap between the little and big core. The values are chosen more accurately than the baseline model represent the PCE of on-chip VRs and to effectively determine the impact of PCE variation on the energy-efficiency of HMP.

An example depicting the EDP of the barnes application while considering different on-chip VR models with varying PCE is shown in Figure 6.14. The VR models are based on the 2-phase and 4-phase dc-dc buck converter models in [131]. In order to effectively present the impact of voltage regulator PCE on the EDP in each case, in this section, the EDP results for one of the studied frequencies (2.8 GHz) is chosen to examine the gap between the energy-efficiency of the two cores for different per-core PCE values.

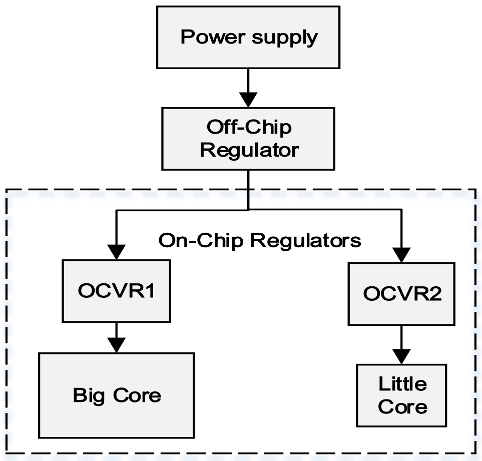


Figure (6.13) The power-supply configurations for the experimental HMP

Note that changing the number of threads interestingly affects the impact of the PCE on the energy-efficiency. As is seen from Figure 6.14-(a) and 6.14-(b), when the number of running threads is low (less than 3), the PCE significantly impacts the choice of selecting the more energy-efficient core as compared to the higher number of threads. The results shown in Figure 6.14-(a) clearly indicate there is a large gap between the energy-efficiency of little and big core when running the application with lower number of threads. However, that is not the case with higher thread counts. As the number of threads increases, the difference between the EDP of the little and big core reduces which makes the big core more competitive with the little core in

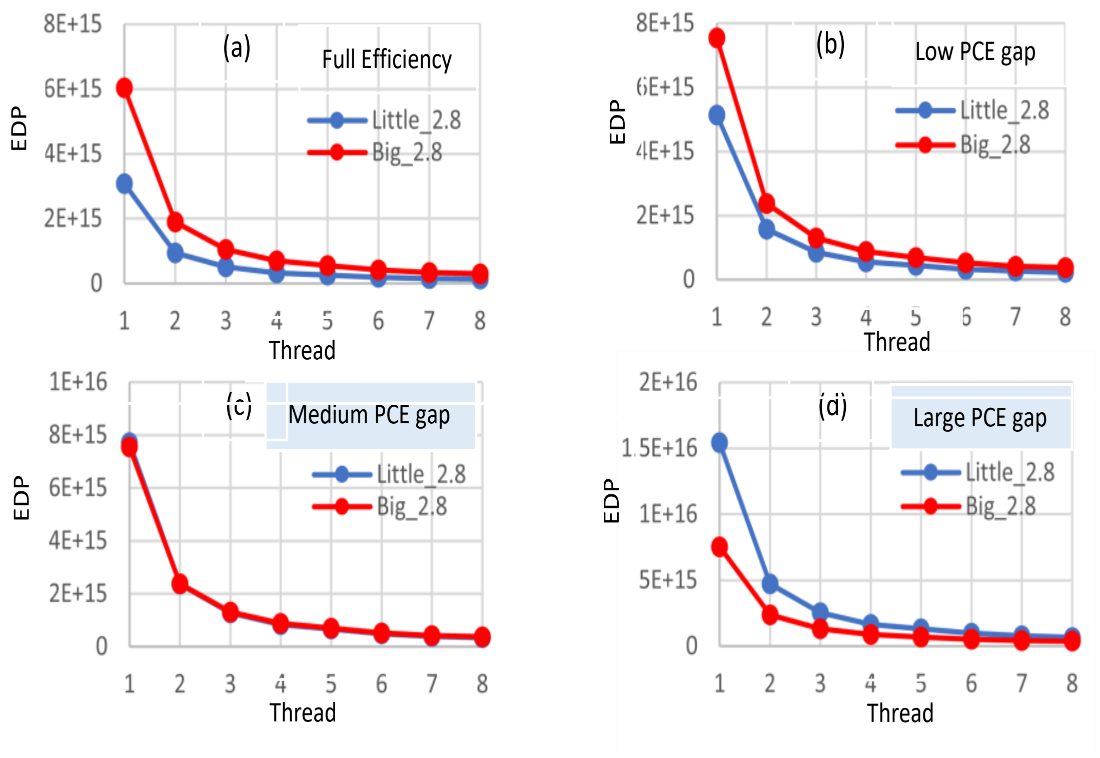


Figure (6.14) Energy-efficiency (in terms of EDP) of barnes for four different PCE gaps between little and big cores: a) full efficiency, b) low, c) medium, d) large

terms of energy-efficiency.

As shown in Figure 6.14-(b), it is assumed that there is a low gap between the PCE of the little and big cores. The assumed Little-PCE is 60% and the Big-PCE is 80%. As can be seen, even though the gap between the EDP of the little and big cores is relatively smaller than the baseline case, where the PCE for both core types is 100% (Figure 6.14-a), as the number of threads is changed accordingly, the little core still outperforms the big core in terms of EDP delivering better energy-efficiency. Therefore, when the gap between the PCE of the little and big cores is relatively small, similar to the case when the PCE for both core types is 100%, it is more energy-efficient to migrate from the big core to the little core and run the application to achieve a lower EDP.

The EDP results for the scenario in which the PCE gap of the little and big core is increased to 40% (medium PCE gap) is depicted in Figure 6.14-c. As is seen, the EDP for the little core increases and overlaps with the EDP of the big core, indicating that the two cores are almost as energy-efficient for PCE gap of 40%. In the last scenario, the PCE gap between the little and big core VRs is further increased. As shown, for large PCE gap, the big core outperforms the little core in terms of EDP. Therefore, as the PCE gap increases, the selection of the little core is no longer optimal in terms of EDP for running multithreaded applications.

The results of PCE analysis indicates that the most energy-efficient choice varies depending on the PCE gap between the big and little cores and the best choice changes compared to the case when the PCE is ignored. Also, as shown in Figure 6.14, the number of threads along with the PCE gap determines the most efficient core. It is, therefore, important to explore the impact of the power conversion efficiency of the on-chip voltage regulators in an HMP to determine the optimal core configuration that achieves the optimized EDP.

6.2.4 Proposed Scheduling Framework

Joint analysis of (Core Type, Freqeuncy, Thread Counts) with respect to various PCE models. To understand the interplay among various tuning parameters and determine the optimum configuration for maximizing the energy-efficiency, all permutations of the parameters described in section III were analyzed. Four voltage/frequency settings were applied on two core types, while executing 11 applications from the SPLASH2 and PARSEC benchmark suits with thread counts of up to 4 and 8.

The interplay among the tuning parameters were comprehensively investigated with respect to each selected voltage regulator PCE model. Due to space limitations, the optimal configurations that yield the optimal EDP for two corner cases are reported which are the full efficiency and large PCE gap models listed in Table 6.10. The relative EDP variation is also calculated for each benchmark, which indicates the relative difference between energy-efficiency for the best configuration of parameters in the little and big cores.

The variation parameter indicates whether to run the application on the little core or big core. For this purpose, a variation threshold is defined that decides what type of core architecture is best suited for executing the corresponding multithreaded application more energy-efficiently. The user-defined threshold is adjusted based on the architecture and available resources as well as the cost of migration. Note that migrating applications from the little core to the big core or, vice versa, comes with power as well as delay overhead.

