University of Massachusetts Amherst

ScholarWorks@UMass Amherst

Doctoral Dissertations | Dissertations and Theses

October 2018

RUN-TIME PROGRAM PHASE DETECTION AND PREDICTION

Meng-Chieh Chiu

Follow this and additional works at: https://scholarworks.umass.edu/dissertations_2

Part of the Programming Languages and Compilers Commons

Recommended Citation

Chiu, Meng-Chieh, "RUN-TIME PROGRAM PHASE DETECTION AND PREDICTION" (2018). *Doctoral Dissertations*. 1332.
https://scholarworks.umass.edu/dissertations_2/1332

This Open Access Dissertation is brought to you for free and open access by the Dissertations and Theses at ScholarWorks@UMass Amherst. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of ScholarWorks@UMass Amherst. For more information, please contact scholarworks@library.umass.edu.

RUN-TIME PROGRAM PHASE DETECTION AND PREDICTION

A Dissertation Presented

by

MENG-CHIEH CHIU

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2018

College of Information and Computer Sciences

© Copyright by Meng-Chieh Chiu 2018

All Rights Reserved

RUN-TIME PROGRAM PHASE DETECTION AND PREDICTION

A Dissertation Presented

by

MENG-CHIEH CHIU

Approved as to style and content by:

____

J. Eliot B. Moss, Chair

____

Benjamin M. Marlin, Member

____

Charles C. Weems, Member

____

Daniel Holcomb, Member

____

James Allan, Chair of the Faculty
College of Information and Computer Sciences

DEDICATION

To those who are interested in program phases, program languages and system behavior.

ACKNOWLEDGMENTS

Thanks to my advisor J. Eliot B. Moss, who was thoroughly helpful in leading me to complete the dissertation. He inspired in me a thirst for inventiveness and knowledge, and led me to a better model for computer system research. Thanks to Benjamin M. Marlin who built the foundation of my knowledge in the machine learning. This research was supported in part by National Science Foundation Grant CCF-1320498.

ABSTRACT

RUN-TIME PROGRAM PHASE DETECTION AND PREDICTION

SEPTEMBER 2018

MENG-CHIEH CHIU

B.Sc., NATIONAL CHUNG CHENG UNIVERSITY

M.Sc., NATIONAL TAIWAN UNIVERSITY

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor J. Eliot B. Moss

It is well-known that programs tend to have multiple phases in their execution. Because phases have impact on micro-architectural features such as caches and branch predictors, they are relevant to program performance (Xian et al., 2007; Roh et al., 2009; Gu and Verbrugge, 2008) and energy consumption. They are also relevant to detecting whether a program is executing as expected or is encountering unusual or exceptional conditions, a software engineering and program monitoring concern (Peleg and Mendelson, 2007; Singer and Kirkham, 2008; Pirzadeh et al., 2011; Benomar et al., 2014).

We present methods for real-time phase change detection and phase prediction in Java, C, (etc.,) and Python programs. After applying a training protocol to a program of interest, our methods can detect and predict phase at run time for that program with good precision and recall (compared with a "ground truth" definition of phases) and with small performance impact. Furthermore, for concrete applications, we explore run-time energy-efficient clock frequency adjustment for statically compiled executables.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS . . . . **v**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xi**

LIST OF FIGURES . . . . **xiii**

CHAPTER

1. INTRODUCTION **1**

1.1 Outline 2

2. RELATED WORK **4**

2.1 Program Phase Detection and Prediction 4
2.2 Phase Information for Saving Energy 7

3. EXPERIMENT SETTING AND APPROACH **10**

3.1 Environment 10
3.2 Choosing Benchmarks 10
3.3 Java Event Traces 11

3.3.1 Trace Generation 11
3.3.2 Feature Extraction 11
3.3.3 Benchmarks Used 12

3.4 C Event Traces 13

3.4.1 Trace Generation 13
3.4.2 Feature Extraction 13
3.4.3 Benchmarks Used 13

3.5 Python Event Traces 14

3.5.1 Trace Generation 14
3.5.2 Feature Extraction 14
3.5.3 Benchmarks Used 15

4. REAL-TIME PROGRAM PHASE CHANGE DETECTION 16

4.1 A Machine Learning Model for Phase Transition Detection 17

4.1.1 Feature Selection 17
4.1.2 Evaluation of the GMM Clustering Approach 19
4.1.3 Number of Clusters and Features 22

4.2 Real-Time Program Phase Change Detection in Java 23

4.2.1 Offline Evaluation 23
4.2.2 Implementing the Real-Time Detector for Java 23
4.2.3 Results 26

4.2.3.1 The Run-Time Overhead of Our System 26
4.2.3.2 The Quality of the Real-time Detector 27

4.3 Real-Time Program Phase Change Detection in C 29

4.3.1 Offline Evaluation 29
4.3.2 Implementing the Real-Time Detector for C 29
4.3.3 Results 30

4.3.3.1 The Run-Time Overhead of Our System 30
4.3.3.2 The Quality of the Real-time Detector 30

4.4 Real-Time Program Phase Change Detection in Python 31

4.4.1 Offline Evaluation 31
4.4.2 Implementing the Real-Time Detector for Python 32
4.4.3 Results 32

4.4.3.1 The Run-Time Overhead of Our System 33
4.4.3.2 The Quality of the Real-time Detector 33

4.5 Summary 34

5. REAL-TIME PROGRAM PHASE PREDICTION 36

5.1 A Machine Learning Model for Phase Prediction 37

5.1.1 Feature Selection 37
5.1.2 Evaluation of the Decision Tree Approach 38

5.1.2.1 Ground Truth 38
5.1.2.2 Comparison Methodology 40
5.1.2.3 Cross Validation of Decision Trees 41
5.1.2.4 Number of Features and Tree Depths 41
5.1.2.5 Comparison between Online Prediction and Offline Ground Truth 42

5.2 Real-Time Program Phase Prediction in Java 42

5.2.1 Offline Evaluation 42

5.2.1.1 Number of Inputs 42
5.2.1.2 Number of Features and Tree Depths 43
5.2.1.3 Comparison of the Predictors 44

5.2.2 Implementing the Real-Time Predictor for Java 44
5.2.3 Results 45

5.2.3.1 The run-time overhead of our system 45
5.2.3.2 Quality of the Run-time Predictor 46

5.3 Real-Time Program Phase Prediction in C 46

5.3.1 Offline Evaluation 46

5.3.1.1 Number of Inputs 46
5.3.1.2 Number of Features and Tree Depths 46
5.3.1.3 Comparison of the Predictors 49

5.3.2 Implementing the Real-Time Predictor for C 49
5.3.3 Results 49

5.3.3.1 The run-time overhead of our system 50
5.3.3.2 Quality of the Run-time Predictor 50

5.4 Real-Time Program Phase Prediction in Python 51

5.4.1 Offline Evaluation 51

5.4.1.1 Number of Inputs 51
5.4.1.2 Number of Features and Tree Depths 51
5.4.1.3 Comparison of the Predictors 54
5.4.1.4 How Far in the Future Can Our Model Predict Phases? 55

5.4.2 Implementing the Real-Time Predictor for Python 56
5.4.3 Results 57

5.4.3.1 The run-time overhead of our system 57
5.4.3.2 Quality of the Run-time Predictor 60

5.5 Summary 60

6. FROM PROGRAM PHASE TO SYSTEM BEHAVIOR PREDICTION 64

6.1 Introduction 64
6.2 A Machine Learning Model for ED Reduction 65

6.2.1 Estimating ED Product from Cache Misses 65
6.2.2 Feature Selection 67

6.2.2.1 History information 68

6.2.3 Offline Evaluation of the Model 69

6.2.3.1 ED Improvement Using a Fixed CPU Frequency 69
6.2.3.2 ED Improvement using Decision Trees 70
6.2.3.3 Offline Cross Validation of Models 71
6.2.3.4 Number of Features and History 72
6.2.3.5 Comparison of the Models 73

6.3 Implementing the Dynamic Clock Frequency Adjustment Tool 74
6.4 Experiments 75

6.4.1 The Cost of Instrumented Programs 75
6.4.2 Performance of the Real-time Decision Tree Model 76

6.4.2.1 Generalizability of the Real-time DT Model 81

7. CONCLUSION 86

BIBLIOGRAPHY 89

LIST OF TABLES

Table **Page**

4.1 Ratio of running time of programs with 12 features to the original benchmark programs for C. . . . 30

4.2 Ratio of running time of programs with 12 features to the original benchmark programs for Python. . . . 33

5.1 Number of inputs for each benchmarks. . . . 42

5.2 Ratio of running time of programs with 8 features treatment to the original benchmark programs for Java. . . . 46

5.3 Number of inputs for each benchmark. . . . 48

5.4 Ratio of running time of programs with 8 features treatment to the original benchmark programs for C. . . . 51

5.5 Number of inputs for each benchmarks. . . . 53

5.6 Ratio of running time of programs with 8, 12, and 16 features treatment to the original benchmark programs. . . . 58

6.1 Correlation between selected features. “In” means the correlation between features selected from partitions of the same trace, and “Out” means the correlation between features selected from different traces of same program. The 6 and 8 after In/Out is the number of selected features. . . . 72

6.2 Run-time overhead of instrumented programs with 6 and 8 features compared to the uninstrumented versions. . . . 76

6.3 T-test of execution time of programs run with and without instrumentation. . . . 77

6.4 Average ratio of ED product of the real-time adjustment system to the best single fixed clock frequency model. . . . 77

6.5 Average of geometric mean of (ED - optimal)/optimal for nine k and V_{th} settings. 81

LIST OF FIGURES

Figure **Page**

4.1 Performance of GMM, ND, and Random detectors. The offline GMMs shown use 8 selected features and 8 clusters. The threshold for ND is 0.8. 21

4.2 Performance of GMMs with various numbers of clusters 24

4.3 SWF Score for self-test GMM, cross-validated GMM, ND, and Random detectors, varying the number of features used by the GMMs. All GMMs use 8 clusters. The threshold for ND is 0.8. The unprimed x-axis give the number of features for averages that include `jython`, and the primed one for those that do not. 25

4.4 SWF Score for offline cross-validated GMMs with 8, 12, 16, and 24 selected features, including Random and ND for comparison for Java programs. All GMMs use 8 clusters. The threshold for ND is 0.8. 25

4.5 SWF Scores for RT, ND, and Random detectors compared with Baseline for Java programs. The number indicate the number of features for the RT scheme; ST = Self-Test; and CV = Cross-Validation. All RT detectors use 8 clusters. The threshold for ND is 0.8. 28

4.6 SWF Score for offline cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for C programs. The threshold for ND is 0.8. 29

4.7 SWF Score for online cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for C programs. The threshold for ND is 0.8. 31

4.8 SWF Score for offline cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for Python programs. The threshold for ND is 0.8. 32

4.9 SWF Score for online cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for Python programs. The threshold for ND is 0.8. 34

5.1 F-score for predicting the next 1 to 500 future phases for Java programs by offline cross-validated decision trees with 8, 12, and 16 selected features. The maximal depth of the decision trees is 6, 10, or 14. 43

5.2 F-score for predicting the next 1 to 500 future phases for Java programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 6. 45

5.3 F-Score for predicting the next 1 to 500 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for Java programs. We use 8 features and maximal tree depth of 10 in our decision tree model. 47

5.4 F-score for predicting the next 1 to 500 future phases for C programs by offline cross-validated decision trees with 8, 12, and 16 selected features. The maximal depth of the decision trees is 6, 10, or 14. 48

5.5 F-score for predicting the next 1 to 500 future phases for C programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 6. 50

5.6 F-Score for predicting the next 1 to 500 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for C programs. We use 8 features and maximal tree depth of 10 in our decision tree model. 52

5.7 F-score for predicting the next phase for Python programs by offline cross-validated decision trees with 4, 8, 12, 16, 20, 24 and 32 selected features. The maximal depth of the decision trees is 8, 10, or 12. 53

5.8 F-score for predicting the next phase for Python programs by offline cross-validated decision trees with maximal tree depths of 4, 6, 8, 10, 12, 14, and 16. The number of features is 8, 12, or 16. 54

5.9 F-score for predicting the next 1 to 49 future phases for Python programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 10 or 12. 55

5.10 F-Score for predicting the next 1 to 49 phases for Python programs by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim. We use 12 features and maximal tree depth of 10 in our decision tree model. 56

5.11 CDF of length of phases in traces for Python programs. 57

5.12 Execution time distributions of baseline and treatment with various numbers of features for Python programs. 59

5.13 F-score for predicting the next 1 to 49 future phases by run-time cross-validated decision trees with 8, 12, and 16 selected features for Python programs. The maximal tree depth is 10. 61

5.14 F-Score for predicting the next 1 to 49 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for Python programs. We use 12 features and maximal tree depth of 10 in our decision tree model. 62

6.1 Average ED improvement using different numbers of available clock frequencies, excluding traces with less than 0.1% improvement. 71

6.2 Cumulative ED degradation of the DT model over seven traces with different numbers of features and history lengths. The “whiskers” indicate the minimum and maximum ED degradation seen. 73

6.3 Cumulative ED degradation over seven traces of *LRCM*, *LR*, *DT* (with 6 and 8 features), and the model with the best fixed clock frequency. 74

6.4 Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.6$. 78

6.5 Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.8$. 79

6.6 Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 1.0$. 80

6.7 Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.6$. 83

6.8 Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.8$. 84

6.9 Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 1.0$. 85

CHAPTER 1

INTRODUCTION

Programs exhibit different phases of execution. For example, a compiler's phases might be: initialization, parsing, semantic analysis, optimization, and code generation, with additional smaller phases in each. The phases might iterate as the compiler processes each function in the input, etc. But why are phases relevant? Software engineers are interested in phases as part of program analysis and understanding (Peleg and Mendelson, 2007; Singer and Kirkham, 2008; Pirzadeh et al., 2011; Benomar et al., 2014). They are further interested in phases as a program runs in order to determine whether the program is operating as expected. Computer architects, and indeed even ordinary users, are interested in phases because of their impact on micro-architectural components such as caches and branch predictors, and thus on performance (Xian et al., 2007; Roh et al., 2009; Gu and Verbrugge, 2008), in terms of both time used and energy consumed. Another way of putting this is that program phases are an important aspect of dynamic program behavior.

Most current program phase analysis methods are offline. They may provide rich and useful results (Georges et al., 2004; Wang et al., 2012; Nagpurkar and Krintz, 2004; Watanabe et al., 2008), but only after the fact. Some methods have "online" in their names (Nagpurkar et al., 2006; Otte and Richardson, 2007), but that means only that they work in a single forward pass over a stream of information about the program, such as the branch instructions that the program executes. These techniques do not operate in real time. Even if they did, the cost of analyzing each branch as it occurs would likely slow program execution unacceptably.

One practical application of run-time phase detection (undoubtedly there are others as well) is saving energy. Phase information can indicate whether lowering clock frequency or voltage, or both, might save energy (Srinivasan et al., 2013) with little impact on performance. This is true, for example, of phases where the CPU is mostly idle waiting on main memory accesses.

Here, we present methods for run-time phase change detection and phase prediction in Java, C (etc.), and Python programs. After applying a training protocol to a program of interest, our methods can detect and predict phases and phase changes at run time for that program with good precision and recall (compared with a "ground truth" definition of phases) and with small performance impact. Furthermore, for C programs, we explore run-time energy-efficient clock frequency adjustment.

1.1 Outline

Here is the plan of the thesis. In Chapter 2 we discuss how other work is related to our approach. In Chapter 3, we present our experimental environment, data set, and the tracing tools, which are the same for all systems and methodology described in later chapters. In Chapter 4, we introduce the processes of *phase change* detection, including feature selection, prediction model learning, and the methodology of our experiments. We also describe our run-time phase change detection systems for different programming languages and offer results from our real-time phase change detection systems. We then introduce our *phase prediction* systems in Chapter 5. We describe the processes of feature selection and prediction model learning, including our methodology of deriving ground truth and evaluating predictor quality. We also describe our run-time phase prediction systems for different programming languages and offer results from our real-time phase prediction systems. In Chapter 6 we present the processes of feature selection and the two stages of learning for clock frequency adjustment, including our methodology of selecting target traces and of evaluating adjuster quality. Then we describe our real-time clock frequency

adjustment system, and present the setting and methodology of our experiments, and offer results from our real-time clock frequency adjustment system. Finally, in Chapter 7 we describe possible future work and conclude.

CHAPTER 2

RELATED WORK

We describe two areas of related work: program phase detection and prediction, and saving energy.

2.1 Program Phase Detection and Prediction

Dhodapkar and Smith (2003) present an overview of proposed techniques for detecting program phase changes and discuss some of the issues related to the definition of program phases. They also present a comparison of techniques that use unbounded hardware resources and compare practical hardware implementations based on working set signatures and accumulator tables. Bui and Kim (2017) aim to fill a gap in the literature on phase detection by characterizing super fine-grained program phases and exhibiting an application where detection of these relatively short-lived phases can be instrumental. They characterize super fine-grained phases based on different models, and they run models using various interval sizes, similarity thresholds, and models of intervals to evaluate the correlation between these model parameters and the phases detected. Duesterwald et al. (2003) study the time-varying behavior of programs using metrics derived from hardware counters on two different micro-architectures. Dropsho and Mckinley (2002) adjusted the thresholds for transitioning a branch between different predictors in a hybrid predictor, by detecting changes in phase using online statistics. They used hardware counters, and would insert flagged instructions into empty pipeline slots to periodically check for changes in branch behavior by calculating statistical measures on the counts. Georges et al. (2004) propose an offline method-level phase analysis approach for Java workloads that includes computing execution time for

each method invocation, subsequently analyzing the dynamic call graph, and measuring performance characteristics for each of the selected phases. Tajik et al. (2016) define memory phases as opposed to program phases, and illustrate the potential differences between them. They also propose mechanisms for light-weight online memory-phase detection and use the information gathered during memory phases to prioritize keeping individual memory pages resident in a multi-core platform in order to improve memory access latency. Wang et al. (2012) propose a phase detection method for loop-based programs on a multiprocessor system-on-a-chip (MPSoC). Based on the "hot" functions of the program, located by a cross compiling tool based on ARM's RealView Development Suite (RVDS), they target function optimization on a Hadoop cluster. These approaches require hardware support or use interrupts to gain hardware-specific information. While it is conceivable that interrupt driven methods could operate at run time on stock hardware, we are not aware of this having been applied to some program languages such as Python and Java programs. Our approach is designed for different programs on stock hardware.

Shen et al. (2004) detect locality phases using variable-distance sampling, wavelet filtering, and optimal phase partitioning. Singer and Kirkham (2008) investigate the possibility of using concept information within a framework for dynamic analysis of programs. They demonstrate two different styles of concept visualization, which show the proportion of overall time spent in each concept and the sequence of concept execution, respectively. Pirzadeh et al. (2011) propose a trace exploration approach based on examining trace execution phases by adopting the Term Frequency, Inverse Document Frequency (TF-IDF) technique and the cosine similarity measure. Benomar et al. (2014) propose an automatic approach for the detection of high-level execution phases from previously recorded execution traces, based on object lifetimes, and without the specification of parameters or thresholds. The method is simple, based on the heuristic that different phases tend to involve different objects. These papers are interested in phases as part of program analysis and understanding, so the methodologies they use are offline, not run time.

In early work, Sherwood et al. (2002) developed automatic techniques that are capable of finding and exploiting the large scale behavior of programs (behavior seen over billions of instructions). Nagpurkar and Krintz (2004) developed a framework for Java virtual machines that allows application developers, as well as architects of dynamic optimization systems, to visualize, investigate, and experiment with phase behavior data in Java programs. Lau et al. (2005) show graphically that there exists a hierarchy of phase behaviors in programs, and they motivate the need for variable length intervals. Cho and Li (2006) use wavelet techniques to represent program phases at multi-resolution scales to analyze, quantify, and classify the dynamics and complexity of program phases. Watanabe et al. (2008) use an LRU cache for observing a working set of objects, and interpret a sharp rise in the frequency of cache updates at a phase transition. Wimmer et al. (2009) perform local phase detection based on trace trees, which are a collection of frequently executed code paths through a code region, and are generated by trace recording and compilation. Pirzadeh et al. (2010) present a phase detection approach that identifies when and where, during the execution of a program, the methods that implement a particular phase start to disappear as new methods begin to emerge, indicating the beginning of another phase. Zhang et al. (2015) propose an investigation of the potential of multilevel phase analysis (MLPA), where phase analyses with different granularity are combined to improve overall accuracy. These approaches provide rich and useful results, but only after the fact—they cannot be applied at run time.

Nagpurkar et al. (2006) focus on the enabling technology of online phase detection using a similarity model. The method is online in that it works via a single pass over trace data. It consumes profile elements from a trace and transforms them into a sequence of similarity values that represent the degree of similarity between recent profile elements. It employs a similarity analyzer that determines whether the similarity is sufficient to signify that execution is in a phase or in transition between phases. Otte and Richardson (2007) present a semi-online program phase analysis using hidden Markov model (HMM) approaches. They present two approaches to the problem of phase detection and prediction: a Vector

Quantized hidden Markov model (VQ-HMM) and a Continuous Density hidden Markov model (CD-HMM), achieving a 90% score in their best result. They work in a single forward pass over a stream of information about the program. Even though these techniques can achieve high accuracy with lower cost than offline phase detection systems, analyzing each branch as it occurs would still have significant performance impact.

2.2 Phase Information for Saving Energy

Herbert and Marculescu (2009) presented two hardware DVFS (Dynamic Voltage and Frequency Scaling) controllers: a simple threshold-based controller (Threshold) and a more complex greedy-search controller (Greedy) to work on variability information for improving energy-efficiency. To measure variation parameters in this approach, hardware support is required, while our selected features can be instrumented in the program itself and our method works entirely in software.

Bhattacharjee and Martonosi (2009) proposed an energy-efficient algorithm to predict thread criticality and perform DVFS accordingly so as to minimize barrier wait-time. They use cache miss rates to help determine whether a running program is in a CPU-bound or memory-bound phase, making decisions about DVFS settings. This approach shows that system state such as cache miss rates can be predicted and observed to determine DVFS settings, but our model has better prediction performance with respect to energy-efficient clock frequency adjustment.

Xie et al. (2003) derived an analytical model for the maximum energy savings that can be obtained using DVFS given a few known program and processor parameters. They used static code analysis applied to programs at compile time. In this model, they insert special instructions that indicate when the system voltage should change during the execution of the thread. This method could allow finer granularity OS-based control, but it cannot respond dynamically to the conditions of actual program execution.

Shin and Kim (2001) proposed an intra-task voltage scheduling algorithm for hard real-time applications based on average-case execution (RAEP) information. Their algorithm improves energy efficiency by controlling the execution speed based on average-case execution cycles while still meeting the real-time constraints. However, they do not provide energy-delay product results.

Wu et al. (2005b) presented a design framework of a run-time DVFS optimizer in a general dynamic compilation system. They implemented a prototype of the DVFS optimizer and integrated it into an industrial-strength dynamic compilation system. They call for dynamic recompilation of program segments based on the measured operating conditions of an initial execution pass. However, the overhead induced when code is profiled and recompiled is higher than ours (3.7%)—we achieve lower run-time overhead and greater improvement than this model.

Dhiman and Rosing (2007) presented a lightweight DVFS technique for a multi-tasking environment. The technique works by utilizing processor run-time statistics and an online learning algorithm based on cycles per instruction (CPI) to estimate the best voltage and frequency setting at any given point in time. Rangan et al. (2009) presented thread motion, which enables rapid movement of threads to adapt to the time-varying computing needs of running applications on a mixture of cores with fixed but different power/performance levels, as a fine-grained, power-management technique for multi-core systems consisting of simple, homogeneous cores capable of operating with heterogeneous power-performance characteristics. They predict instructions per cycle (IPC) information to support thread motion to save energy. This approach needs to be run in a multi-core environment with thread motion while our model does not. Li et al. (2005) proposed a variable-supply-voltage scaling technique to reduce processor power without undue impact on performance by scaling down the supply voltage of certain sections of the processor during an L2 miss while being able to carry on independent computations at a lower speed. These works measure a particular micro-architectural event, such as a cache miss, that could trigger a

change in DVFS settings, with the setting remaining fixed between such events. They work by analyzing micro-architectural counters rather than proposing feedback control at the hardware level to determine the appropriate operating voltage, giving them high potential for accurate fine-grained DVFS algorithms. However, our system outperforms them in ED product improvement.

Choi et al. (2005) proposed a DVFS technique that relies on dynamically-constructed regression models that allow the CPU to calculate the expected workload and slack time for the next time slot. Thus it can adjust its voltage and frequency in order to save energy while meeting soft timing constraints. Wu et al. (2004) model a multiple clock domain processor as a queue-domain network and model online DVFS as a feedback control problem with issue queue occupancies as a feedback signal. They proposed a dynamic stochastic queuing model that they linearize through an accurate linearization technique. In addition, they proposed a dynamic-time-interval online DVFS scheme in which the reaction time is self-tuned and adaptive to application and workload changes (Wu et al., 2005a). Magklis et al. (2006) proposed to increase the efficiency of a hardware system by separating the core into two clock domains (front-end and back-end) and by allowing independent DVFS for each domain on a clustered GALS (Globally Asynchronous Locally Synchronous) microprocessor. Similar to our work, these proposals measure the values of relevant counters over time and sample this average at some interval to predict the DVFS setting. Furthermore, these micro-architectural techniques respond directly to micro-architectural conditions, giving great potential for accurate fine-grained DVFS algorithms. However, our model achieves better ED product improvement.

CHAPTER 3

EXPERIMENT SETTING AND APPROACH

3.1 Environment

For our run-time measurements we used a server system consisting of two CPUs, each an Intel Xeon E5-2690 v3, clocked at 2.6 GHz. Because these CPUs will otherwise adjust their voltage and clock frequency, we pinned the clock speed to the design point of 2.6 GHz, for consistent results. The server has a large amount of memory and disk space, not relevant to these relatively small programs. These processors have AVX2 vector units, which we exploit to speed the run-time computation of the current cluster for detecting phase transitions. In terms of software, the operating system is Red Hat Enterprise Linux 7.0. For Java programs, we used IBM's J9 Java Virtual Machine, build 2.4 for Java 1.6.0. All the software uses 64-bit mode.

3.2 Choosing Benchmarks

In order to develop real-time phase detection and prediction systems, the benchmarks used are important for evaluation of the systems. Which benchmarks are practical for learning from and evaluating our systems? We believe the benchmarks chosen should satisfy three characteristics, as follows. First, the benchmarks should have high variability. Our systems aim to detect and predict not for a narrow range of programs but for most programs in the programming languages, so we want the benchmark suites to consist of programs that perform different behaviors for testing different aspects of the programming language. Second, each benchmark should have multiple inputs. Our system aims to detect and predict phases in a new run of a program at run time by being trained offline with the same program

with different inputs, so we need multiple inputs for each program in the benchmark suites to provide us with traces that cover more behaviors as training data to learn a better model for that program. Third, a benchmark should not be too long. When developing phase detection and prediction systems, long running benchmarks containing much repetitive behavior provide more information than needed for model training and we don't want to waste computing resources for such a long running process. In short, we need benchmarks that are diverse, with multiple inputs, and where running time is not too long.

3.3 Java Event Traces

3.3.1 Trace Generation

For obtaining fine-grained traces of program features, we apply Elephant Tracks (ET) (Ricci et al., 2011, 2013), a dynamic program analysis tool for Java that produces detailed traces of a variety of program events. ET's original focus was on logging events relevant to garbage collection (object allocations and deaths, and heap mutations), along with method calls and returns (to provide additional temporal context). We modified ET to log each conditional branch that is executed, including the next instruction (i.e., which way the branch goes). We also include method calls and returns in the logs. Each branch instruction is identified by its containing class and method, and a unique index within the method. We employ means to map the features (event types) of different runs of the same program, which may load and exercise some different classes, into a commonly numbered space of features.

3.3.2 Feature Extraction

To build feature vectors, we group branch records together in batches of 100,000. A feature vector then indicates, for each static branch edge in the program (and each method entry and return), how many times that event occurred in that interval of 100,000 branches. (We also include binary features that indicate only whether a particular event occurred or not.)

Why 100,000? The choice of interval length must balance competing pressures. On the one hand, if the interval is too short, the feature counts may be noisy and overly sensitive to the exact boundaries between intervals. Furthermore, if the interval is too short, we need to increase the history length, which means we need to concatenate more feature vectors to get what we need, increasing the complexity of the predictive models. Finally, short intervals, if applied in the real-time setting, incur more overhead in computing the model's output. On the other hand, if the interval is too long, we will not detect phase changes as quickly, and we risk conflating phases.

While we did not explore a variety of interval lengths in our experiments, 100,000 branches per interval seems to be a reasonable value, and corresponds reasonably to previous work.

To shorten feature vectors, we drop features whose value is always zero across all runs of a program. These typically arise because of application code or library code that is not exercised. Such features often comprise 90% of the raw set of features, so this reduction is quite helpful for later processing steps.

3.3.3 Benchmarks Used

Most of our benchmarks come from the DaCapo Java benchmark suite (Blackburn et al., 2006), a set of open source, real world applications with non-trivial memory loads. DaCapo offers Java benchmarking to the programming language, memory management, and computer architecture communities. We use 7 DaCapo benchmarks, with 4 to 19 different inputs for each benchmark.

We add `javac` to the DaCapo benchmarks, the compiler from Java source code to bytecode provided as part of the Oracle Java Development Kit. We modified `javac` to force it to clear its cache of previously compiled classes between each compilation. This gives more iterative behavior and is intended to model a long-running server application. As it compiles a sequence of input source files, each file gives rise to similar, but still somewhat

varying, profiles of the volume of live objects, as observed by Dieckmann and Hölzle (1999). It is also easy to devise inputs to `javac` that require whatever amount of work is desired.

3.4 C Event Traces

3.4.1 Trace Generation

For obtaining fine-grained traces of program features, we have designed Branchgrind, a Valgrind tool (Nethercote and Seward, 2007) that produces detailed traces of a variety of branch and cache events. Branchgrind logs each conditional and indirect branch that is executed, including its position in the code and the next instruction (i.e., which way the branch goes). Branchgrind also logs the type (function call, function return, and local) and the flag (taken vs. not taken) of branches. Each branch instruction is identified by its address in virtual memory. Furthermore, as we discussed in Section 3.3.2, we group branch records together in batches of 100,000 to build feature vectors and drop features whose value is always zero across all runs of a program to shorten feature vectors.

3.4.2 Feature Extraction

To build feature vectors, we group branch records together in batches of 100,000 as described in Section 3.3.2.

3.4.3 Benchmarks Used

Our benchmarks come from the SPEC CPU2006 benchmark suite (Henning, 2006), a collection of 30 compute-intensive, non-trivial programs that run across the widest practical range of hardware, used to evaluate the performance of a computer's CPU, memory system, and compilers. The benchmarks in this suite were chosen to represent real-world applications, and thus exhibit a wide range of run-time behaviors. We use all 30 SPEC CPU2006 benchmarks, with their "train" and "ref" inputs, to generate 84 traces.[1] There are 48 short

[1] Some benchmarks, such as `gcc`, run the program more than once for a given "input".

traces in which we are not interested because they have very few time steps for model learning.

3.5 Python Event Traces

3.5.1 Trace Generation

For obtaining fine-grained traces of program features, we designed BranchProfiler, a dynamic program analysis tool for Python that uses the existing cProfile package. cProfile provides deterministic profiling of Python programs. A profile is a set of statistics that describes how often and for how long various parts of the program executed. These statistics can be formatted into reports via the pstats module. BranchProfiler is implemented in C via Cython in order to reduce the overhead of profiling. BranchProfiler logs each conditional branch that is executed, including its position in the source code and the next line of source code (i.e., which way the branch goes). BranchProfiler also records calls and returns. We further developed means to map the features (event types) of different runs of the same program, which may load and exercise different python functions, into a commonly numbered space of features. Here a feature is a particular branch, and for conditional branches, which way the branch goes.

3.5.2 Feature Extraction

To build feature vectors, we group branch records together in batches of 10,000. A feature vector then indicates how many times an event happened for each branch, function call, and function return in that interval of 10,000 branches.

While we did not explore a variety of interval lengths in our experiments, 10,000 branches per interval seems to be a reasonable value, and corresponds well to Java work (Chiu et al., 2016) since Python code is more concise and compiles into smaller executables than Java (Nanz and Furia, 2015).

To shorten feature vectors, we drop features whose value is always zero across all runs of a program. Such features often comprise 20% of the raw set of features.

3.5.3 Benchmarks Used

Our benchmarks come from the Pyperformance suite (Gaynor et al.). Pyperformance is intended to be an open and authoritative source of benchmarks for all Python implementations. The focus is on real-world benchmarks, rather than synthetic ones, using whole applications when possible. We use 13 of 44 Pyperformance benchmarks, with 3 to 16 different inputs for each benchmark. The other 31 benchmarks are very short, having few branches. They cannot be partitioned into enough phases for useful phase prediction.