In this work, a conservative implementation with a delay overhead of 10K cycles is assumed, which is much longer than the overhead to flush the pipeline and copy the content of private cache [56, 70]. Moreover, a 20% variation threshold is assumed to select the more energy-efficient core to run the multithreaded application. As a result, if the percentage of variation between the best-little and best-big architectures is found to be less than 20%, we use the little core for scheduling instead of the big core to avoid migration overhead.

An important observation from the optimal configurations highlighted in Table 6.10 is that as the gap in PCE increases, the optimal configurations corresponding to the best EDP show to occur more on the big core. The percentage of applications executed on each of the two core types for the four PCE scenarios is shown in Figure 6.15. As shown, for the full efficiency model, the little core has a higher probability of being the optimal configuration. However, as the PCE gap gradually increases, Table (6.10) Optimal configurations with optimization target EDP for full efficiency PCE and large gap PCE model

Benchmark	Full Efficiency					Large PCE Gap				
	Best-Little		Best-Big		Var. (%)	Best-Little		Best-Big		Var. (%)
	Freq. (GHz)	*#Thread*	*Freq. (GHz)*	*#Thread*		*Freq. (GHz)*	*#Thread*	*Freq. (GHz)*	*#Thread*	
barnes	2.4	8	2.8	4	-444.8	2.4	8	2.8	4	-36.2
fmm	2.4	8	2.4	4	2.2	2.4	8	2.4	4	75.5
cholesky	2.4	8	2	4	28	2.4	8	2	4	77
radix	2.8	8	2.8	4	-138.7	2.8	8	2.8	4	40.3
radiosity	2.4	8	1.6	4	-102.8	2.4	8	1.6	4	49.3
raytrace	2.4	5	2	4	-28.9	2.4	5	2	4	67.7
fft	2	4	2	2	36.3	2	4	2	2	79.1
lu.cont	2.4	8	2.8	4	27.2	2.4	8	2.8	4	98.7
blackscholes	2.8	4	2.4	4	83.85	2.8	4	2.4	4	195.3
bodytrack	2	3	2	3	41.23	2	3	2	3	112.6
ferret	2	6	2	6	2.4	2	6	2	6	132.6

the energy-efficient core is shifted from the little to the big core. As a result, for the large PCE gap model, as compared to the full efficient scheme, the possibility of the big core being the more energy-efficient than the little core increases by 45%. The reason is due to the significant increase in the EDP of the little core as compared to the big core when the PCE gap between the two cores increases. As shown in Figure 6.15, close to 55% of the optimal configurations indicate that the little core is more energy-efficient.

The number of optimal configurations reduces to less than 10% as the PCE gap increases to 60%. Therefore, considering the PCE of the OCVRs in scheduling decisions of multithreaded applications on HMPs is critical. In order to perform a comprehensive EDP characterization of the studied architectures, all possible configurations (core types and number of threads) are categorized into four classes. The first two are Fully-Little and Fully-Big configurations that refer to cases in which the lowest EDP is achieved with full utilization of either the little or big core, respectively. In other words, the optimum number of threads is equal to the maximum number of existing little/big cores. On the other hand, Partially-Little and Partially-Big configurations are utilized when the best number of threads is lower than the maximum available cores.

The diversity of optimum configurations across various applications and on-chip voltage regulator PCE scenarios demonstrates that when running a given multithreaded workload on an HMP, depending on the application and the PCE of the OCVRs, different core configuration parameters (core type, voltage/frequency, number of threads) lead to the best energy-efficiency. The simulation results indicate that the optimal configuration varies across the applications. The dispersed pattern of optimum results implies that there is a necessity of developing a prediction method to guide scheduling decisions of unknown multithreaded applications in order to improve the EDP of an HMP with respect to a given PCE of the OCVRs.

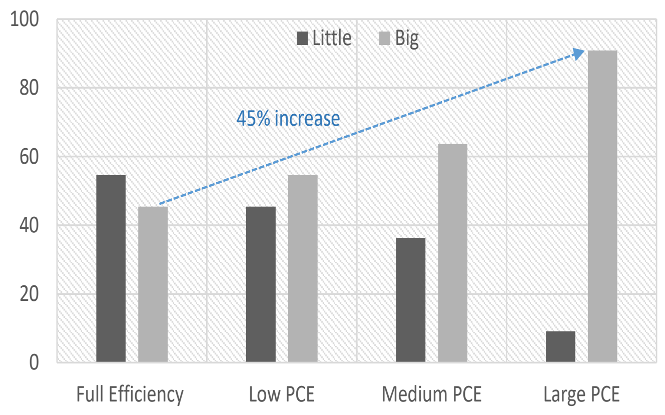


Figure (6.15) Optimal core type selection for different power conversion efficiencies

6.2.5 Prediction Model for Energy-efficiency

Model selection: Recent studies have proposed ordinary least squares regression (OLSR) modeling to estimate the power and performance of a processor at run-time. The results of this work indicate that OLSR is not the best suited algorithm for performance and power estimation as outliers, particularly for heterogeneous architectures, mislead the model. In fact, various applications experience different phases with different behavior. In addition, superscalar processors are complex, which makes it difficult to develop a general model for estimation of power/performance. OLSR models are highly sensitive to the outliers and potentially produce misleading results as even a single point of data substantially impacts the regression efficiency.

As a result, in this research, based on a comprehensive characterization of various applications, a more robust regression algorithm is evaluated in addition to OLSR, referred as the Quantile Linear Regression (QLR) model, to predict the energy-efficiency for various configurations of the studied HMP. The primary advantage of QLR as compared to OLSR is the robustness against outliers. The QLR model is useful to more comprehensively analyze the relationship between variables and provides a richer classification of the data, allowing for a characterization of the impact of a covariate on the entire distribution of target variables. For the QLR model, a specific quantile of data is set instead of the mean value. The quantile is set to 0.1, which results in minimizing the median of the error values.

Although the use of non-linear regression or neural network models potentially provides a more accurate estimation of the energy-efficiency of an application, the complexity of the design is increased, with a corresponding increase in hardware complexity. The overhead in area, power and performance of implementing a linear regression model in hardware is minimal and shown to be easily integrated into a core [56]. The QLR model achieves higher accuracy as compared to OLSR. A comparison between the derived coefficients of the two different predictors using ordinary linear regression and quantile linear regression is shown in Figure 6.16. In the figure, the black dotted line is the slope coefficient for the QLR and the red lines are the least squares estimate for OLSR and the corresponding confidence interval. The lower and upper quantiles are well beyond the least squares estimate. The effects of L2 cache access and branch miss prediction vary over quantiles, and the magnitude of the effects

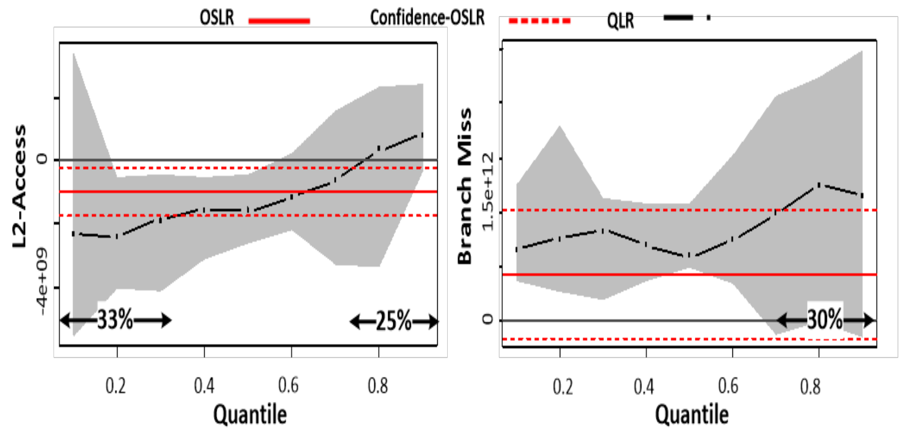


Figure (6.16) Quantile graphs for predictors: a) L2-Access, b) Branch miss prediction

at various quantiles differs considerably from the OLSR coefficient, even in terms of the confidence intervals around each coefficient (58% for the L2-access and 30% for the branch miss predictor). Therefore, an ordinary least squares regression is not an optimal solution to capture the actual behavior of applications when predicting the energy-efficiency.