CHAPTER 4

REAL-TIME PROGRAM PHASE CHANGE DETECTION

We now introduce a method for real-time phase change detection. A portion of this work appears in PPPJ 2016 (Chiu et al., 2016). After applying a training protocol to a program of interest, our method can detect phase changes at run time for that program with good precision and recall (compared with a "ground truth" definition of phases) and with small performance impact. To accomplish this, we first trace a number of executions of the program of interest, recording information about each call/return and conditional branch that occurs. These traces are suitable for computing "ground truth," that is, phases and phase changes according to a precise definition. Ground truth is defined in terms of all the branch instructions exercised in the program.

We also use the traces to develop an efficient real-time phase change detector, as follows. We choose a small number of branch instruction that we will instrument at run time. Next, we develop *Gaussian mixture model* (GMM) (McLachlan and Peel, 2004) that clusters feature vectors. At run time, we count the instrumented branches, and when their total count reaches a threshold, we determine a cluster for the recent past. If it is different from the previous cluster, we judge the program to have transitioned from one phase to another.

Our method does not significantly impact performance because (1) it instruments only a small number of branches, (2) it evaluates the clustering model only occasionally, and (3) the clustering model is not overly expensive to evaluate. Our method corresponds well to ground truth because (1) the GMM model is faithful to the ground truth, even though it uses a much smaller number of features, and (2) the run time modeling does not diverge much from that of the training runs of the same program.

4.1 A Machine Learning Model for Phase Transition Detection

In our pursuit of accurate real-time phase transition detection, machine learning and feature selection play an important role. Our approach to detecting transitions is to cluster time intervals based on the similarity of their feature vectors. We consider there to be a phase *transition* whenever the current phase differs from the previous one. The specific clustering model we use is the Gaussian mixture model (GMM) (McLachlan and Peel, 2004) [1]. This is a probabilistic model that can be seen as generalizing k-means clustering to incorporate information about the covariance structure of the clusters. GMMs assume the data (feature vectors) are generated from a mixture of a finite number of Gaussian distributions (the clusters) with unknown parameters. Each feature vector has a probability of having been generated by the Gaussian distribution associated with each cluster. A feature vector is assigned to the cluster for which its probability is highest. We associate clusters with phases in a program.

At first it might seem strange that this would work. After all, a program can have an arbitrary number of phases, and to apply GMMs we must choose a number of clusters in advance. As our results show (Chiu et al., 2016), at least for the range of Java programs/runs we considered, the quality of the results is not strongly sensitive to the number of clusters we form, as long as there are more than just a few. Since different loop structures, recursions, etc., result in feature vectors that point in different directions (recall that feature vectors are based on counts of traversals of branch edges), in the end we found it not so surprising that this approach might work.

4.1.1 Feature Selection

No matter what program languages work with, our feature vectors from the traces might have thousands or tens of thousands of features. While the clustering algorithms might work on these, for real-time phase transition detection we cannot use all the features—the

[1] We use python scikit-learn 0.19.1 package to implement GMM

instrumentation cost would be very high. Rather, we need to choose a smaller number of features that still capture the various phases. We have observed that for Java programs many features in our vectors are highly correlated with one another, suggesting that there is a lot of underlying redundancy and that reduction in dimensionality by feature selection is likely to work well in practice.

The features we select for clustering need to satisfy two characteristics: high diversity (low correlation with each other), and low cost to collect. Increasing the diversity of features gives GMMs more ability to identify cluster structure in the data, potentially increasing the ability of our model to differentiate various phases. Selecting features that cost less to collect from a running program is important for us in building a real-time system.

To achieve high diversity, we randomize the order of the features, and then select features sequentially. For each new candidate feature, we compute its correlation with previously selected features, and reject the candidate if it is highly correlated (i.e., correlation above a chosen threshold) with any previously selected feature. We use a threshold of 0.80 for the Pearson correlation coefficient. We iterate this process until we have selected a target number of features. In our experiments we try various numbers of features, and find that a surprisingly small number of features is adequate. We also perform this randomized feature selection process a number of times, and report the distribution of results.

After choosing features using the filtered randomized process just described, for each chosen feature we consider replacing it with a lower cost proxy feature. The point of this step is to reduce the run-time cost of collecting feature event counts. A chosen proxy must be highly correlated (we use the same threshold as before) with the originally chosen feature. We sort the features in increasing order of cost to collect, assuming that the number of times the feature's event occurs is a good estimate of that feature's cost. For each originally selected feature, we find the first feature in the cost-sorted list that is highly correlated with the original, and replace the original with that feature (or keep the original, if we do not find a lower cost one). We do not explicitly demand that the new feature be uncorrelated with

the other original features as this property should be sufficiently maintained by transitivity of correlation.[2] In sum, we choose a small number of features, uncorrelated with each other, and with low cost to count at run time. This helps guarantee that our run-time overheads for real-time phase transition detection will be as small as possible and the quality high.

4.1.2 Evaluation of the GMM Clustering Approach

To understand the quality of results we might achieve with GMM clustering, we compare it with the detector of Nagpurkar et al., hereafter referred to as ND (for Nagpurkar et al.'s Detector), and with a detector that simply makes phase transition decisions at random, which we call Random. We use the baseline definition of phases and transitions of Nagpurkar et al. (2006), hereafter called Baseline, described further below. The baseline defines phases in terms of a parameter one must supply, the minimum phase length (MPL).

ND works by considering what amounts to our feature vectors (in their weighted set model), but with all features included. Their method indicates phase transitions according to the similarity of the current window's feature vector with that of the previous window.

The Random scheme works by simply choosing "in phase" with probability p and "in transition" with probability $1-p$, where p is the proportion of windows that Baseline identifies as in-phase.

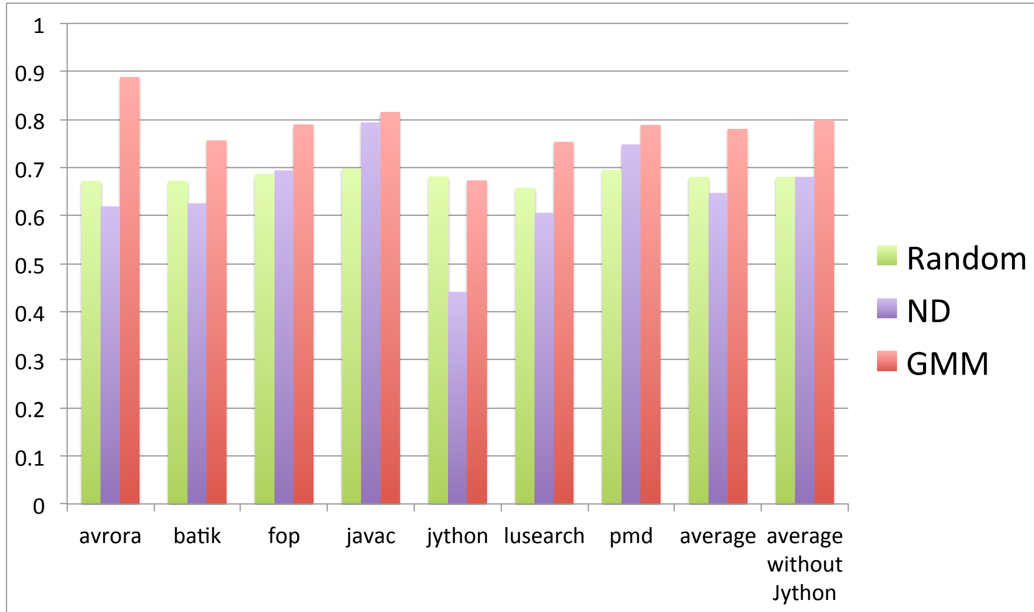
Baseline works by building up phases from *complete repetitive instances* (CRIs). A CRI is either a complete recursive call (not nested within another call of the same routine), which can include just a single non-recursive call, or a loop execution (all iterations) within a (possibly) recursive execution. The scheme concatenates adjacent CRIs and indicates in-phase if the concatenated CRIs meet the minimum phase length (MPL) criterion. Otherwise it indicates in-transition.

[2] In other words, if features A and B have low correlation and features A and A' have high correlation, then features A' and B will tend to have low correlation.

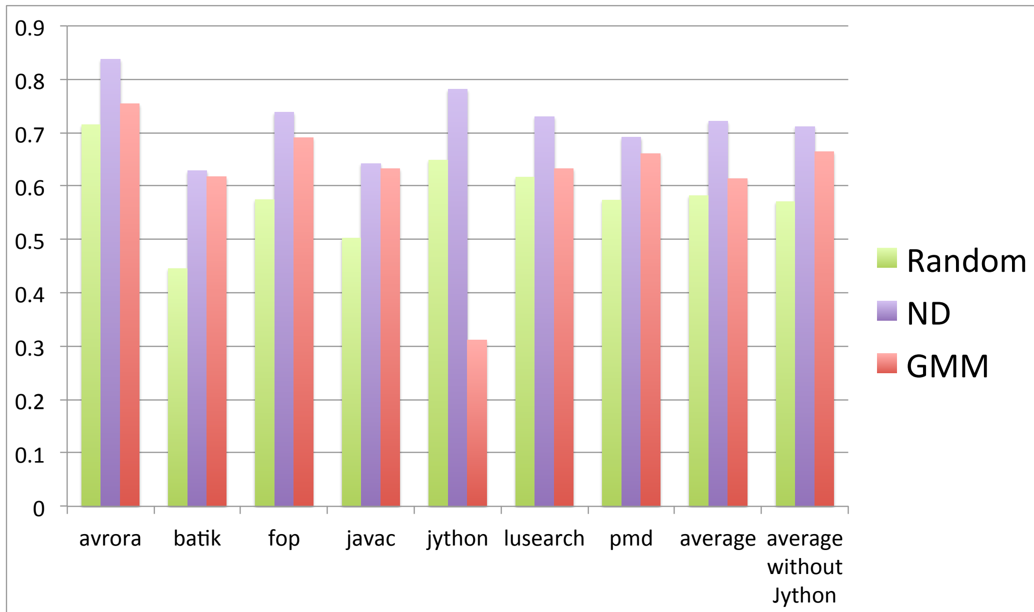
To evaluate GMM, ND, and Random, we explored two scoring metrics, the Accuracy Score of Nagpurkar et al. (2006), and an F-score based on measuring precision and recall, with shifted weights (so that phase transitions need not be recognized exactly when they occur). All scoring is based on comparing a given scheme's judgment of in-phase vs. in-transition against Baseline's determination. The accuracy score of Nagpurkar et al. (2006) is a weighted sum of three comparison measures. The first, which they call "correlation" (more usually called accuracy), is the fraction of windows in which the judgments agree. The other two measures are based on how well a scheme's phase transition boundaries agree with Baseline. Their "sensitivity" measure is the fraction of Baseline boundaries that the detector also identifies as boundaries. Their third measure is the fraction of a detector's reported boundaries that match Baseline boundaries. The total score is a weighted sum of these, where the weights are 50% correlation, and 25% each for the other two measures. (For details of what is recognized as a matched or unmatched boundary, we refer the reader to their paper.)

In our experiments (Chiu et al., 2016), we found that this measure did not discriminate well in various cases. Figure 4.1a shows Random obtains a score of about 70%, about the same as ND. In particular, when there are few phase transitions, it gives a high score even if a detector reports no transitions at all. Likewise, Random achieves a high score when there are few transitions. Therefore we developed an alternative, based on the F-score used in document retrieval and similar evaluations. The F-score combines precision and recall into a single number. Precision is the fraction of detected transitions that correspond to actual ones in Baseline, while recall is the fraction of actual Baseline transitions found by the detector. Given precision p and recall r, the F-score is defined as $2 \cdot \frac{p \cdot r}{p+r}$.

However, given that detection may lag actual transitions, we find it reasonable to allow for approximate matching of transitions, but with a lowered score. Therefore, we developed a scheme that gives a weighted value to transitions that are not perfectly aligned. We call it the *Shifted Weighted F (SWF) score*, and it works as follows. We consider a set of window



(a) Accuracy Score for GMM, ND, and Random



(b) SWF Score for GMM, ND, and Random

Figure 4.1: Performance of GMM, ND, and Random detectors. The offline GMMs shown use 8 selected features and 8 clusters. The threshold for ND is 0.8.

sizes 1 through δ. Assuming that there are N original time periods, for a given window size w we consider all $N-(w-1)$ distinct overlapping windows containing w consecutive elements. For each window, we consider a transition to be found by a given detector (or

Baseline) if for the detector (respectively, Baseline) any element in the window indicates transition. Thus, window size 1 considers perfectly aligned transitions, while window size 2 considers those within one time step of one another, etc. We compute precision and recall on the vectors of length $N-(w-1)$ and, to penalize for non-alignment, divide the result by w. We sum these values for w from 1 up to a chosen limit δ. To normalize the result, we divide by the similar number obtained from comparing a detector (or Baseline) with itself, as appropriate for precision and recall. It is these normalized p and r values that we use to compute SWF.

Here is how we choose δ. If windows are very large, then we will almost always find transitions within them. Therefore, we desire δ small enough that the likelihood (for Random) that there is no transition within a window will be at least α for some chosen α. This results in the following equation:

$$\delta < \frac{\log \alpha}{\log(1-(1/\mathrm{MPL}))}. \tag{4.1}$$

MPL appears in this equation because, in Baseline, MPL gives the minimum possible distance between phase transitions. Equation 4.2 expresses the solution to this equation, which gives the probability of no phase transition within δ steps:

$$\alpha > (1-(1/\mathrm{MPL}))^{\delta} \tag{4.2}$$

We chose to use $\alpha = 0.1$ to determine our δ value.

Figure 4.1b shows how the SWF score discriminates between Random and the other detectors. We believe that the SWF score is a more appropriate metric for these comparisons.

4.1.3 Number of Clusters and Features

The number of GMM clusters and number of selected features we choose to use are important parameters of our real-time phase transition detection system. Having more clusters or

selected features increases the cost of the run-time phase change detection function, which is linear in the number clusters and number features. Having fewer clusters or features, though, may decrease the quality of the detector. In order to choose an appropriate number of clusters, we developed GMM clusterings with various numbers of clusters and features. Figures 4.2 and 4.3 show, for GMMs, going beyond eight clusters does not give much additional improvement, and the model needs at least twelve selected features to achieve good quality in phase transition detection.

4.2 Real-Time Program Phase Change Detection in Java

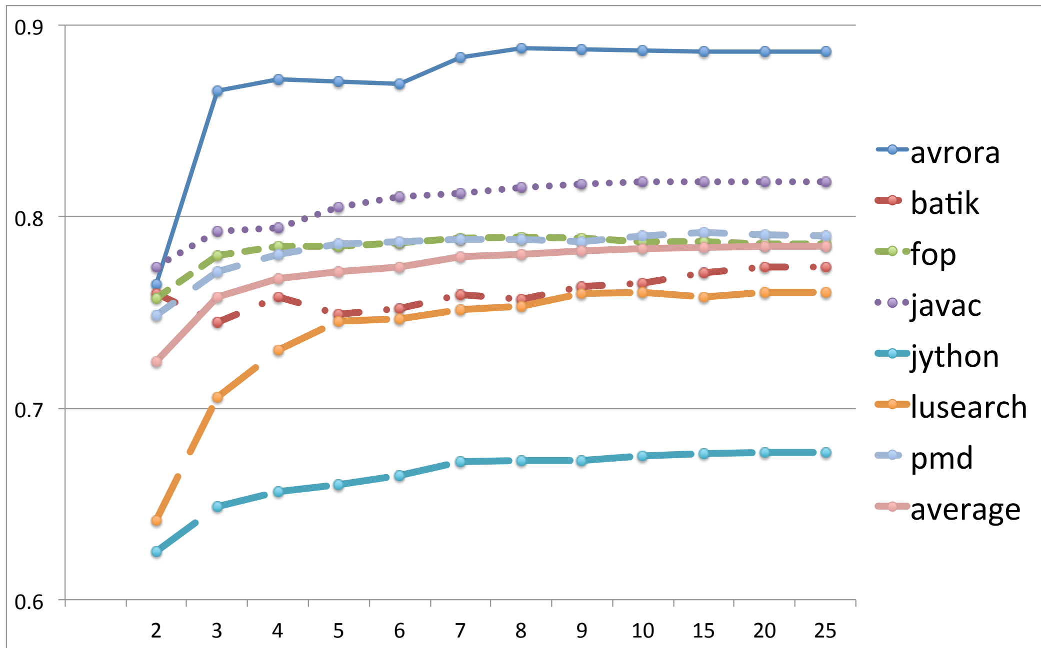
4.2.1 Offline Evaluation

Figure 4.4 presents offline evaluation results for GMMs using 8, 12, 16, and 24 features. Even though our GMMs use fewer features than ND (which uses complete feature sets), they actually perform similarly on the Accuracy Score, an average of 67% versus ND's 72%, and both are higher than Random's 58%. The GMM approach achieves a similar score to ND for all programs except `jython`. GMMs might not give the best results for an offline analysis unconcerned with real-time performance, but give surprisingly good quality in the face of using such little information. Note that ND uses a similarity threshold parameter, which affects the quality of its results. We used the best value of those reported by Nagpurkar et al. (2006), namely 0.8.

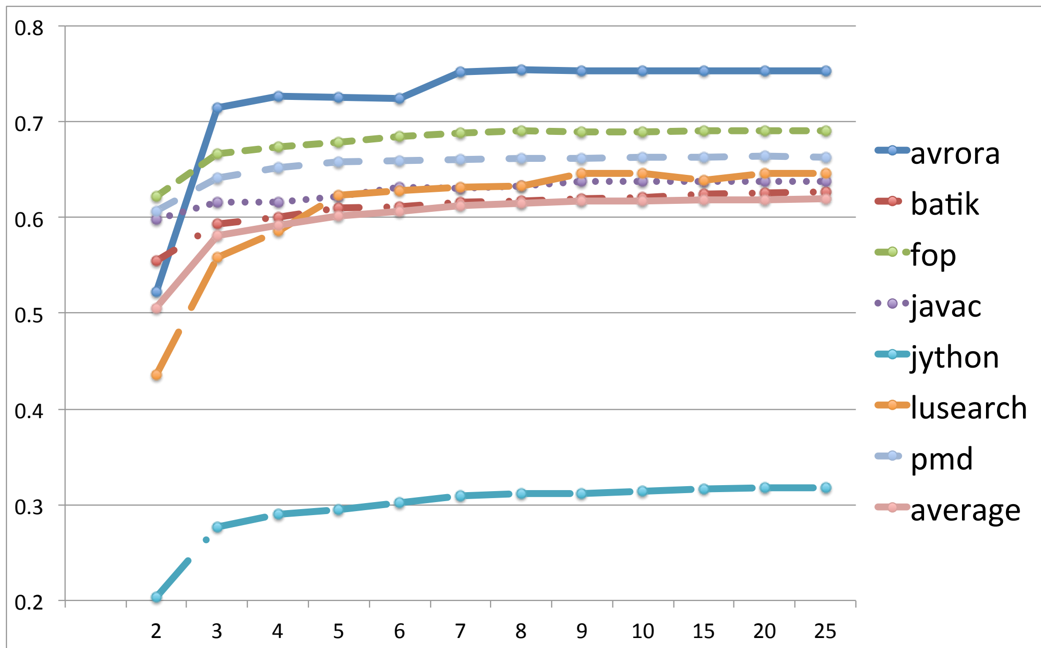
4.2.2 Implementing the Real-Time Detector for Java

We implemented the phase transition detector as a Java agent, using the Java Virtual Machine Tool Interface (JVMTI).[3] Each instrumented feature calls a Java method to increment a counter for that feature. It also counts down a total number of events. Once the desired number of events is reached, the Java counting method calls a native method in the agent, which

[3]We use ibm-java-x86_64-60 Java virtual machine.



(a) Accuracy Score of GMMs for various numbers of clusters



(b) SWF Score of GMMs for various numbers of clusters

Figure 4.2: Performance of GMMs with various numbers of clusters

obtains the counter values and then, using vector intrinsics supplied by gcc, determines the highest probability cluster using probabilistic inference.

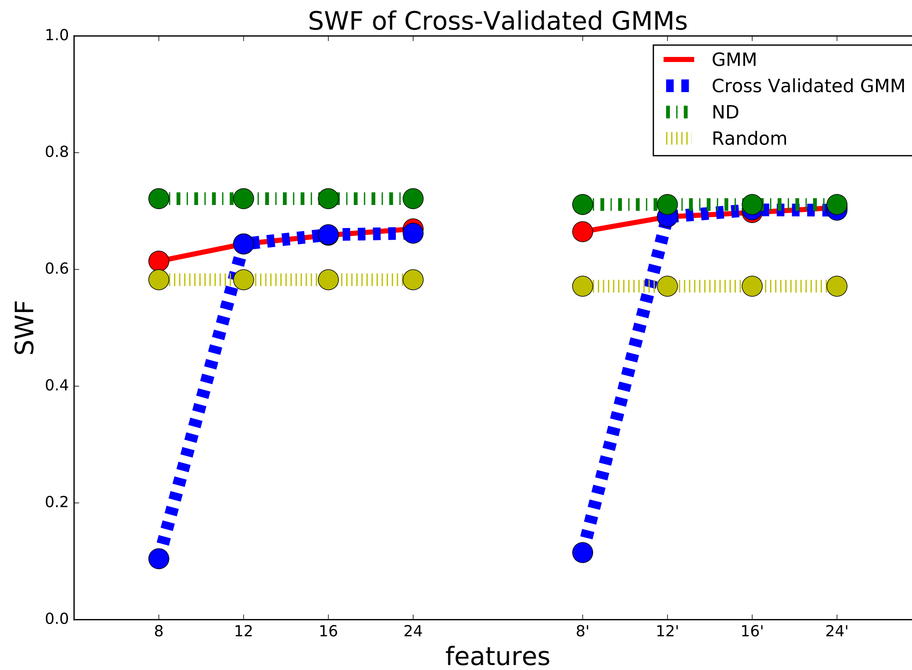


Figure 4.3: SWF Score for self-test GMM, cross-validated GMM, ND, and Random detectors, varying the number of features used by the GMMs. All GMMs use 8 clusters. The threshold for ND is 0.8. The unprimed x-axis give the number of features for averages that include `jython`, and the primed one for those that do not.

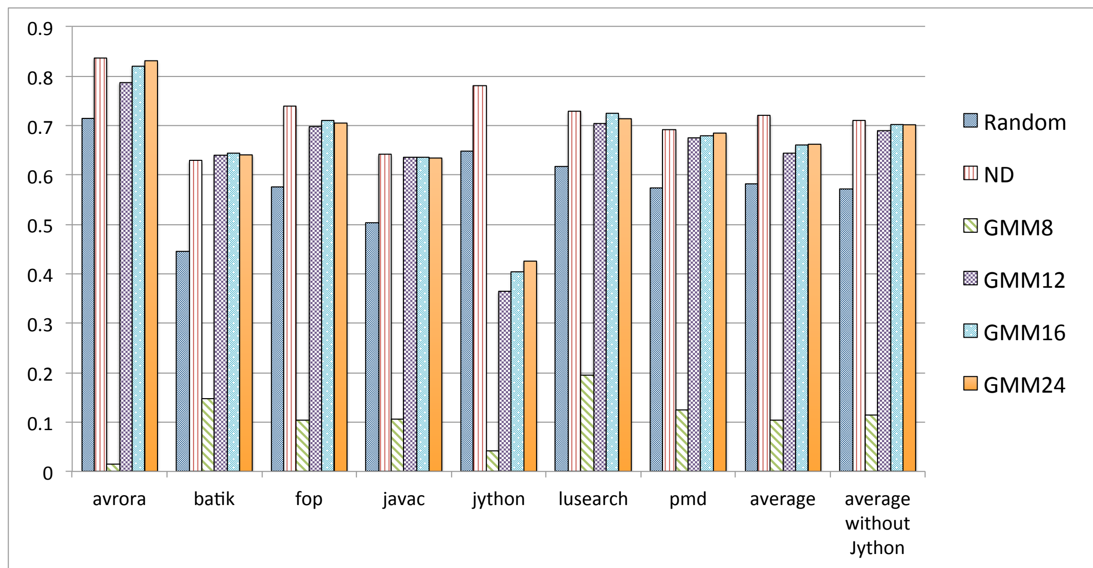


Figure 4.4: SWF Score for offline cross-validated GMMs with 8, 12, 16, and 24 selected features, including Random and ND for comparison for Java programs. All GMMs use 8 clusters. The threshold for ND is 0.8.

To insert the instrumentation into the Java code, we use a modified version of Elephant Tracks that instruments only the selected features. It does this by applying bytecode rewriting to the compiled classes of the application. We perform these rewrites as the code loads. Note that we iterate runs of a benchmark program to factor out JVM JIT compilation costs, etc., which also factors out our bytecode rewriting. However, the cost of that rewriting is not extreme, and is incurred only on start up. The modified version of Elephant Tracks also automatically generates the Java class holding the event counters and counter increment methods, which the Java agent then loads into the JVM. We record the cluster number predicted by the GMM model in a memory buffer that we dump out at the end of a program run.

4.2.3 Results

We now consider the run-time overhead of our real-time detector, and its quality as a detector.

4.2.3.1 The Run-Time Overhead of Our System

We compare the cost of running with and without the instrumentation that our system adds. For each benchmark and input, we selected features some number of times, and then performed multiple runs of the instrumented program. We also performed 30 runs of each program and input with no instrumentation. We did runs with 8 features early on, to assess the acceptability of the overhead of our instrumentation. These runs were for a self-test version (mode ST in the table). Given the poor score with 8 features, we did not do cross-validated runs for that case. The runs for 12 and 16 features are cross-validated, however (mode CV).

We insert instrumentation into a program's class files in advance, using a different version of our Java agent that simply rewrites byte codes at the instrumented code locations. At run time we use a very simple agent that just collects and dumps out phase detection

statistics. Overall, we use three agents, one when generating the full ET trace, one when instrumenting classes, and one when running the instrumented application.

We find, as one would expect, that the run-time overhead tends to increase as we instrument more features. The smallest number of features that gives good detection quality is 12, resulting in average overhead of 2.0%. Using 16 features increases the overhead to 5.0%.

In addition to observing that the average overhead for 12 features is less than 2%, we see that only 13 of 48 traces incur overhead of 1% or more. Curiously, a number of program inputs perform better with instrumentation. We can only speculate as to why, but possibilities include impact on the JIT compiler, branch predictors, and data and instruction caches. Of course one would expect the cost to go up, but if some inner loops happen to run faster we could see this kind of effect. We found a small number of outliers in the underlying data. For example, `lusearch-default` typically runs in about 14 seconds, but one instrumented run took 91.6 seconds. Also some `pmd` runs took 39 times longer when instrumented. To test the statistical significance of the difference in running times, we performed T-tests comparing the instrumented and uninstrumented time distributions. The 12 feature case shows that there is a 11% chance that the two distributions were drawn from the same underlying distribution.

4.2.3.2 The Quality of the Real-time Detector

There are a number of reasons why real-time phase detection might work differently from original program runs. A program may not execute in exactly the same way, even on the same input—even deterministic Java programs may be affected by background threads in the Java system. However, an effect of more direct concern is that we cannot easily trigger our phase detection function at the same points during execution that the offline method uses. This is because we are not counting *all* the conditional branches. We choose a feature from our selected feature set with small variation from window to window across the run,

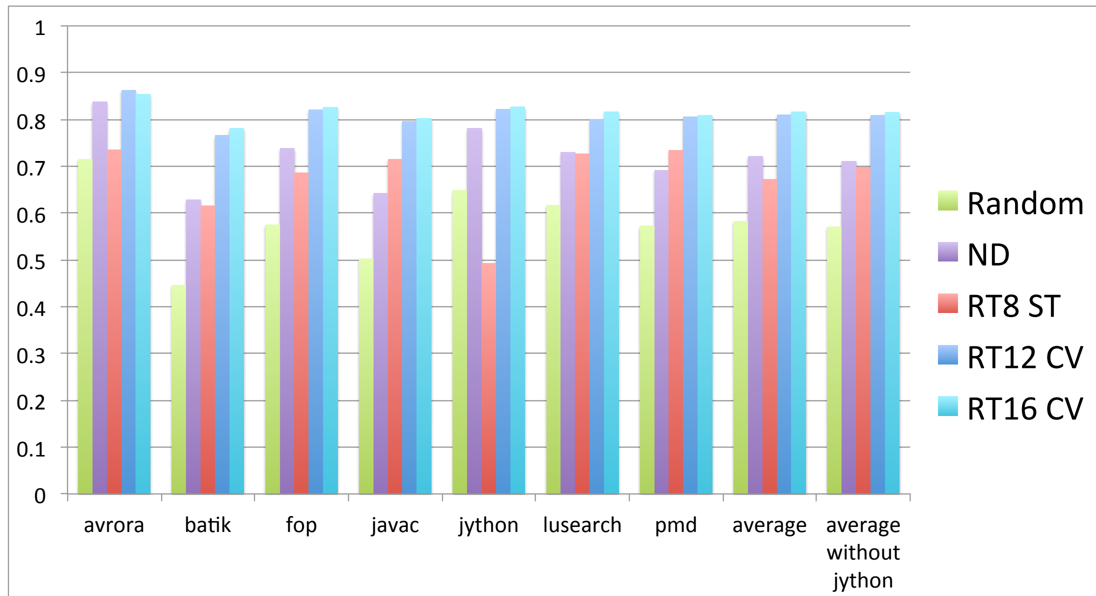


Figure 4.5: SWF Scores for RT, ND, and Random detectors compared with Baseline for Java programs. The number indicate the number of features for the RT scheme; ST = Self-Test; and CV = Cross-Validation. All RT detectors use 8 clusters. The threshold for ND is 0.8.

and we use it as a proxy for time, triggering our phase detection function when this feature has occurred the average number of times that it occurs in a window in the original ET trace.

The most interesting comparison is against Baseline, i.e., "ground truth," but given the different lengths of time intervals, how shall we compare? First, we assume that each detector's intervals are of uniform length in real time. Then, if detector B reports a transition anywhere during a given interval of detector A, we consider B to have reported a transition for that interval. If we compare against Baseline, and use the same number of time intervals for each detector as just described, we obtain the SWF scores shown in Figure 4.5.

We find that cross-validated RT gives good results, better than ND, when comparing with Baseline, and much better than Random. RT 12 achieves an average SWF score of 81%, compared to 58% for Random and 72% for ND. In sum, our real-time phase transition system can detect phase changes with good precision and recall and with small impact on execution time.

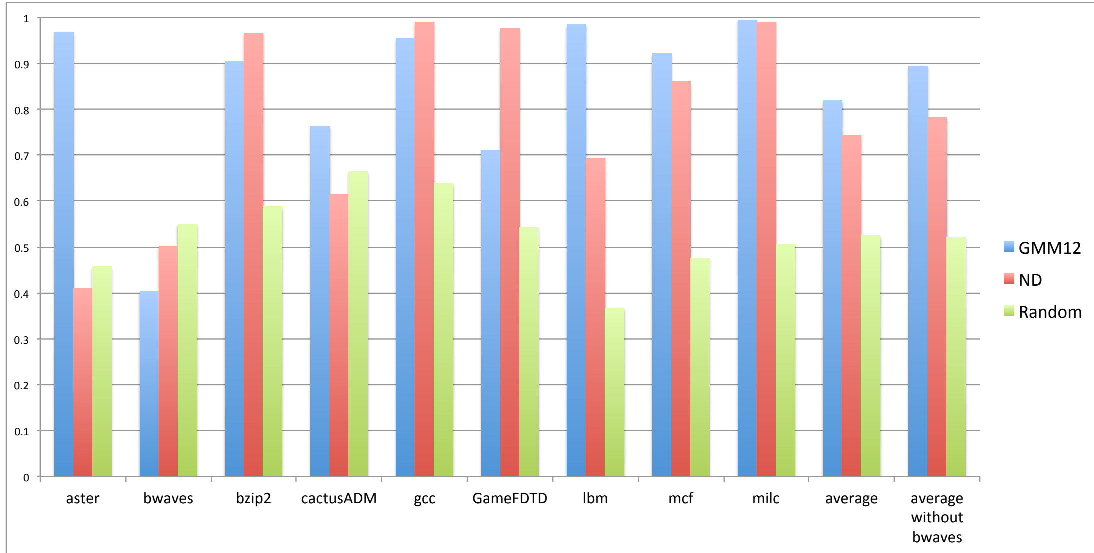


Figure 4.6: SWF Score for offline cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for C programs. The threshold for ND is 0.8.

4.3 Real-Time Program Phase Change Detection in C

4.3.1 Offline Evaluation

Figure 4.6 presents offline evaluation results for GMMs using 12 features. The results show GMMs actually perform better on the SWF score, an average of 82% versus ND's 74% and Random's 52%. The GMM approach achieves more than 90% score in 6 of 9 programs, and achieves better score than ND and Random for all programs except `bwaves`. Without `bwaves`, the GMM approach achieves 89% SWF score.

4.3.2 Implementing the Real-Time Detector for C

We implemented a tool to instrument C programs (compiled binaries, actually) for phase change detection. To insert the instrumentation, we implemented a PEBIL tool (Laurenzano et al., 2010), which does this by using function level code relocation. Each instrumented feature increments a counter for that feature. It also counts down a total number of occurrences of the feature that has smallest variation from window to window across the run. Once the desired number of events is reached, the phase detection function is triggered and then, using vector intrinsics supplied by `gcc`, determines the highest probability cluster

using probabilistic inference with the learned GMM model. We record the cluster number predicted by the GMM model in a memory buffer that we dump out at the end of a program run.