QLR derivation and training: To derive the prediction model for energy-efficiency, the development of the training data set for the prediction model is required. Training involves finding the best processor and application configuration and extracting feature values for each training workload, reducing the extracted features to the most vital performance counters, and developing a learning model from the training data. Note that the input variables in the developed classifiers are extracted performance counter information from different training applications as well as the PCE value for each on-chip voltage regulator, while the output variable is the EDP for a given set of tuning parameters. Therefore, a subset (less than two third) of applications from the SPLASH2 and PARSEC multithreaded benchmark suites is considered. The studied

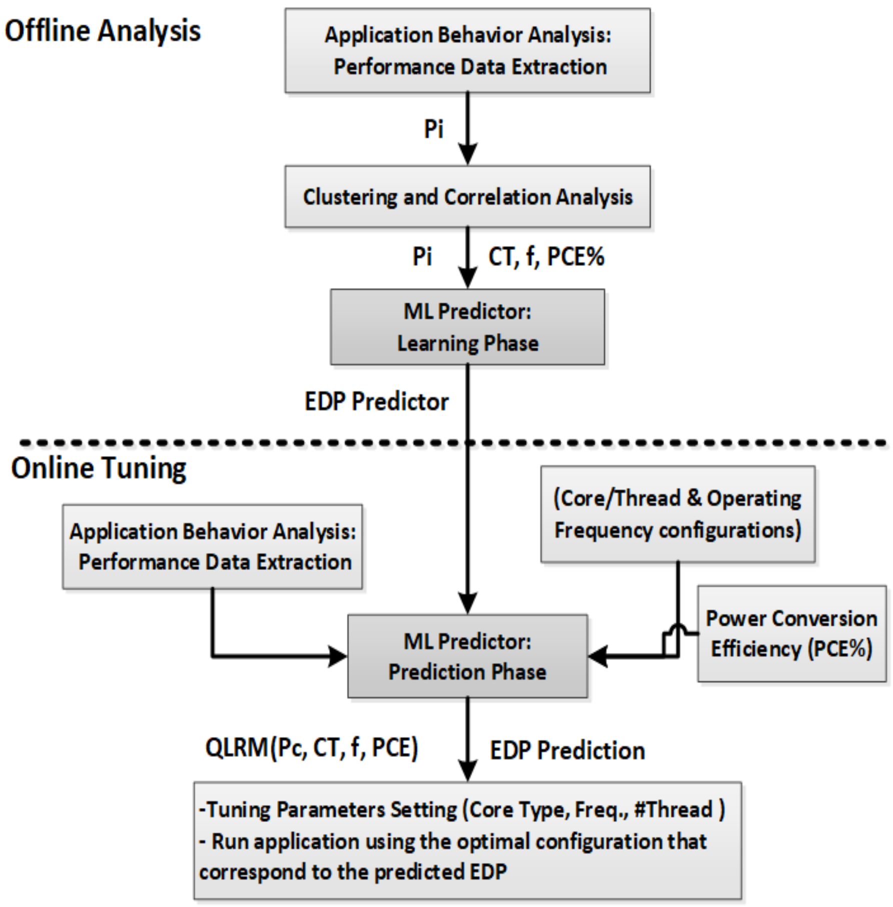

Offline Analysis
Application Behavior Analysis: Performance Data Extraction
Pi
Clustering and Correlation Analysis
Pi
CT, f, PCE%
ML Predictor: Learning Phase
EDP Predictor
Online Tuning
Application Behavior Analysis: Performance Data Extraction
(Core/Thread & Operating Frequency configurations)
Power Conversion Efficiency (PCE%)
ML Predictor: Prediction Phase
QLRM(Pc, CT, f, PCE)
EDP Prediction
-Tuning Parameters Setting (Core Type, Freq., #Thread)
- Run application using the optimal configuration that correspond to the predicted EDP

Figure (6.17) Proposed PCE-aware scheduling scheme with energy-efficiency prediction

applications represent diverse compute, memory and I/O intensity behavior. For each benchmark, twelve pieces of hardware performance counter data are collected on all possible configurations of core types, voltage/frequency operating points, and thread counts.

Given the twelve hardware performance counters, Principle Component Analysis (PCA) and correlation analysis are used on the training set to monitor the critical micro-architecture parameters and capture application characteristics. By applying the attribute reduction method, the four most related performance counters are determined which include the L1 D-cache access, L2 cache-access, L2 cache-miss and branch miss prediction. Since the primary purpose of the prediction model is to predict energy-efficiency across various application, system and micro-architectural parameters, the primary tuning parameters in must be considered in the model as well. Therefore, along with the identified key performance counter parameters, three tuning parameters (core type, frequency, threads) are included as input variables to the model to enable predicting the EDP for each configuration that results when changing the core type, operating frequency, and/or thread counts. After identifying the four key hardware performance parameters and considering the tuning parameters, the proposed PCE-aware energy-efficiency prediction model is formulated using quantile linear regression as follows:

$$QLRM - PCE = \beta 0 + (\sum_{i=1}^{4} \beta i + Pi) + \beta 5 \times CT + \beta 6 \times f \tag{6.1}$$

where $\beta 0$ is the intercept, βi denotes the corresponding coefficients of the regression model, Pi are extracted hardware performance counters, and QLRM-PCE is the estimated energy-efficiency (in terms of EDP) given the PCE of the on-chip voltage regulators. In addition, the core/thread configurations are given by CT, and f represents the frequency on the corresponding core architecture. The βi coefficients can be interpreted as the expected change in EDP per unit change in L1 D-cache access, L2 cache-access, L2 cache-miss, branch misprediction, core/thread and frequency setting. The model predicts continuous values representing the energy delay product as a function of performance counter inputs and tuning parameters, which are then used to make the scheduling decisions at run-time. During run-time, given an unknown application, the QLR model predict the EDP of all possible configurations based on a single set of executing data. The configuration corresponding to the lowest estimated EDP is then selected for the run.

Energy-Efficient PCE-aware Scheduling Algorithm: An overview of the PCE-aware scheduling scheme using the regression-based prediction model is provided in Figure 6.17. As illustrated, the scheduling algorithm is split between an offline step and an online step. In offline analysis, the prediction model is trained using quantile linear regression, as described in section V-B. For the online tuning step, a multi-threaded application is run with the most aggressive configuration settings, where all tuning parameters are set to maximum (maximum number of threads, highest frequency, and the big core). Next, date from the hardware performance counters is extracted by profiling the multithreaded application, as it is running with the maximum configuration settings. The profiling stage is used for run-time characterization and resource utilization of the applications.

The regression classifier takes the key performance counter parameters and configuration settings as inputs, and outputs the system energy-efficiency for the given configuration and given PCE. Note that the linear weights are estimated using the training data set. Given the input configuration parameters during run-time, the QLRM-PCE predicts the optimal energy-efficiency. The output resulting in the optimal energy-efficiency and the corresponding new configuration is then chosen as the current operating point at run-time. The predictive model, by observing the run-time behavior of a multithreaded application running with a specific configuration, predicts the right configuration of parameters that includes the number of threads, operating voltage and frequency, and core type (big or little) to achieve the maximum energy-efficiency for a given PCE. It is important to note that the QLRM-PCE can be trained for other objectives such as ED2P optimization.