4.3.3 Results

We now consider the run-time overhead of our real-time detector, and its quality as a detector.

4.3.3.1 The Run-Time Overhead of Our System

As described in Section 4.3.3, we measure the overhead of our system with 12 features. Table 4.1 shows that the GMMs with 12 features give acceptable quality with only 2.2% overhead on average. We also performed T-tests comparing the instrumented and uninstrumented time distributions, to test the statistical significance of the difference in running times. The t-test result shows that there is only a 15% chance that the two distributions were drawn from the same underlying distribution.

Table 4.1: Ratio of running time of programs with 12 features to the original benchmark programs for C.

Benchmark	RTFs 12
astar	1.033
bwaves	1.009
bzip2	1.020
cactusADM	1.066
gcc	1.014
GameFDTD	1.007
lbm	1.014
mcf	1.028
milc	1.008
Average	1.022

4.3.3.2 The Quality of the Real-time Detector

Figure 4.7 shows the SWF scores that our GMM-based run time detector, ND, and Random achieve, comparing against Baseline. We see that GMMs with 12 features and

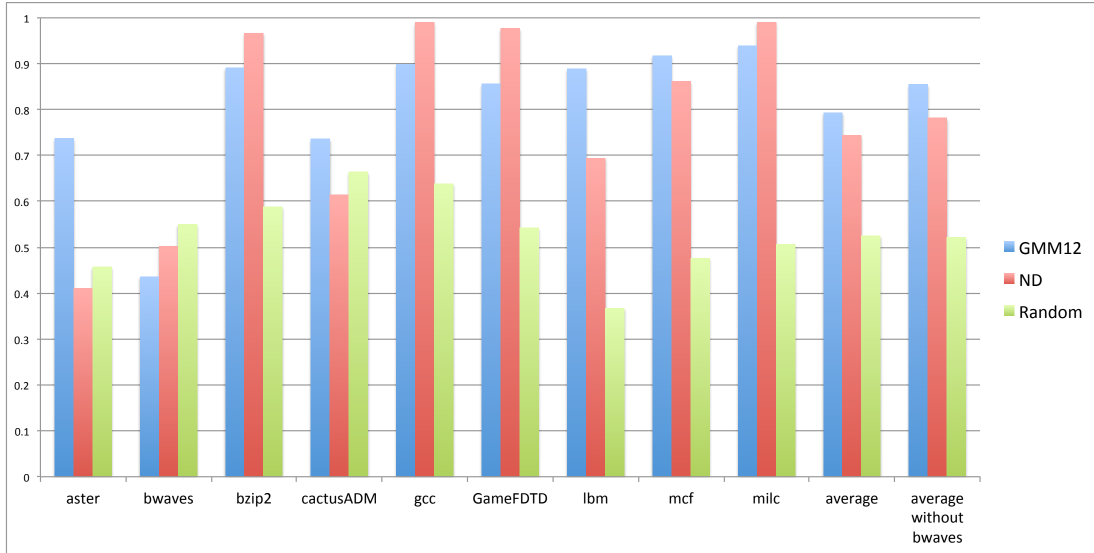


Figure 4.7: SWF Score for online cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for C programs. The threshold for ND is 0.8.

8 clusters detect phase changes with high quality, better than ND and Random, when comparing with Baseline. GMM 12 achieves an average SWF score of 79.3%, compared to 74.4% for ND and 52.5% for Random, and without `bwaves`, in which other models outperform the GMMs in offline experiments, the GMM approach achieves 85.5% SWF score.

4.4 Real-Time Program Phase Change Detection in Python

4.4.1 Offline Evaluation

Figure 4.8 presents offline evaluation results for GMMs using 12 features. Even though our GMMs use fewer features than ND (which uses complete feature sets), they actually perform better on the SWF score, an average of 86% versus ND's 60% and Random's 66%. The GMM approach achieves more than 90% score except 4 programs, and achieves better score than ND and Random for all programs except `spambayes` and `sympy`. Note that Random has better performance than ND because ND achieves 0% SWF score in 3

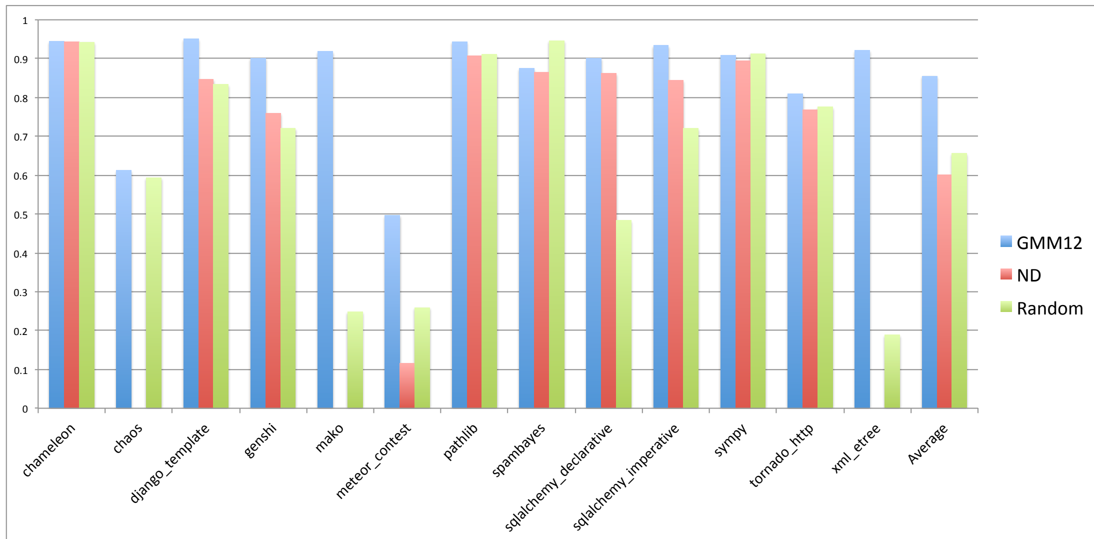


Figure 4.8: SWF Score for offline cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for Python programs. The threshold for ND is 0.8.

programs for which ND detects no transition. After excluding these programs, ND achieves 78% SWF score, which is better than Random's 75% but still not good as the GMM's 87%.

4.4.2 Implementing the Real-Time Detector for Python

We implemented the phase predictor inside the CPython implementation—similar to the trace module of the Python[4] standard library. Each instrumented feature is a key into a hash table of counts, with the count being incremented when that feature (branch) occurs. The code also counts a feature that has low variance across the traces, in order to trigger periodic invocation of the detection function. We record the cluster number predicted by the GMM model in a memory buffer that we dump out at the end of a program run.

4.4.3 Results

We now consider the run-time overhead of our real-time detector, and its quality as a detector.

---

[4]We use Python 2.7

4.4.3.1 The Run-Time Overhead of Our System

As described in Section 4.3.3, we measure the overhead of our system with 12 features. Table 4.2 shows that the overhead of 12 features is 2.6% overhead on average and the t-test result shows that there is only a 12% chance that the two distributions were drawn from the same underlying distribution, so the effect is real.

Table 4.2: Ratio of running time of programs with 12 features to the original benchmark programs for Python.

Trace	RTFs 12
bm_chameleon	1.037
bm_chaos	1.020
bm_django_template	1.001
bm_genshi	1.002
bm_mako	1.039
bm_meteor_contest	1.046
bm_pathlib	1.034
bm_spambayes	1.007
bm_sqlalchemy_declarative	1.008
bm_sqlalchemy_imperative	1.032
bm_sympy	1.022
bm_tornado_http	1.038
bm_xml_etree	1.053
Average	1.026

4.4.3.2 The Quality of the Real-time Detector

Figure 4.9 shows the SWF scores that our GMM-based run time detector, ND, and Random achieve, comparing against Baseline. We see that GMMs with 12 features and 8 clusters detect phase changes with high quality, better than ND and Random, when comparing with Baseline. GMM 12 achieves an average SWF score of 73%, compared to 63% for Random and 55% for ND.

When comparing the results shown in Figure 4.9 and Figure 4.8, we find that our real-time detectors have lower scores versus the offline GMMs for most programs. One factor that has direct influence on these differences is that the two systems have different lengths of

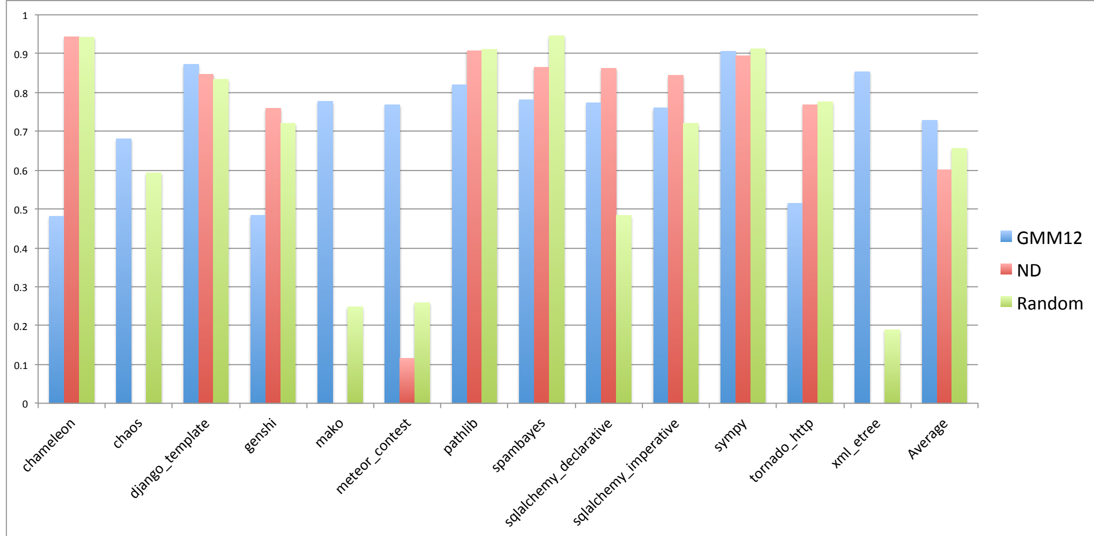


Figure 4.9: SWF Score for online cross-validated GMMs with 12 selected features and 8 clusters, including Random and ND for comparison for Python programs. The threshold for ND is 0.8.

time intervals when we use the feature with small variation from window to window across the run as a proxy for time. For one program run in each of `genshi` and `tornado_http`, that contain only three traces, the GMMs detect no transition and achieve 0% SWF score, lowering the score on average. After excluding these programs, the real-time detector achieves 77% SWF score, compared to 57% for Random and 64% for ND.

4.5 Summary

In general, after applying a training protocol to a program of interest, our method can detect phase changes, at run time, for that program with good quality and without significant run-time overhead for all three programming languages, the over head being 2.0% for Java, 2.2% for C, and 1.7% for Python. Our experimental results show that our detector corresponds well with the underlying definition of phases ("ground truth"). The SWF score that our real-time phase transition detector achieves is better than offline ND and the random detector for all three programming languages.

Our methodology achieve highest F-score for the Java programs, with 81% SWF score for detecting phase transitions, and the C results are similar (79%). For the Python programs, our methodology has lower SWF score (73%) compared to the model's performance for Java and C programs, but the quality versus ND is not significantly different: 5% to 10% improvement compared to ND in all three programming languages. (Note that GMM uses less information than ND, which uses complete feature sets and cannot be run without significant performance impact.)

The GMM approach gives surprisingly good quality in the face of using little information, but it does not work well with a few programs, i.e., `bwaves` (C) and `genshi` (Python). We will explore other machine learning models to enhance our detector in future work.

CHAPTER 5

REAL-TIME PROGRAM PHASE PREDICTION

We now introduce a method for run-time phase prediction[1]. After training on different runs of each program of interest, our method can predict the future phase k steps later, for $k = 1, \ldots$, at run time, for a new run of the same program but on different inputs, with good precision and recall (compared with a "ground truth" definition of phases) and with small run-time overhead.

To achieve this, we first generate traces from a number of executions of the program of interest, recording each function call/return and branch that occurs, accumulating them into feature vectors. These traces are also suitable for computing "ground truth", that is, phases computed by clustering the feature vectors within phase boundaries derived directly from traces according to a precise definition (that of Nagpurkar et al. (2006)). Ground truth is defined in terms of all the branch instructions exercised in the program.

We also use the traces to develop an efficient run-time phase predictor, as follows. We choose a small number of features that we will count at run time. Next we apply decision trees built over this small number of features to do the actual phase prediction. At run time, we count the selected branches, and count down a total number of events, in order to trigger periodic invocation of a function that encodes the predictor's decision tree.

Our method does not induce significant run-time overhead because (1) it instruments only a small number of features, (2) it triggers decision tree model evaluation only occasionally, and (3) the decision tree model is not very expensive to evaluate. Our method corresponds

[1]This work is submitted to 15th International Conference on Managed Languages & Runtimes (ManLang'18).

well to ground truth because (1) the decision tree model is faithful to the ground truth even with only a small number of features, and (2) the run-time model does not diverge much from the decision tree learned on training runs of the same program (with different inputs).

5.1 A Machine Learning Model for Phase Prediction

In our pursuit of accurate and low overhead run-time phase prediction, we now turn to using machine learning methods to build a prediction model. In the end, we will use decision trees built over a small number of features to do the actual phase prediction. However, this means we need an effective way to select which features to use. That will be a two step process. The first step is determining which features are good for predicting phases. To do that, we use properties of decision trees *over single features* to obtain a quantitative measure of each feature's predictive ability. We then use a heuristic technique to select a small, but effective, subset of branches (features) to instrument at run time. From those features we build the decision trees used for run-time prediction. This section concludes with a theoretical (i.e., offline) evaluation of the performance of the decision trees.

5.1.1 Feature Selection

In order to predict future phases, we need to select features that are correlated with the phases. However, phases are represented as numbers in the ground truth and these numbers cannot reasonably be compared numerically since they are arbitrary identities, i.e., categories. Hence, to obtain an initial quality ranking for each feature on the phase prediction task, we need a scoring mechanism to estimate the predictive value of individual features on the phase prediction task.

We train a decision tree model for each *individual* feature, using entropy to measure the quality of a split, and assess a decision tree's predictive performance using prediction error and the tree's average depth. Thus, each tree references only one feature, but may perform multiple tests to predict different future phases. To compare these trees' predictive

quality numerically, we use the loss function presented in Equation (5.1). In that equation, E is predictive error, indicating the raw quality of prediction, and D is the tree's (weighted) average depth, indicating the complexity of the constructed predictor. Thus, we prefer low error and low complexity. In the equation, α controls the relative weight of these two properties of the decision tree. We found that $\alpha = 0.01$ gives good results. This means that D is essentially a "tie-breaker" between features with nearly equal E values.

$$\text{loss} = E + \alpha \cdot D \tag{5.1}$$

We rank the features in ascending order of predictive loss. We include up to F features for some chosen F, working from lowest to highest loss. As described in Section 4.1.1, we drop the feature if it is highly correlated with a feature already in the set to make sure that each feature we use contributes significant additional information and substitute each feature in the set by a feature that is highly correlated with the feature and lowest cost.

5.1.2 Evaluation of the Decision Tree Approach

Now we introduce our approach for model evaluation.

5.1.2.1 Ground Truth

To understand the quality of results we might achieve with our prediction model, we need ground truth. However, defining phases in a way that is meaningful across all programs is challenging—how can we generate ground-truth phase numbers? First, we use the baseline definition of phases and phase transitions of Nagpurkar et al. (2006). They compute phase boundaries based on execution of loop nests. One does need to supply a parameter to their algorithm, namely the minimum phase length (MPL). An execution of a loop nest that is shorter than the MPL will not be treated as a phase. In our experiments, we use 100K branches as MPL for Python programs and 10M branches as MPL for Java and C programs since Python code is more concise and compiles into smaller executables than Java and C (Nanz and Furia, 2015).

After we have these phases and transitions, we need a way to assign phase *numbers* to each temporal phase, based on what the program is doing in that phase. If the program does something similar again, we wish to label both phases with the same phase number. Recall that we have one feature vector per certain number of instructions. We sum the features for the instruction chunks that comprise a given phase and use the L_1 norm (divide each vector element by the sum of the absolute values of all the elements) to normalize the resulting vector. We assign phase numbers by clustering these vectors, that is, we give the same phase number to two phases when their summary vectors belong to the same cluster. We use a Gaussian mixture model (GMM) (McLachlan and Peel, 2004) [2] to perform the clustering. GMM is a clustering technique that runs a probabilistic model. It can be considered as generalizing k-means clustering to incorporate information about the covariance structure of the clusters.

One further difficulty is that we do not know how many different phases each program has. The choice of the number of mixture components of the GMM must balance competing pressures. If we choose a small number of clusters, we might identify different behaviors in the program as belonging to the same phase. If we choose a large number of clusters, it might result in overfitting. In order to learn the appropriate GMM setting for each program, we run GMM multiple times with different numbers of clusters, in ascending order, and compute the log probabilities as a score. When increasing the number of clusters no longer significantly increases the score—say it increases by less than 5%—we stop the process and use the resulting number of clusters. For most of the benchmarks, the number of clusters / phases determined by this method is 3–5.

[2] We use python scikit-learn 0.19.1 package to implement GMM

5.1.2.2 Comparison Methodology

To understand the quality of our decision tree model,[3] we compare it with five other predictors. Note that in every case we may be predicting the phase k steps later, for $k = 1, \ldots$. Here are those five predictors:

Uniform: Rule: Always choose the most common phase, as determined by the ground truth for the given trace. Its prediction is just a sequence of the same phase number, over and over. The most common phase is determined according to the total number of *time steps* when execution is in that phase, i.e., it is weighted by phase length. This could also be called the *maximum likelihood* predictor. Formally, for each time step t between M and n where M is the maximum value for k, the number of steps we predict into the future, and n is the last time step in the trace, if $f_i = \sum_{t=M+1}^{n} \mathbb{1}[\phi_t = i]$ then Uniform chooses $\arg\max_i f_i$.[4]

Random: Rule: Choose the next phase according to the relative frequency of the various phases in the ground truth of the particular trace. Formally, the probability of choosing phase i is $f_i / \sum_j f_j$.

CondUniform: Rule: Choose the most likely next phase, given the phase we are in now. To compute it, we use the frequencies $f_{i,j}$ of being in phase i in the current time step and in phase j after k time steps. Formally, $f_{i,j} = \sum_{t=M+1}^{n} \mathbb{1}[\phi_t = j] \cdot \mathbb{1}[\phi_{t+k} = i]$. So, for current phase i, CondUniform chooses $\arg\max_j f_{i,j}$

CondRandom: Rule: Choose the next phase according to its probability, *given* our current phase now. Thus, the probability of predicting phase j when we are currently in phase i is $f_{i,j} / \sum_j f_{i,j}$.

ProbSim: Rule: Iterate CondRandom k times.

---

[3]We use the python scikit-learn 0.19.1 package to derive decision trees.

[4]Here $\mathbb{1}[e]$ is the *indicator* function. Its value is 1 if the logical expression e is true, and 0 if e is false.

To evaluate and compare Uniform, Random, CondUniform, CondRandom, ProbSim, and our decision model, we use an F-score based on measuring precision and recall. The scoring is based on comparing a given scheme's phase prediction against the phases as determined in the ground truth. The F-score combines precision and recall into a single number. Precision is the fraction of predicted phases that correspond to actual ones in ground truth, while recall is the average fraction of correct predictions for each phase number predicted. Given precision p and recall r, the F-score is defined as $2 \cdot \frac{p \cdot r}{p+r}$.

5.1.2.3 Cross Validation of Decision Trees

For our decision tree based prediction models, we aim for run-time use, which implies developing a model from some number of program runs—the training set—and using that model on new program runs. Since we are developing program-specific phase predictors, we use new runs of the *same program*, but on different inputs. Thus, *cross-validation* plays an important role in the learning process. We use "leave-one-out" cross-validation. If we have k runs of a given program, then each cross-validation fold trains on $k-1$ of the runs and evaluates on the run that was left out of the training.

5.1.2.4 Number of Features and Tree Depths

The number of features and tree depths we choose to use are important hyper-parameters of our run-time phase prediction system. Having more features increases the cost of feature counting, and higher tree depth increases the cost of the run-time prediction function. Having fewer features and lower tree depths, though, may decrease the quality of the predictors. Note that we expect the number of features to have much higher impact on cost than the tree depth, since the decision trees are evaluated relatively rarely, approximately once every 10,000 branches if we match our feature interval length, while a given feature must be incremented every time it is encountered at run time.

In order to choose appropriate settings for the learning model, we developed decision trees with various numbers of features and tree depths.

5.1.2.5 Comparison between Online Prediction and Offline Ground Truth

When facing the same problem as we described in Section 4.2.3, the solution to minimize the difference between window boundaries in run-time and offline program execution is choosing a single timing feature from outside of the selected feature set, such that the feature's variation from window to window across the run is small. We trigger our phase prediction function when this feature has occurred the average number of times that it occurs in a window in the original trace.

Another challenge is to compare run-time results against offline "ground truth" given the different lengths of time intervals. Here is our approach. First, we assume that each predictor's windows have uniform length in running time. (This is not quite true, but is the best we can do with the knowledge at hand.) Then, for each phase reported by B, we interpolate phases between two known points in the sequence of phases predicted by A by choosing the phase that is closest to the interpolated point and compare only the interpolated phases predicted by A to the phases reported by B.

5.2 Real-Time Program Phase Prediction in Java

5.2.1 Offline Evaluation

5.2.1.1 Number of Inputs

Table 5.1 presents the number of inputs for each benchmarks in our evaluation.

Table 5.1: Number of inputs for each benchmarks.

Benchmark	Inputs
avrora	4
batik	3
fop	3
jython	5
lusearch	3
pmd	5

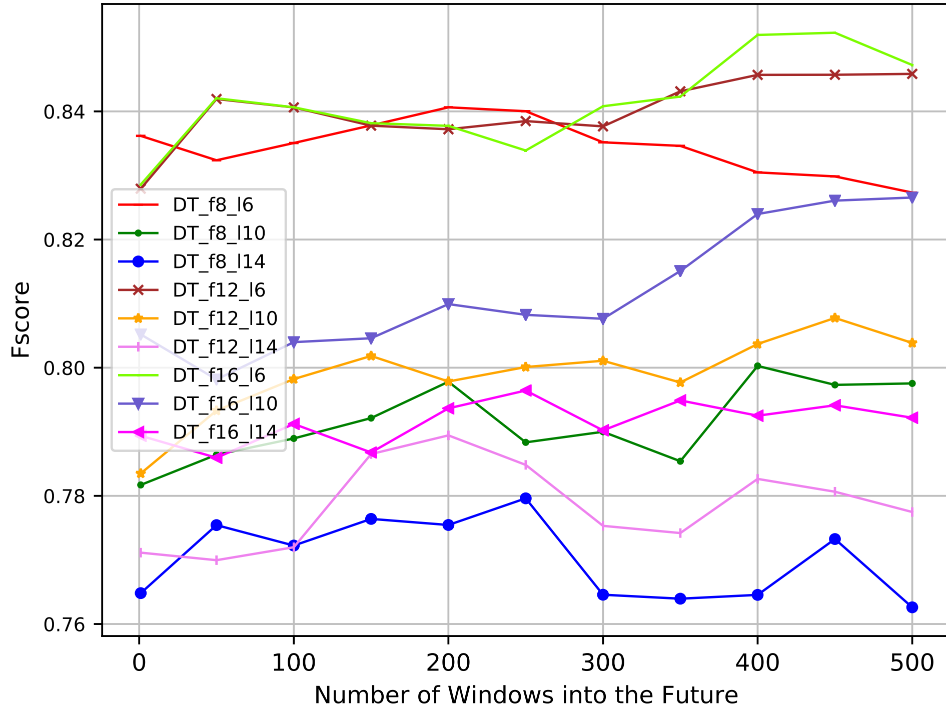


Figure 5.1: F-score for predicting the next 1 to 500 future phases for Java programs by offline cross-validated decision trees with 8, 12, and 16 selected features. The maximal depth of the decision trees is 6, 10, or 14.

5.2.1.2 Number of Features and Tree Depths

In order to choose appropriate settings for the learning model, we developed decision trees with various numbers of features and tree depths. Figures 5.1 show the prediction results. We see that in general a tree depth of at most 6 gives a higher score for predicting future phases. The result shows that increasing maximal tree depth causes decision tree model overfitting for Java programs. We also see that 8 features gives a higher score for predicting the next phase, but has lower score in predicting farther future phases, while 12 and 16 features have higher score in predicting phases 300 or more time steps in the future, but give a lower score for predicting the next phase. In our experiment, we use 8 features since it has lower run-time overhead.

5.2.1.3 Comparison of the Predictors

Figure 5.2 show the F-scores of different predictors to predict the next 1 to 500 future phases. We see that cross-validated decision trees with various numbers of features and tree depths predict future phases with high quality. Even for phases 250 time steps in the future, all of our decision tree models achieve an F-score of 84%, and still achieve 83% to 84% at $k = 500$.

In Figure 5.2, we also see that our decision trees actually perform better than Uniform, Random, and ProbSim, an average of 84% versus Uniform's 64%, Random's 55%, and ProbSim's 71% for predicting the next phase. Even after the next step, our model clearly outperforms these models for almost all values of k. CondUniform's and CondRandom's quality are better than our model for predicting phases of Java programs, but these predictors need correct *current phase* information, which is not guaranteed at run time. Nevertheless, the gap between these models and our model gives us space to improve in future work.

5.2.2 Implementing the Real-Time Predictor for Java

We implemented the phase predictor as a Java agent, using the Java Virtual Machine Tool Interface (JVMTI).[5] Each instrumented feature calls a Java method to increment a counter for that feature. It also counts down a total number of events. Once the desired number of events is reached, the Java counting method calls a native method in the agent, which obtains the counter values and then, using trained decision tree it predicts future phases. The code also counts a feature that has low variance across the traces, in order to trigger periodic invocation of a function that implements the predictor's decision tree. To insert the instrumentation into the Java code, we use a modified version of Elephant Tracks that instruments only the selected features. It does this by applying bytecode rewriting to the compiled classes of the application. We perform these rewrites as the code loads. The modified version of Elephant Tracks also automatically generates the Java class holding the

[5] We use ibm-java-x86_64-60 Java virtual machine

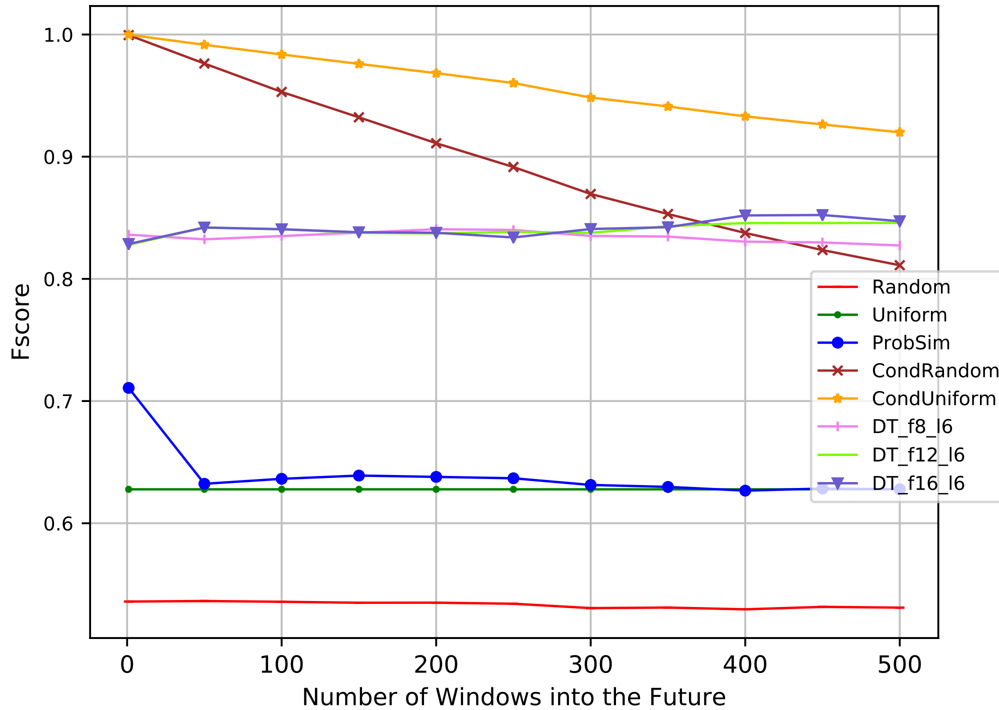


Figure 5.2: F-score for predicting the next 1 to 500 future phases for Java programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 6.

event counters and counter increment methods, which the Java agent then loads into the JVM.

5.2.3 Results

We now consider the performance impact of our run-time predictor, and its prediction quality at run time.

5.2.3.1 The run-time overhead of our system

We compare the cost of running with and without our run-time system. For each benchmark and input, we performed multiple runs of the program with and without treatment. We did runs with 8 features, to assess the acceptability of the overhead of our system. Table 5.2 shows that 8 features gives acceptable quality with only 3% overhead on average.

Table 5.2: Ratio of running time of programs with 8 features treatment to the original benchmark programs for Java.

Benchmark	Inst 8
avrora	1.038
batik	1.007
fop	1.000
jython	1.005
lusearch	1.133
pmd	1.004
Average	1.030

5.2.3.2 Quality of the Run-time Predictor

By using the same number of time intervals for each predictor as described in Section 5.1.2.5 to compare against "ground truth", we obtain the F-scores shown in Figure 5.3. We see that our decision trees are actually more accurate than most of the other models, an average of 78% versus Uniform's 64%, Random's 55%, and ProbSim's 71% for predicting the next phase. Our model also outperforms these models for all values of k. Even though CondUniform and CondRandom can achieve better prediction results (Both score is 99% for predicting next phase), this predictor needs correct *current phase* information, which is not guaranteed at run time.

5.3 Real-Time Program Phase Prediction in C

5.3.1 Offline Evaluation

5.3.1.1 Number of Inputs

Table 5.3 presents the number of inputs for each benchmark in our evaluation.

5.3.1.2 Number of Features and Tree Depths

We developed decision trees with various numbers of features and tree depths for choosing appropriate settings for the learning model. Figures 5.4 show the prediction results. We see that 16 features gives a higher score for predicting the next phase, but has lower score in predicting farther future phases, while 8 features gives general good prediction

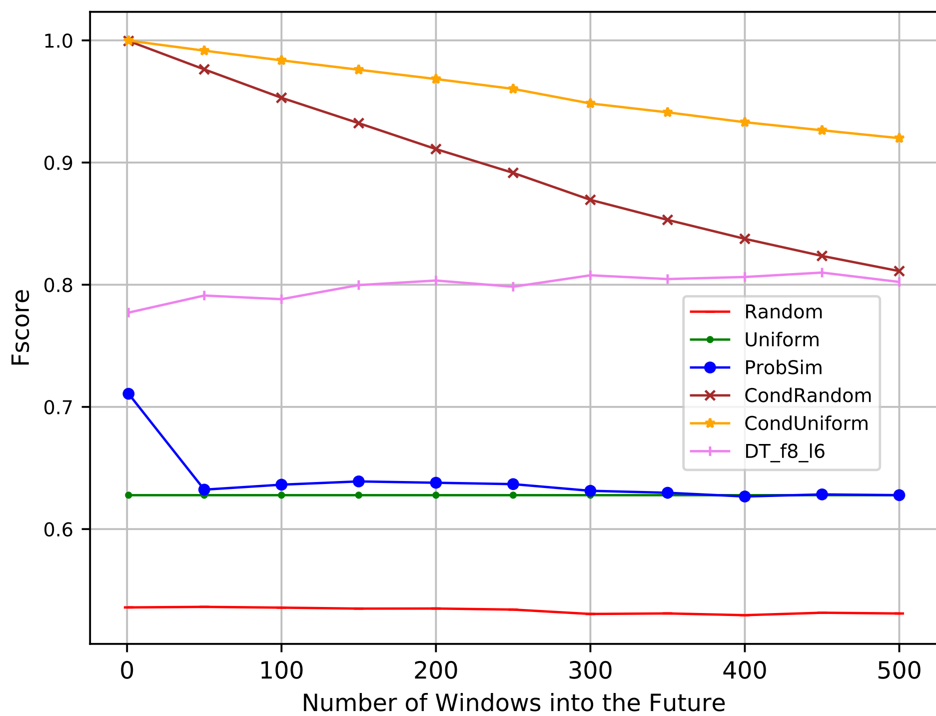


Figure 5.3: F-Score for predicting the next 1 to 500 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for Java programs. We use 8 features and maximal tree depth of 10 in our decision tree model.

Table 5.3: Number of inputs for each benchmark.