Table (6.11) Average relative error

	Core/Thread Configurations			
Freq.	Full-Little	Partial-Little	Full-Big	Partial-Big
2.8 GHz	10.5%	10.74%	11.69%	2.03%
2.4 GHz	22.49%	21.4%	4.67%	4.87%
2.0 GHz	1.9%	3.9%	1.74%	3.1%
1.6 GHz	3.35%	2.2%	3.6%	2.61%

6.2.6 Evaluation Results

In order to evaluate the accuracy of the prediction model, the value of the relative mean absolute error (RMAE) is calculated which is defined as $(|estimated - actual|)/((actual))100\%$. The RMAE metric indicates the relative difference between the predicted and observed maximum energy-efficiency. To validate the QLRM-PCE model, we applied percentage split method to divide the dataset into two sets, using 60% (known applications) of the data to train the model and 40% (unknown applications) to simulate and evaluate.

The average relative errors of the QLRM-PCE are listed in Table 6.11. As shown, all possible configurations of the 16 operating points consisting of various frequencies and core/thread configurations are characterized. As shown, the proposed prediction classifier is most accurate in estimating the energy-efficiency of the Fully-Big and Fully-Little architectures, both operating at 2 GHz. In addition, the developed learning model achieves an average error of 6.85% across all training data samples and possible configurations.

The proposed classifier assists the scheduling decisions of multithreaded applications on an HMP that include choosing the core type, setting the operating voltage and frequency, and adapting the number of running threads. The performance overhead of implementing the QLRM-PCE in hardware and calculating values at each

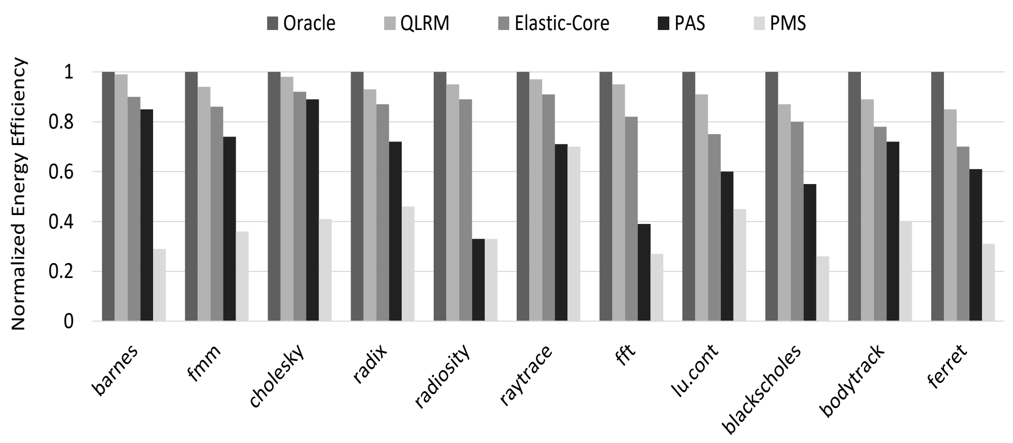


Figure (6.18) Normalized energy-efficiency of applications on various scheduling schemes relative to Oracle scheduling

interval is negligible. The power overhead of implementing the QLRM-PCE is 5uW, which is further reduced by gating idle units during each interval. In order to evaluate the efficiency of the prediction model, the following scheduling schemes are studied for comparison:

- Oracle: This model is based on the heterogeneous architecture with an ideal energy-efficiency predictor, where all future behavior of the application as well as the power and performance for various configurations are known in advance. Since the Oracle scheme provides the upper bound for energy-efficiency, it is used to normalize and compare the other schemes.

- QLRM-PCE: This scheme is based on the proposed PCE-aware quantile linear regression model to estimate the EDP for various core sizes, frequency/voltage points, and number of threads for a given voltage regulator PCE.

- Elastic-Core [56]: This dynamic scheme proposed recently uses a linear regression model to predict the power and performance of single-threaded applications as a function of core type and frequency settings. However, unlike the model in this paper, the impact of PCE is not accounted for. In addition, although Elastic-Core does not account for the number of threads, the thread counts in this paper are set to maximum values to better evaluate the model by fairly comparing it against a recent dynamic scheduling solution.

- Performance Aggressive Scheduling (PAS): In this scheme, all tuning parameters are set to maximum values to achieve the maximum performance. Therefore, each application is executed on big cores operating at 2.8 GHz and with 4 threads.

- Power Minimized Scheduling (PMS): This scheduling scheme attempts to minimize power consumption. The application is running on a little core and the frequency and number of threads are set to minimum values.

The energy-efficiency of the studied applications normalized to the Oracle model with a fully efficient VR is shown in Figure 6.18. The QLRM-PCE on average achieves close to 95% efficiency as compared to the Oracle model. The QLRM-PCE has improved energy-efficiency as compared to the Elastic-Core and PAS schemes by an average of 10% and 30% across all benchmarks, respectively. The average energy-efficiency of different scheduling schemes across various studied PCE gap models is shown in Figure 6.19. By increasing the PCE gap, the energy-efficiency of the Elastic-Core, PAS, and PMS scheduling schemes diminishes. For the largest PCE gap, the proposed QLRM-PCE outperforms a state of the art solution, the Elastic-Core, in energy-efficiency by 60%. The results verify the efficacy of the proposed prediction model and the effectiveness of the proposed scheduling scheme to harness the power of an HMP for enhanced energy-efficiency.

Concluding Remarks. Emerging heterogeneous multicore architectures are complex processors with various tuning optimization knobs for improving performance and energy-efficiency. Scheduling multithreaded applications in these architectures is a challenging problem given the various optimization parameters at the application

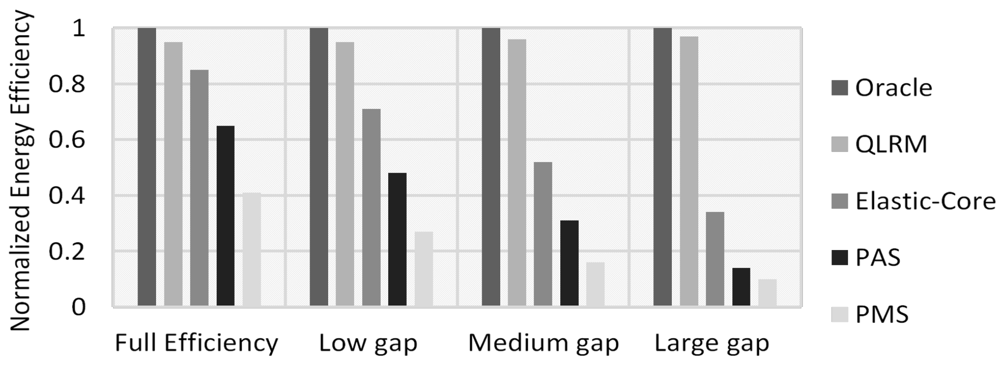


Figure (6.19) Average energy-efficiency results of different scheduling schemes with respect to various PCE models

(number of running threads), system (operating voltage and frequency), and architecture (core type- big vs. little) levels. In addition, unlike homogeneous architectures, the efficiency of on-chip voltage regulators and the power conversion efficiency gap between the big and little cores in these architectures are critical parameters that must be accounted for.

The interplay among the tuning parameters and the influence each has on the energy-efficiency, make the scheduling and tuning of the application even more challenging. In this paper, a PCE-aware scheduling and tuning solution is developed that highlights the importance of accounting for the PCE of big and little core to find the appropriate core type that optimizes the EDP. A predictive model is developed for estimating the energy-efficiency of multithreaded applications. Based on the predictive model, a scheduling scheme is developed for effective mapping of multithreaded applications to an HMP by setting the tuning parameters to maximize the energy-efficiency. The results indicate that the proposed scheduling scheme achieves on average close to 95% efficiency as compared to the Oracle scheduler.