Benchmark	Inputs
astar	3
bzip2	8
cactusADM	3
gcc	4
mcf	2
milc	2

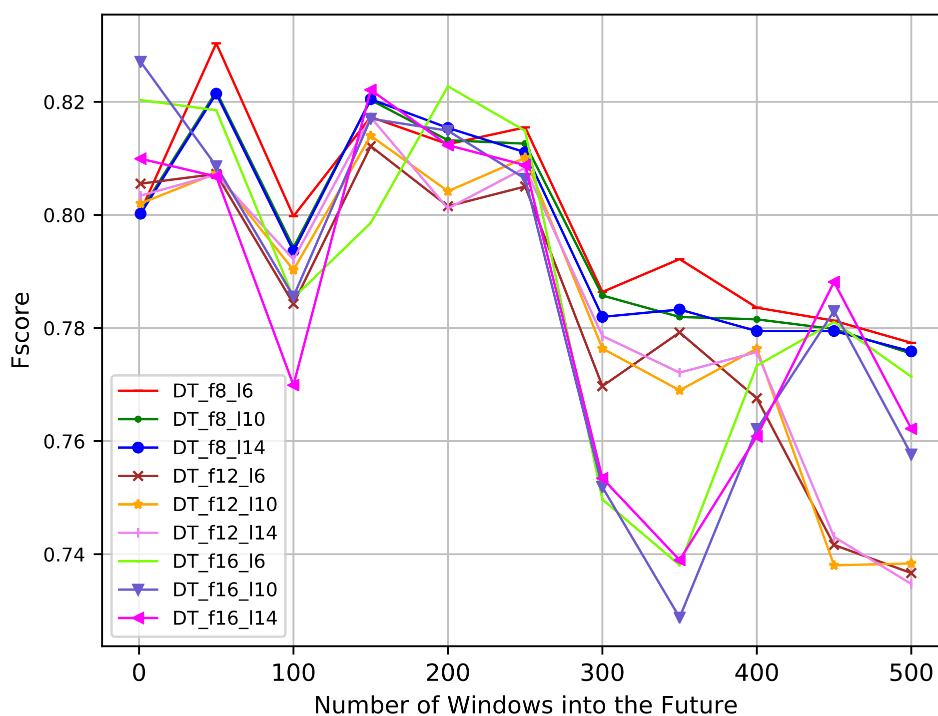


Figure 5.4: F-score for predicting the next 1 to 500 future phases for C programs by offline cross-validated decision trees with 8, 12, and 16 selected features. The maximal depth of the decision trees is 6, 10, or 14.

results. The results show using more than 8 features causes our decision tree model to overfit for C programs. Also, a tree depth of at most 6 is enough to obtain good prediction quality. This shows that it is possible to reduce from thousands of features to just a few bits of information in order to predict future phases.

5.3.1.3 Comparison of the Predictors

Figure 5.5 show the F-scores of different predictors to predict the next 1 to 500 future phases. We see that cross-validated decision trees with various numbers of features and tree depths predict future phases with high quality. Even for phases 250 time steps in the future, all of our decision tree models achieve an F-score of 80% to 82%, and still achieve 74% to 78% at $k = 500$.

In Figure 5.5, we also see that our decision trees actually perform better than most of the other models, an average of 82% versus Uniform's 72%, Random's 39%, CondUniform's 97%, CondRandom's 99%, and ProbSim's 52% for predicting the next phase. We know that adjacent time steps share the same phase number with high probability, so it is not surprising how well CondUniform and CondRandom do for $k = 1$, even better than our decision trees. However, as we increase k, CondRandom's accuracy decreases dramatically, first to 70% and then quickly to 68% at $k = 50$ and $k = 100$. After step $k = 50$, our model achieves results similar to CondUniform, and clearly outperforms all the other models for almost all values of k.

5.3.2 Implementing the Real-Time Predictor for C

To instrument C programs (compiled binaries, actually), we implemented a PEBIL tool (Laurenzano et al., 2010), which does this by using function level code relocation. Each instrumented feature increments a counter for that feature. The code also counts a feature that has low variance across the traces, in order to trigger periodic invocation by which the program calls a function that obtains the counter values and then, using a trained decision tree, determines the phase for the future time interval.

5.3.3 Results

We now consider the performance impact of our run-time predictor, and its prediction quality at run time.

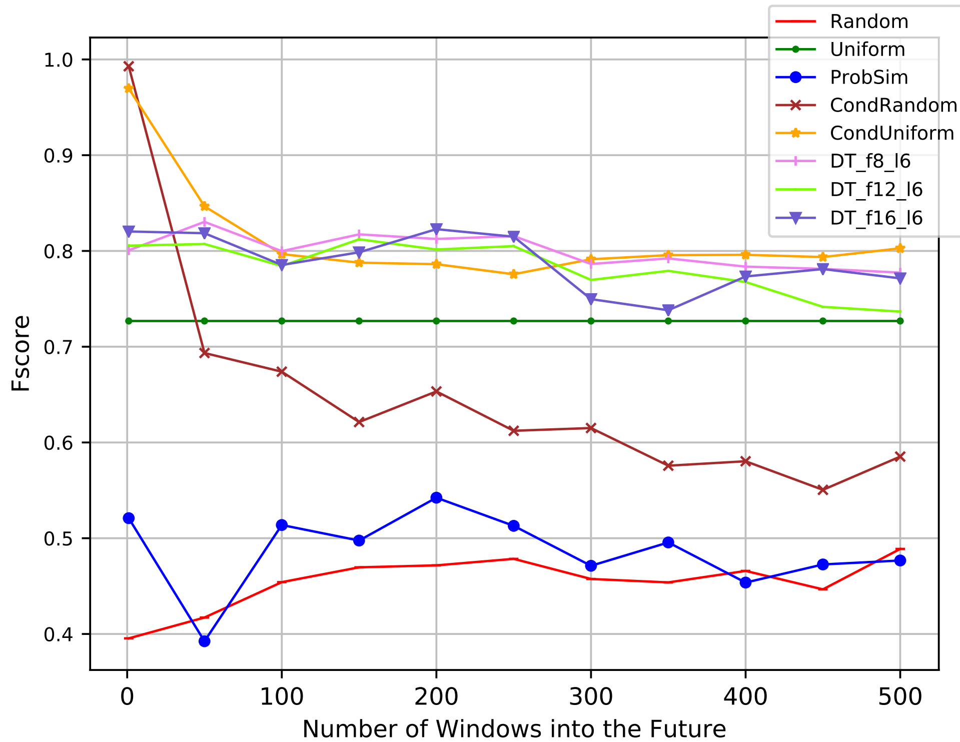


Figure 5.5: F-score for predicting the next 1 to 500 future phases for C programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 6.

5.3.3.1 The run-time overhead of our system

We compare the cost of running with and without our run-time system. For each benchmark and input, we performed multiple runs of the program with and without treatment. We did runs with 8 features, to assess the acceptability of the overhead of our system. Table 5.4 shows that 8 features gives acceptable quality with only 1.7% overhead on average.

5.3.3.2 Quality of the Run-time Predictor

By using the same number of time intervals for each predictor as described in Section 5.1.2.5 to compare against "ground truth", we obtain the F-scores shown in Figure 5.6. We see that our decision trees are actually more accurate than most of the other models, an average of 78% versus Uniform's 72%, Random's 39%, and ProbSim's 52% for predicting

Table 5.4: Ratio of running time of programs with 8 features treatment to the original benchmark programs for C.

Benchmark	Inst 8
astar	1.012
bzip2	1.015
cactusADM	1.051
gcc	1.013
mcf	1.009
milc	1.003
Average	1.017

the next phase. After $k = 50$, our model clearly outperforms most of the other models for almost all values of k. In particular, our model can achieve 80% accuracy for predicting the phase 200 time steps in the future. Going even as far as $k = 300$, our decision trees achieve 79%, and still achieve 76% at $k = 500$. CondUniform, which needs correct *current phase* information, can achieve better prediction results (99%) for predicting next phase, but our model performs similarly to it after $k = 150$.

5.4 Real-Time Program Phase Prediction in Python

5.4.1 Offline Evaluation

5.4.1.1 Number of Inputs

Table 5.5 presents the number of inputs for each benchmarks in our evaluation.

5.4.1.2 Number of Features and Tree Depths

In order to choose appropriate settings for the learning model, we developed decision trees with various numbers of features and tree depths. Figures 5.7 and 5.8 show the prediction results. We see that 8 features gives a fairly high score, and going beyond 16 features does not give much additional improvement. Also, in all cases with 8 to 16 features, a tree depth of at most 12 is enough to obtain good prediction quality.

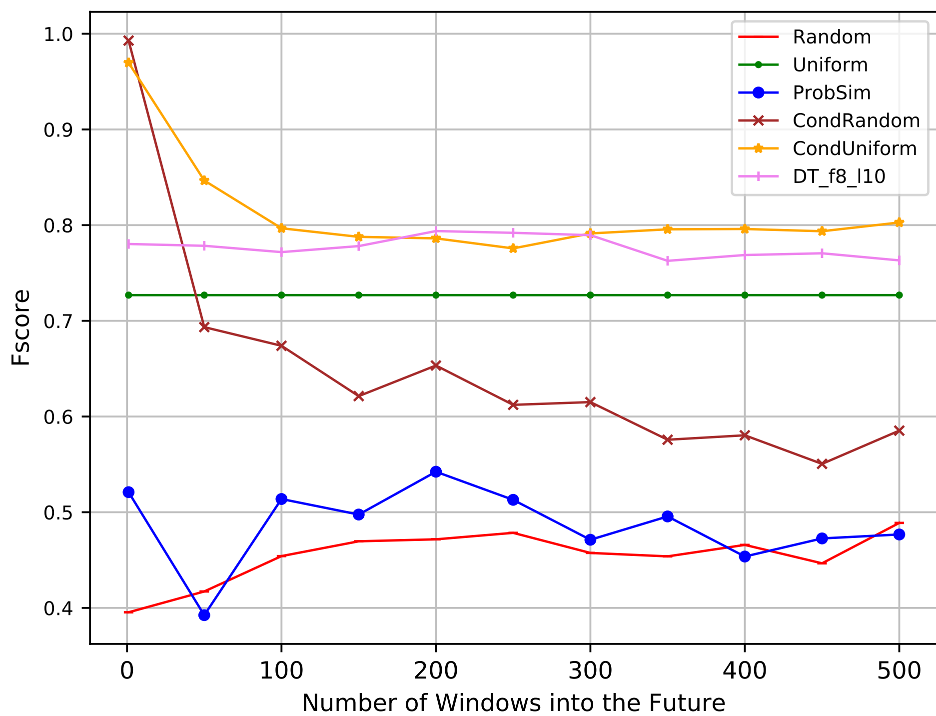


Figure 5.6: F-Score for predicting the next 1 to 500 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for C programs. We use 8 features and maximal tree depth of 10 in our decision tree model.

Table 5.5: Number of inputs for each benchmarks.

Benchmark	Inputs
chameleon	4
chaos	16
django_template	4
genshi	3
mako	3
meteor_contest	3
pathlib	4
spambayes	8
sqlalchemy_declarative	3
sqlalchemy_imperative	3
sympy	4
tornado_http	3
xml_etree	4

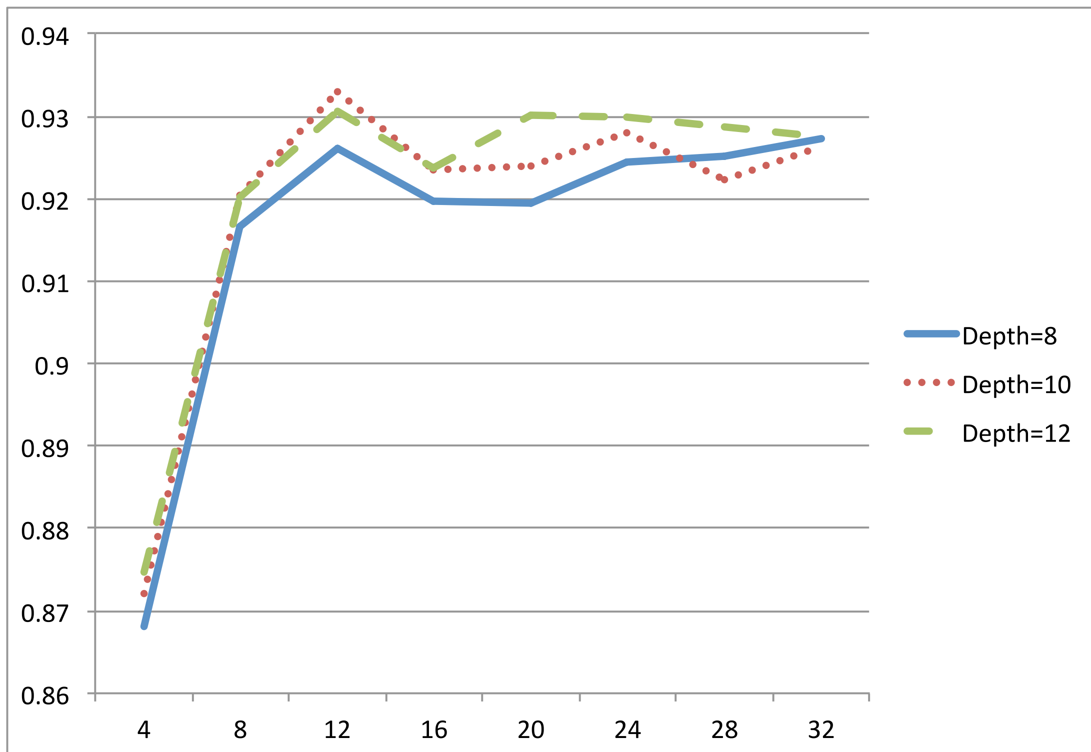


Figure 5.7: F-score for predicting the next phase for Python programs by offline cross-validated decision trees with 4, 8, 12, 16, 20, 24 and 32 selected features. The maximal depth of the decision trees is 8, 10, or 12.

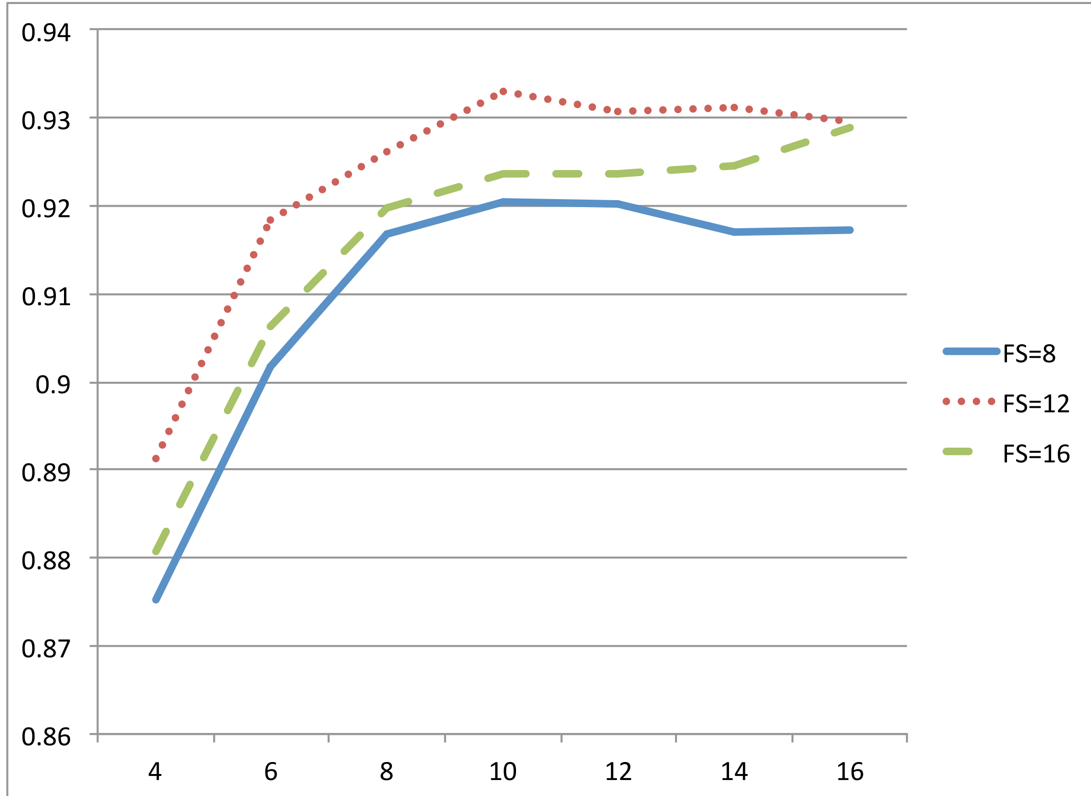


Figure 5.8: F-score for predicting the next phase for Python programs by offline cross-validated decision trees with maximal tree depths of 4, 6, 8, 10, 12, 14, and 16. The number of features is 8, 12, or 16.