Chapter 7: Conclusion

Hardware performance counter registers are built-in hardware units available in modern microprocessors that are designed to count low-level micro-architectural events during the applications' execution. Since HPC registers are already provided on-chip in modern CPU architectures, they are assumed as an existing resource that can be carefully deployed to improve the applications performance, power, and security without incurring any additional hardware overhead to the computer system. While cyber-attacks such as malicious software are increasing in number and sophistication, the extensive majority of these attacks are still not able to perform much malicious activity without leaving basic hardware signatures, such as missed branch predictions, cache misses, etc. As a result, by considering the influential role of the run-time underlying hardware events, this research have mainly investigated the applicability of effective machine learning techniques for designing secure and energy-efficient computer architectures using hardware performance counter-based profiles.

To realize a highly accurate run-time malware detection for the purpose of security enhancement, this work identified various challenges associated with prior efforts on hardware cybersecurity in the context of run-time hardware-assisted malware detection. For a large database of malware and benign applications, we found that the detection rate and performance of HMD highly depend on the number of available HPCs, the type of microarchitectural events to monitor, the type of ML algorithm to classify, and whether the malware spawns as a new thread or embeds in a running benign application. In response to these challenges, we proposed complexity-effective machine learning-based solutions to realize run-time and specialized HMD by taking advantage of the low-level features of microprocessor i.e., HPC events to capture the application behavior.

Furthermore, to highlight the importance of deploying low-level hardware features for power and performance optimization, in this thesis we have discussed the suitability of proposing effective machine learning-based solutions for enhancing the energy-efficiency of heterogeneous multicore architectures by making use of applications' run-time hardware-based information. In particular, we showed that how machine learning and hardware performance counter information can be effectively deployed for energy-efficiency prediction and scheduling of multithreaded applications running on multicore heterogeneous computer systems. In overall, the outcome of this research opens a path for computer designers and architects in making appropriate and efficient architectural decisions for implementing future computer systems and processors, to most effectively improve the performance of machine learning algorithms for different optimization goals such as security and energy-efficiency of computer systems for different emerging applications.

Appendix A: List of Publications

[**DATE**] **H. Sayadi**, H. Makrani, S. Manoj P D, T. Mohsenin, A. Sasan, S. Rafatirad, and H. Homayoun, 2SMaRT: A Two-Stage Machine Learning-Based Approach for Run-Time Hardware-Assisted Malware Detection, in Proceedings of Design, Automation Test in Europe (DATE'19), Florence, Italy, 2019.

[**DAC**]. S. Manoj P D, S. Amberkar, S. Bhat, **H. Sayadi**, S. Rafatirad, and H. Homayoun, Adversarial Attack on Microarchitectural Based Malware Detectors, to appear in 56th ACM/IEEE Design Automation Conference (DAC'19), Las Vegas, Nevada, 2019.

[**DAC**] **H. Sayadi**, Y. Gao, S. Rafatirad, J. Lin, and H. Homayoun, CHASE: A Customized Time Series Machine Learning Approach for Hardware-Based Stealthy Malware Detection, to appear in (Work-in-Progress Sessions) 56th ACM/IEEE Design Automation Conference (DAC'19), Las Vegas, Nevada, 2019.

[**DATE**] S. Manoj P D, **H. Sayadi**, H. M. Makrani, C. Nowzari, S. Rafatirad, and H. Homayoun, Lightweight Node-level Malware Detection and Network-level Malware Confinement in IoT Networks, in Proceedings of Design, Automation Test in Europe (DATE'19), Florence, Italy, 2019.

[**FPL**] H. Makrani, F. Farahmand, **H. Sayadi**, S. Rafatirad, and Houman Homayoun Pyramid: Machine Learning Framework to Estimate the Optimal Timing and Resource Usage of a High-Level Synthesis Design in International Conference on Field-Programmable Logic and Applications (FPL), 2019.

[**GOMACTech**] M. Taram, D. M. Tullsen, A. Venkat, **H. Sayadi**, H. Wang, S Manoj P D, and H. Homayoun, Fast and Efficient Deployment of Security Defenses via Context Sensitive Decoding, in Proceedings of the 44th Government Microcircuit Applications and Critical Technology Conference (GOMACTech'19), March 2019.

[**ASP-DAC**] H. Makrani, **H. Sayadi**, T. Mohsenin, S. Rafatirad, A. Sasan, and H. Homayoun, XPPE: Cross-Platform Performance Estimation of Hardware Accelerators Using Machine Learning, in Proceedings of 24th IEEE/ACM Asia South Pacific Design Automation Conference (ASP-DAC'19), Tokyo, Japan, January 2019.

[**DAC**] **H. Sayadi**, N. Patel, S. Manoj P D, A. Sasan, S. Rafatirad, and H. Homayoun, Ensemble Learning for Effective Run-Time Hardware-Based Malware Detection: A Comprehensive Analysis and Classification, in Proceedings of 55th ACM/IEEE Design Automation Conference (DAC'18), San Francisco, California, June 2018.

[**ASP-DAC**] **H. Sayadi**, D. Pathak, I. Savidis, and H. Homayoun, Power Conversion Efficiency-Aware Mapping of Multithreaded Applications on Heterogeneous Architectures: A Comprehensive Parameter Tuning, in Proceedings of IEEE/ACM 23rd Asia and South Pacific Design Automation Conference (ASP-DAC'18), South Korea, January 22-25, 2018.

[**CASES**] F. Brasser, L. Davi, A. Dhavlle, T. Frassetto, S. Manoj P D, S. Rafatirad, A. Sadeghi, A. Sasan, **H. Sayadi**, S. Zeitouni, and H. Homayoun, Special Session: Advances and Throwbacks in Hardware-Assisted Security, in Proceedings of IEEE International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES'18), Italy, October 2018.

[**TrustCom**] **H. Sayadi**, H. Makrani, S. Manoj P D, S. Rafatirad, and H. Homayoun, Customized Machine Learning-Based Hardware-Assisted Malware Detection in Embedded Devices, in Proceedings of IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'18), New York, August 2018.

[**CF**] **H. Sayadi**, S. Manoj P D, A. Houmansadr, S. Rafatirad, and H. Homayoun, Comprehensive Assessment of Hardware-Supported Malware Detection Using General and Ensemble Learning, in Proceedings of ACM International Conference on Computing Frontiers (CF'18), Italy, May 2018.

[**SoCC**] H. M. Makrani, **H. Sayadi**, D. Motwani, H. Wang, S. Rafatirad, and H. Homayoun, Energy-aware and Machine Learning-based Resource Provisioning of In-Memory Analytics on Cloud, in Proceedings of the ACM Symposium on Cloud Computing (SoCC'18), California, 2018.

[**CHEST**] **H. Sayadi**, H. Homayoun, Adversary Resilient and Complexity-Aware Runtime Malware Detection, in Workshop in Hardware and Embedded Systems Security and Trust (CHEST'18), Fairfax, Virginia, August 2018. (Poster Presentation)

[**MEMSYS**] H. Makrani, **H. Sayadi**, S. Rafatirad, and H. Homayoun, A Comprehensive Memory Analysis of Data Intensive Workloads on Server Class Architecture, in ACM International Symposium on Memory Systems (MEMSYS'18), Washington DC, 2018.

[**ASAP**] H. Makrani, **H. Sayadi**, S. Manoj P D, and H. Homayoun, Compressive Sensing on Storage Data: An Effective Solution to Alleviate I/O Bottleneck in Data-Intensive Workloads, in Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'18), Italy, 2018.

[**ICCD**] **H. Sayadi**, N. Patel, A. Sasan, and H. Homayoun, Machine Learning-Based Approaches for Energy-Efficiency Prediction and Scheduling in Composite Cores Architectures, in Proceedings of IEEE International Conference on Computer

Design (ICCD'17), Boston, MA, November 5-8, 2017. - **Runner-up for Best Paper Award** - Selected as top ranked paper for publishing at IEEE Transactions on Emerging Topics in Computing

[**IGSC**] **H. Sayadi**, H. Homayoun, Scheduling of Multithreaded Applications onto Heterogeneous Composite Cores Architectures, in Proceedings of IEEE International Green and Sustainable Computing Conference (IGSC'17), Orlando, Florida, October 23-25, 2017.

[**DAC**] **H. Sayadi**, H. Homayoun, Characterization and Scheduling of Multithreaded Applications on Composite Cores Architectures in (Work-in-Progress Sessions) ACM/IEEE Design Automation Conference (DAC'17), Austin, Texas, June 2017.

[**IBM-ET**] **H. Sayadi**, H. Homayoun, Comprehensive Analysis of Hardware-Based Malware Detectors, in IBM/IEEE CAS EDS Emerging Technology Symposium (IBM-ET'17), Thomas J. Watson Research Centre, Yorktown Heights, NY, October 2017. (Poster Presentation)

[**DFT**] **H. Sayadi**, H. Farbeh, A. M. Monazzah, and S. Gh. Miremadi, A Data Recomputation Approach for Reliability Improvement of Scratchpad Memory in Embedded Systems, in Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology System (DFT'14), pp. 228-233, Amsterdam, Netherlands, October 2014.

Bibliography

[1] H. Sayadi, N. Patel, S. M. PD, A. Sasan, S. Rafatirad, and H. Homayoun, "Ensemble learning for effective run-time hardware-based malware detection: A comprehensive analysis and classification," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[2] H. Sayadi, H. M. Makrani, S. M. Pudukotai Dinakarrao, T. Mohsenin, A. Sasan, S. Rafatirad, and H. Homayoun, "2smart: A two-stage machine learning-based approach for run-time specialized hardware-assisted malware detection," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 728–733.

[3] H. Sayadi, H. Farbeh, A. M. H. Monazzah, and S. G. Miremadi, "A data re-computation approach for reliability improvement of scratchpad memory in embedded systems," in *2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 2014, pp. 228–233.

[4] A. Mosenia and N. K. Jha, "A comprehensive study of security of internet-of-things," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 4, pp. 586–602, 2016.

[5] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.

[6] A. M. Nia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Physiological information leakage: A new frontier in health information security," *IEEE Transactions on Emerging Topics in Computing*, vol. 4, no. 3, pp. 321–334, 2015.

[7] A. Mosenia, S. Sur-Kolay, A. Raghunathan, and N. K. Jha, "Caba: Continuous authentication based on bioaura," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 759–772, 2016.

[8] S. M. Pudukotai Dinakarrao, H. Sayadi, H. M. Makrani, C. Nowzari, S. Rafatirad, and H. Homayoun, "Lightweight node-level malware detection and network-level malware confinement in iot networks," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 776–781.

[9] F. Brasser, L. Davi, A. Dhavlle, T. Frassetto, S. M. P. Dinakarrao, S. Rafatirad, A. Sadeghi, A. Sasan, H. Sayadi, S. Zeitouni, and H. Homayoun, "Special session: Advances and throwbacks in hardware-assisted security," in *2018 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Sep. 2018, pp. 1–10.

[10] H. Sayadi, H. M. Makrani, O. Randive, S. M. PD, S. Rafatirad, and H. Homayoun, "Customized machine learning-based hardware-assisted malware detection in embedded devices," in *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2018, pp. 1685–1688.

[11] K. Zamiri Azar, H. Mardani Kamali, H. Homayoun, and A. Sasan, "Threats on logic locking: A decade later," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '19. New York, NY, USA: ACM, 2019, pp. 471–476. [Online]. Available: http://doi.acm.org/10.1145/3299874.3319495

[12] H. M. Kamali, K. Z. Azar, H. Homayoun, and A. Sasan, "Full-lock: Hard distributions of sat instances for obfuscating circuits using fully configurable logic and routing blocks," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 89:1–89:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317831

[13] H. Sayadi, A. R. S. SM PD, Houmansadr, and H. Homayoun, "Comprehensive assessment of run-time hardware-supported malware detection using general and ensemble learning," in *ACM International Conference on Computing Frontiers, CF*, 2018.

[14] K. Neshatpour, M. Malik, A. Sasan, S. Rafatirad, T. Mohsenin, H. Ghasemzadeh, and H. Homayoun, "Energy-efficient acceleration of mapreduce applications using fpgas," *Journal of Parallel and Distributed Computing*, vol. 119, pp. 1–17, 2018.

[15] K. Neshatpour, H. Mohammadi Makrani, A. Sasan, and H. H. Hassan Ghasemzadeh, Setareh Rafatirad, "Design space exploration for acceleration of machine learning applications," *FCCM*, 2018.

[16] K. Neshatpour, H. M. Makrani, A. Sasan, H. Ghasemzadeh, S. Rafatirad, and H. Homayoun, "Architectural considerations for fpga acceleration of machine learning applications in mapreduce," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, 2018.

[17] A. Bettany and M. Halsey, *What Is Malware?* Berkeley, CA: Apress, 2017, pp. 1–8. [Online]. Available: https://doi.org/10.1007/978-1-4842-2607-0_1

[18] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in Computer Virology*, vol. 4, no. 3, pp. 251–266, Aug 2008.

[19] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar, "A survey on malware detection using data mining techniques," *ACM Computing Surveys*, vol. 50, no. 3, pp. 1–40, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3101309.3073559

[20] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. ACM, 2013, pp. 559–570.

[21] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2015, pp. 651–661.

[22] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. New York, NY, USA: ACM, 2017, pp. 483–493. [Online]. Available: http://doi.acm.org/10.1145/3052973.3052999

[23] X. Wang and R. Karri, "Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters," in *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, May 2013, pp. 1–7.

[24] M. Kazdagli, L. Huang, V. Reddi, and M. Tiwari, "Emma: New platform to evaluate hardware-based mobile malware analyses," *CoRR*, vol. abs/1603.03086, 2016. [Online]. Available: http://arxiv.org/abs/1603.03086

[25] A. hardware performance counters a cost effective way for integrity checking of programs, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the Sixth ACM Workshop on Scalable Trusted Computing*, ser. STC '11. ACM, 2011, pp. 71–76.

[26] N. C. Doyle, E. Matthews, G. Holland, A. Fedorova, and L. Shannon, "Performance impacts and limitations of hardware memory access trace collection," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, March 2017, pp. 506–511.

[27] S. M. P. Dinakarrao, S. Amberkar, S. Bhat, A. Dhavlle, H. Sayadi, A. Sasan, H. Homayoun, and S. Rafatirad, "Adversarial attack on microarchitectural events based malware detectors," in *Proceedings of*

the 56th Annual Design Automation Conference 2019, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 164:1–164:6. [Online]. Available: http://doi.acm.org/10.1145/3316781.3317762

[28] M. Labs, "Infographic: Mcafee labs threats report," December 2018.

[29] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. ACM, 2017, pp. 25:1–25:6.

[30] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, ser. RAID 2015. Springer-Verlag New York, Inc., 2015, pp. 3–25.

[31] M. B. Bahador, M. Abadi, and A. Tajoddin, "Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition," in *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, Oct 2014, pp. 703–708.

[32] X. Wang, S. Chai, M. Isnardi, S. Lim, and R. Karri, "Hardware performance counter-based malware identification and detection with adaptive compressive sensing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 3:1–3:23, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2857055

[33] S. J. Stolfo, K. Wang, and W.-J. Li, "Towards stealthy malware detection," in *Malware Detection*, M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, Eds. Boston, MA: Springer US, 2007, pp. 231–249.

[34] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult, "A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys Tutorials*, vol. 19, no. 2, pp. 1145–1172, Secondquarter 2017.

[35] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *ACM Asia Conference on Computer and Communications security (ASIACCS'14)*, 2014.

[36] H. Sayadi and H. Homayoun, "Scheduling multithreaded applications onto heterogeneous composite cores architecture," in *Green and Sustainable Computing Conference (IGSC), 2017 Eighth International*. IEEE, 2017, pp. 1–8.

[37] H. Sayadi, N. Patel, A. Sasan, and H. Homayoun, "Machine learning-based approaches for energy-efficiency prediction and scheduling in composite cores architectures," in *2017 IEEE International Conference on Computer Design (ICCD)*, Nov 2017, pp. 129–136.

[38] H. Sayadi, D. Pathak, I. Savidis, and H. Homayoun, "Power conversion efficiency-aware mapping of multithreaded applications on heterogeneous architectures: A comprehensive parameter tuning," in *Design Automation Conference (ASP-DAC), 2018 23rd Asia and South Pacific*. IEEE, 2018, pp. 70–75.

[39] M. Malik, D. M. Tullsen, and H. Homayoun, "Co-locating and concurrent fine-tuning mapreduce applications on microservers for energy efficiency," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2017, pp. 22–31.

[40] H. Sayadi, "Energy-efficiency prediction of multithreaded workloads on heterogeneous composite cores architectures using machine learning techniques," *arXiv preprint arXiv:1808.01728*, 2018.

[41] H. M. Makrani, H. Sayadi, T. Mohsenin, A. Sasan, H. Homayoun *et al.*, "Xppe: cross-platform performance estimation of hardware accelerators using machine learning," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 727–732.

[42] H. M. Makrani and H. Homayoun, "Memory requirements of hadoop, spark, and mpi based big data applications on commodity server class architectures," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 112–113.

[43] H. M. Makrani, S. Tabatabaei, S. Rafatirad, and H. Homayoun, "Understanding the role of memory subsystem on performance and energy-efficiency of hadoop applications," in *Green and Sustainable Computing Conference (IGSC), 2017 Eighth International*. IEEE, 2017, pp. 1–6.

[44] H. M. Makrani and H. Homayoun, "Mena: A memory navigator for modern hardware in a scale-out environment," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, 2017, pp. 2–11.

[45] H. M. Makrani, H. Sayadi, S. M. Pudukotai, S. Rafatirad, and H. Homayoun, "A comprehensive memory analysis of data intensive workloads on server class architecture," in *ACM International Symposium on Memory Systems (MEMSYS)*, 2018.

[46] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2015, pp. 544–551.

[47] M. Malik, K. Neshatpour, T. Mohsenin, A. Sasan, and H. Homayoun, "Big vs little core for energy-efficient hadoop computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1480–1485.

[48] K. Neshatpour, M. Malik, A. Sasan, S. Rafatirad, and H. Homayoun, "Hardware accelerated mappers for hadoop mapreduce streaming," *IEEE Transactions on Multi-Scale Computing Systems*, 2018.

[49] H. M. Makrani, H. Sayadi, S. Manoj, S. Raftirad, and H. Homayoun, "Compressive sensing on storage data: An effective solution to alleviate i/0 bottleneck in data-intensive workloads," in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–8.

[50] H. M. Makrani, H. Sayadi, D. Motwani, H. Wang, S. Rafatirad, and H. Homayoun, "Energy-aware and machine learning-based resource provisioning of in-memory analytics on cloud," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2018, pp. 517–517.

[51] M. Malik, S. Rafatirad, and H. Homayoun, "System and architecture level characterization of big data applications on big and little core server architectures," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, vol. 3, no. 3, p. 14, 2018.

[52] M. Kazdagli, V. J. Reddi, and M. Tiwari, "Quantifying and improving the efficiency of hardware-based mobile malware detectors," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 37:1–37:13. [Online]. Available: http://dl.acm.org/citation.cfm?id=3195638.3195683

[53] A. Garcia-Serrano, "Anomaly detection for malware identification using hardware performance counters," *CoRR*, vol. abs/1508.07482, 2015. [Online]. Available: http://arxiv.org/abs/1508.07482

[54] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *International Workshop on Recent Advances in Intrusion Detection*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer, 2014, pp. 109–129.

[55] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, Nov 2016.

[56] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun, “Elasticcore: Enabling dynamic heterogeneity with joint core and voltage/frequency scaling,” in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[57] M. G. Moghaddam and C. Ababei, “Dynamic energy management for chip multi-processors under performance constraints,” *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 54, pp. 1–13, 2017. [Online]. Available: https://doi.org/10.1016/j.micpro.2017.08.005

[58] ——, “Dynamic energy management for chip multi-processors under performance constraints,” *Microprocessors and Microsystems - Embedded Hardware Design*, vol. 54, pp. 1–13, 2017. [Online]. Available: https://doi.org/10.1016/j.micpro.2017.08.005

[59] G. Liu, J. Park, and D. Marculescu, “Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems,” in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 54–61.

[60] C. Kim, S. Sethumadhavan, M. S. Govindan, N. Ranganathan, D. Gulati, D. Burger, and S. W. Keckler, “Composable lightweight processors,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, Dec 2007, pp. 381–394.

[61] J. Cong and B. Yuan, “Energy-efficient scheduling on heterogeneous multi-core architectures,” in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED ’12. New York, NY, USA: ACM, 2012, pp. 345–350. [Online]. Available: http://doi.acm.org/10.1145/2333660.2333737

[62] H. Homayoun, V. Kontorinis, A. Shayan, T. Lin, and D. M. Tullsen, “Dynamically heterogeneous cores through 3d resource pooling,” in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012, pp. 1–12.

[63] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez, “Core fusion: Accommodating software diversity in chip multiprocessors,” in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 186–197. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250686

[64] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke, “Composite cores: Pushing heterogeneity into a core,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2012, pp. 317–328.

[65] V. Kontorinis, M. K. Tavana, M. H. Hajkazemi, D. M. Tullsen, and H. Homayoun, "Enabling dynamic heterogeneity through core-on-core stacking," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[66] G. Liu, J. Park, and D. Marculescu, "Procrustes 1: Power constrained performance improvement using extended maximize-then-swap algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1664–1676, 2015.

[67] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack cap: Adaptive dvfs and thread packing under power caps," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2011, pp. 175–185.

[68] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *Proceedings of the 3rd Conference on Computing Frontiers*, ser. CF '06. New York, NY, USA: ACM, 2006, pp. 29–40. [Online]. Available: http://doi.acm.org/10.1145/1128022.1128029

[69] J. F. Martinez and E. Ipek, "Dynamic multicore resource management: A machine learning approach," *IEEE Micro*, vol. 29, no. 5, pp. 8–17, Sep. 2009.

[70] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (pie)," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 213–224.

[71] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 116–125.

[72] D. Spinellis, "Reliable identification of bounded-length viruses is np-complete," *IEEE Transactions on Information Theory*, vol. 49, no. 1, pp. 280–284, Jan 2003.

[73] E. Aghaei and G. Serpen, "Ensemble classifier for misuse detection using n-gram feature vectors through operating system call traces," *Journal of Hybrid Intelligent Systems*, 2017.

[74] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996.

[75] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: An update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, Nov 2009.

[76] M. Helsely, "Lxc: Linux container tools," in *IBM developer works technical library*, 2009.

[77] Intel, "Intel 64 and ia-32 architectures software developer manual, volume 3b: System programming guide," 2016.

[78] G. Serpen and E. Aghaei, "Host-based misuse intrusion detection using pca feature extraction and knn classification algorithms," *Intelligent Data Analysis*, vol. 22, no. 5, pp. 1101–1114, 2018.

[79] C. O. S. Sorzano, J. Vargas, and A. P. Montano, "A survey of dimensionality reduction techniques," *ArXiv e-prints*, Mar. 2014.

[80] G. Yan, N. Brown, and D. Kong, "Exploring discriminatory features for automated malware classification," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'13)*, K. Rieck, P. Stewin, and J.-P. Seifert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 41–61.

[81] K. Neshatpour, M. Malik, and H. Homayoun, "Accelerating machine learning kernel in hadoop using fpgas," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2015, pp. 1151–1154.

[82] H. M. Kamali, K. Z. Azar, and S. Hessabi, "Ducnoc: A high-throughput fpga-based noc simulator using dual-clock lightweight router micro-architecture," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 208–221, 2017.

[83] S. Rezaei, K. Kim, and E. Bozorgzadeh, "Scalable multi-queue data transfer scheme for fpga-based multi-accelerators," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE, 2018, pp. 374–380.

[84] S. Rezaei, C. Hernandez-Calderon, S. Mirzamohammadi, E. Bozorgzadeh, A. Veidenbaum, A. Nicolau, and M. J. Prather, "Data-rate-aware fpga-based acceleration framework for streaming applications," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–6.

[85] H. Mardani Kamali and A. Sasan, "Much-swift: A high-throughput multi-core hw/sw co-design k-means clustering architecture," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI '18. ACM, 2018, pp. 459–462. [Online]. Available: http://doi.acm.org/10.1145/3194554.3194648

[86] H. M. Kamali, K. Z. Azar, K. Gaj, H. Homayoun, and A. Sasan, "Lut-lock: A novel lut-based logic obfuscation for fpga-bitstream and asic-hardware protection," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2018, pp. 405–410.

[87] B. Sprunt, "The basics of performance-monitoring hardware," *IEEE Micro*, vol. 22, no. 4, pp. 64–71, Jul 2002.

[88] K. Shen and et al., "Hardware counter driven on-the-fly request signatures," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. ACM, 2008, pp. 189–200.

[89] M. Kayaalp and et al., "Scrap: Architecture for signature-based protection from code reuse attacks," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2013, pp. 258–269.

[90] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for android," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 15–26. [Online]. Available: http://doi.acm.org/10.1145/2046614.2046619

[91] A. A. Elhadi, M. Aizaini Maarof, and A. Hamza Osman, "Malware detection based on hybrid signature behaviour application programming interface call graph," *American Journal of Applied Sciences*, vol. 9, no. 3, p. 283, 2012.

[92] G. Gu, P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee, "Bothunter: Detecting malware infection through ids-driven dialog correlation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 12:1–12:16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362915

[93] R. Sekar and et al., "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 144–.

[94] M. Christodorescu and et al., "Mining specifications of malicious behavior," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 5–14.

[95] A. Moser and et al., "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, Dec 2007, pp. 421–430.

[96] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray, "A semantics-based approach to malware detection," in *Proceedings of the 34th Annual ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190270

[97] A. S. Gazafroudi, M. Shafie-khah, E. Heydarian-Forushani, A. Hajizadeh, A. Heidari, J. M. Corchado, and J. P. Catalão, "Two-stage stochastic model for the price-based domestic energy management problem," *International Journal of Electrical Power & Energy Systems*, vol. 112, pp. 404–416, 2019.

[98] A. S. Gazafroudi, F. Prieto-Castrillo, T. Pinto, and J. M. Corchado, "Energy flexibility management in power distribution systems: Decentralized approach," in *2018 International Conference on Smart Energy Systems and Technologies (SEST)*, Sep. 2018, pp. 1–6.

[99] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[100] L. Yu and H. Liu, "Feature selection for high-dimensional data: A fast correlation-based filter solution," in *Proceedings of the 20th international conference on machine learning (ICML-03)*, 2003, pp. 856–863.

[101] H. Liu and H. Motoda, *Feature selection for knowledge discovery and data mining.* Springer Science & Business Media, 2012, vol. 454.

[102] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[103] V. intelligence service, "Virustotal intelligence service," in *http://www.virustotal.com/intelligence/*, April 2018.

[104] J. Roberts, "virusshare.com," in *http://www.virusshare.com/*, April 2018.

[105] R. Duda, P. Hart, and D. Stork, *Pattern Classification.* John Wiley & Sons, 2001.

[106] S. Wold and et al., "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, no. 1, pp. 37 – 52, 1987, proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0169743987800849

[107] Y. Freund and R. ESchapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, May 2002.

[108] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. D. Keromytis, "A study of malcode-bearing documents," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment(DIMVA'07)*. Berlin, Heidelberg: Springer, 2007, pp. 231–250.

[109] M. Z. Shafiq, S. A. Khayam, and M. Farooq, "Embedded malware detection using markov n-grams," pp. 88–107, 2008.

[110] L. Ye and E. Keogh, "Time series shapelets: a new primitive for data mining," in *KDD'09*. ACM, 2009, pp. 947–956.

[111] J. Lin and Y. Li, "Finding structural similarity in time series data using bag-of-patterns representation," in *SSDBM'18*. Springer, 2009, pp. 461–477.

[112] T. Rakthanmanon and E. Keogh, "Fast shapelets: A scalable algorithm for discovering time series shapelets," in *proceedings of the 2013 SIAM International Conference on Data Mining*. SIAM, 2013, pp. 668–676.

[113] P. Schäfer, "Scalable time series classification," *Data Mining and Knowledge Discovery*, vol. 30, no. 5, pp. 1273–1298, 2016.

[114] X. Li and J. Lin, "Linear time complexity time series classification with bag-of-pattern-features," in *ICDM'17*, 2017, pp. 277–286.

[115] F. Karim, S. Majumdar, H. Darabi, and S. Chen, "Lstm fully convolutional networks for time series classification," *IEEE Access*, vol. 6, pp. 1662–1669, 2017.

[116] Y. Zheng, Q. Liu, E. Chen, Y. Ge, and J. L. Zhao, "Time series classification using multi-channels deep convolutional neural networks," in *International Conference on Web-Age Information Management*. Springer, 2014, pp. 298–310.

[117] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," *Data Mining and Knowledge Discovery*, vol. 33, no. 4, pp. 917–963, Jul 2019. [Online]. Available: https://doi.org/10.1007/s10618-019-00619-1

[118] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *2017 International joint conference on neural networks (IJCNN)*. IEEE, 2017, pp. 1578–1585.

[119] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[120] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[121] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[122] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[123] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster, "Dynamically tuning processor resources with adaptive processing," *Computer*, vol. 36, no. 12, pp. 49–58, Dec 2003.

[124] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer, "Quickia: Exploring heterogeneous architectures on real prototypes," in *IEEE International Symposium on High-Performance Comp Architecture*, Feb 2012, pp. 1–8.

[125] S. Kamdar and N. Kamdar, "big. little architecture: Heterogeneous multicore processing," *International Journal of Computer Applications*, vol. 119, no. 1, 2015.

[126] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, June 2004, pp. 64–75.

[127] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 28:1–28:25, Aug. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629677

[128] 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 *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[129] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Proceedings 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.

[130] 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*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454128

[131] D. Pathak, M. H. Hajkazemi, M. K. Tavana, H. Homayoun, and I. Savidis, "Energy efficient on-chip power delivery with run-time voltage regulator clustering," in *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2016, pp. 1210–1213.

[132] H. Asghari-Moghaddam, H. R. Ghasemi, A. A. Sinkar, I. Paul, and N. S. Kim, "Vr-scale: Runtime dynamic phase scaling of processor voltage regulators for improving power efficiency," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

Biography

Hossein Sayadi has been a Ph.D. student in Department of Electrical & Computer Engineering at George Mason University, Fairfax, VA from 2015 to 2019. He was a member of the Accelerated, Secure, and Energy-Efficient Computing Lab (ASEEC), working under the supervision of Dr. Houman Homayoun. Hossein's research interests mainly lie in Computer Architecture, Hardware & Architecture Security, Malware Detection, Applied Machine Learning, Embedded Systems and IoT, and Energy-Efficient Computing. He received his Master Degree in Computer Engineering (Computer Architecture) in 2014 from Sharif University of Technology, Tehran, Iran, where he was a member of Dependable Systems Laboratory (DSL). Furthermore, he obtained his Bachelor Degree in Computer Engineering (Computer Hardware) from K. N. Toosi University of Technology, Tehran, Iran in 2012.