5.4.1.3 Comparison of the Predictors

Figures 5.9 and 5.10 show the F-scores of different predictors to predict the next 1 to 49 future phases. In Figure 5.9, we see that cross-validated decision trees with various numbers of features and tree depths predict future phases with high quality. Even for phases 30 time steps in the future, all of our decision tree models achieve an F-score of 87% to 91%. Considering Figure 5.10, we see that our decision trees actually perform better than most of the other models, an average of 93% versus Uniform's 68%, Random's 43%, CondUniform's 69%, CondRandom's 91%, and ProbSim's 64% for predicting the next phase. We know that adjacent time steps share the same phase number with high probability, so it is not surprising how well CondRandom does for $k = 1$, similar to our decision trees. However, as we increase k, CondRandom's accuracy decreases dramatically, first to 84% and then

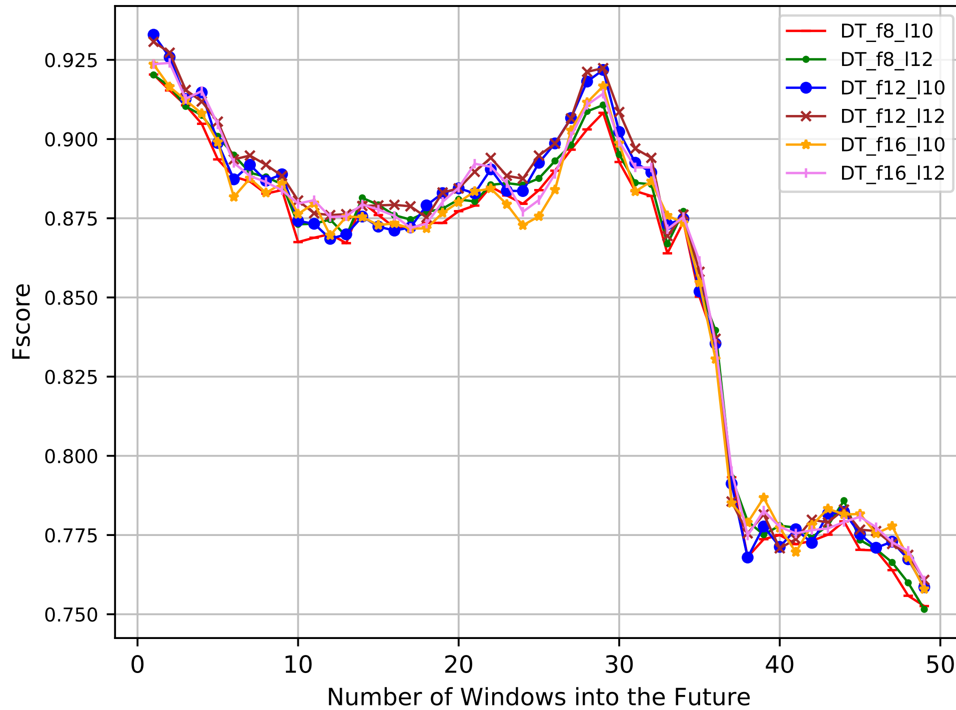


Figure 5.9: F-score for predicting the next 1 to 49 future phases for Python programs by offline cross-validated decision trees with 8, 12, and 16 selected features, and maximal tree depth of 10 or 12.

quickly to 78%. However, going even as far as $k = 30$, our decisions trees achieve 90%, and still achieve 76% at $k = 49$. They clearly outperform all the other models for all values of k.

5.4.1.4 How Far in the Future Can Our Model Predict Phases?

Looking again at Figure 5.10, we see that the F-score of the decision tree model gradually decreases from 93% at one phase later to 89% at six steps later, and it is relatively stable until 30 time steps out. After that, the decision tree model's predictive ability decreases dramatically, to only 76% at 49 time steps out. It is not surprising that prediction is more difficult farther into the future, but it is perhaps surprising how well we can predict 30 steps in the future, especially given that the minimum phase length (MPL) is 10. This suggests that many phases are longer than that. In the 62 traces with length from 58 to 590, Figure 5.11

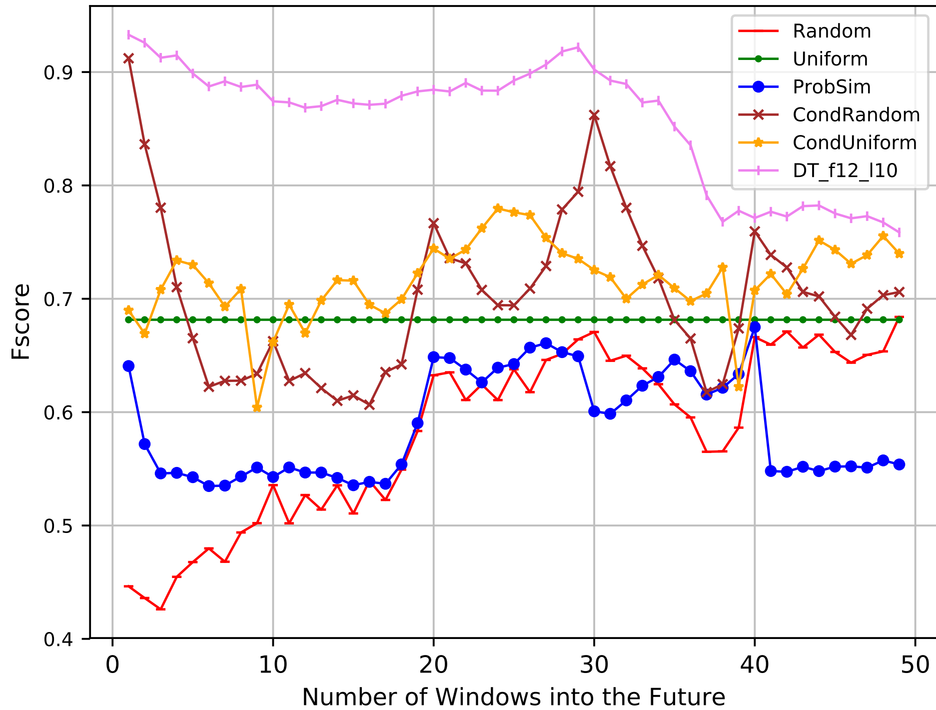


Figure 5.10: F-Score for predicting the next 1 to 49 phases for Python programs by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim. We use 12 features and maximal tree depth of 10 in our decision tree model.

shows the CDF of length of the phases in these traces. These results are averaged over all inputs of all the benchmarks we used. We see that many phases are quite long, which help explain why our models perform so well in predicting relatively far into the future.

5.4.2 Implementing the Real-Time Predictor for Python

We implemented the phase predictor inside the CPython implementation—similar to the trace module of the Python[6] standard library. Each instrumented feature is a key into a hash table of counts, with the count being incremented when that feature (branch) occurs. The code also counts a feature that has low variance across the traces, in order to trigger periodic invocation of a function that implements the predictor's decision tree. We record the current

[6]We use Python 2.7

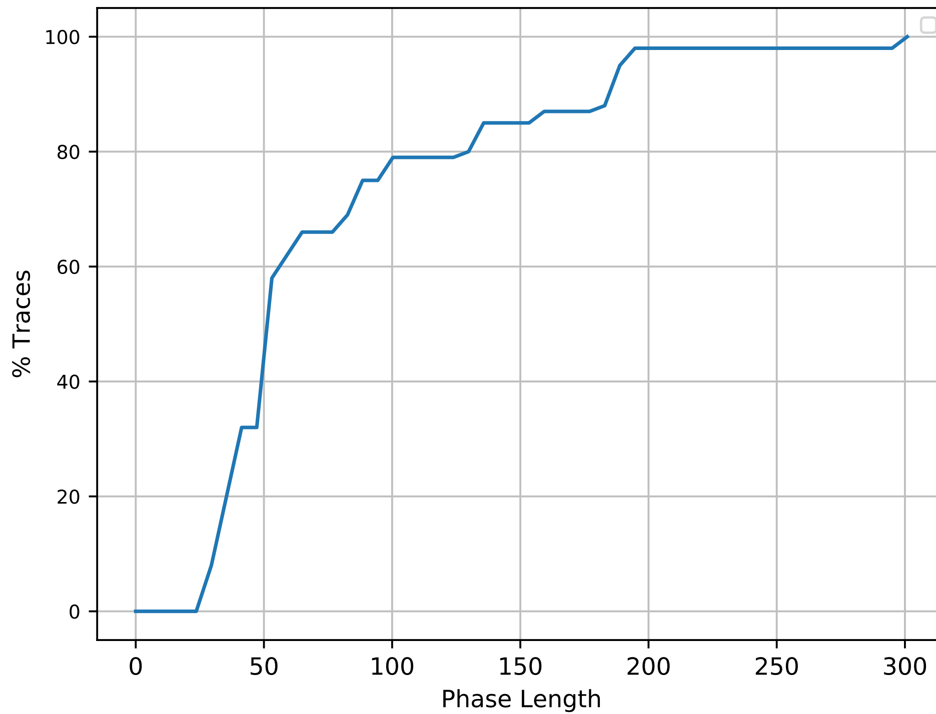


Figure 5.11: CDF of length of phases in traces for Python programs.

phase and the predicted future phase in a memory buffer that we dump out at the end of a program run.

5.4.3 Results

We now consider the performance impact of our run-time predictor, and its prediction quality at run time.

5.4.3.1 The run-time overhead of our system

We compare the cost of running with and without our run-time system. For each benchmark and input, we performed multiple runs of the program with and without treatment. We did runs with 8, 12, and 16 features, to assess the acceptability of the overhead of our system.

We find, as one would expect, that the run-time overhead tends to increase as we instrument more features. The smallest number of features that gives good detection quality is 12, resulting in average[7] overhead of 0.7% (see Table 5.6), while 8 features gives acceptable quality with only 0.5% overhead on average. Using 16 features increases the overhead to 0.8%.

Table 5.6: Ratio of running time of programs with 8, 12, and 16 features treatment to the original benchmark programs.

Benchmark	RTFs 8	RTFs 12	RTFs 16
bm_chameleon	1.010	1.013	1.014
bm_chaos	1.005	1.006	1.008
bm_django_template	1.005	1.007	1.010
bm_genshi	1.004	1.005	1.006
bm_mako	1.005	1.006	1.006
bm_meteor_contest	1.005	1.008	1.008
bm_pathlib	1.007	1.008	1.012
bm_spambayes	1.004	1.005	1.006
bm_sqlalchemy_declarative	1.002	1.004	1.005
bm_sqlalchemy_imperative	1.007	1.011	1.011
bm_sympy	1.013	1.013	1.014
bm_tornado_http	1.002	1.005	1.005
bm_xml_etree	1.003	1.004	1.005
Average	1.005	1.007	1.008

Figure 5.12 shows the distribution of relative running times for each program, showing the original program (baseline) and treatment with various numbers of features. To plot the data, for each trace we determined the median running time for the baseline since the median gives a better estimate of typicality than the mean does, because the mean is more perturbed by outliers. This is the value to which the other data are normalized. (This applies to Table 5.6 as well.) Thus the baseline shows its relative median as being 1. For programs running in our system, the distribution includes multiple runs under each of the feature selections. The distributions show that they are mostly not dispersed. Returning to Table 5.6,

[7] Our averages are geometric means. However, arithmetic means are quite similar.

in addition to observing that the average overhead for 12 features is less than 1%, we see that only 11 of 62 traces incur overhead of 1% or more.

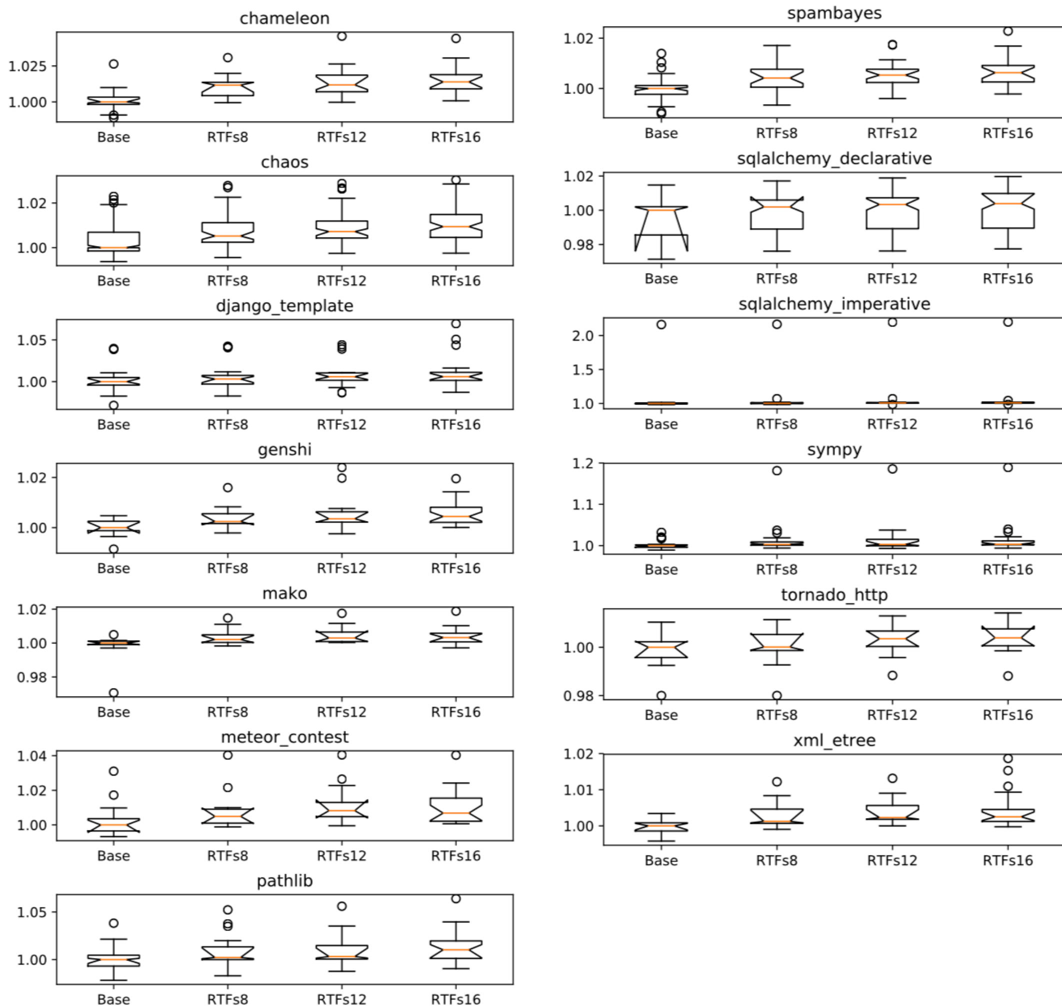


Figure 5.12: Execution time distributions of baseline and treatment with various numbers of features for Python programs.

To test the statistical significance of the difference in running times between baseline and treatments, we performed T-tests comparing the time distributions. The 8 feature case shows quite weak significance ($p = 0.21$ means that there is a 21% chance that the two distributions were drawn from the same underlying distribution). With higher numbers of features the significance is higher. We conclude that the effect is real, not just chance in our measurements, but that the effect is small.

5.4.3.2 Quality of the Run-time Predictor

In Figure 5.13, we see that cross-validated decision trees with various numbers of features and tree depths predict future phases with high quality. Even for phases 30 time steps in the future, all of our decision tree models achieve an F-score of 88% to 90%. Considering Figure 5.14, we see that our decision trees are actually more accurate than most of the other models, an average of 84% versus Uniform's 57%, Random's 43%, and ProbSim's 64% for predicting the next phase. After $k = 5$, our model clearly outperforms all the other models for all values of k. In particular, our model can achieve 91% accuracy for predicting the phase 20 time steps in the future while the best score the other models achieve is only 77%. Going even as far as $k = 30$, our decision trees achieve 88%, and still achieve 79% at $k = 49$. They clearly outperform all the other models for almost all values of k. Even though CondUniform and CondRandom can achieve similar quality, and occasionally better prediction results (CondUniform's 91% for predicting next phase), these predictors need correct *current phase* information, which is not guaranteed at run time.

5.5 Summary

In general, after applying a training protocol to a program of interest, our method can predict the future phase k steps later, for $k = 1, \ldots$, at run time, for that program with good precision and recall and without significant run-time overhead for all 3 programming languages.

Our methodology achieves the highest F-score for the Python programs, with 88% to 90% for predicting future phases in the next 30 steps. It also outperforms in general all of the other models, including CondUniform and CondRandom, which need correct current phase information, which is not guaranteed at run time. For the C programs, our methodology has lower F-score on average compared to the model's performance for Java and Python programs. However, its predictive performance is relatively stable, with 79% at $k = 1$ and 76% at $k = 500$, and achieves F-score similar to CondUniform after $k = 150$. Our decision

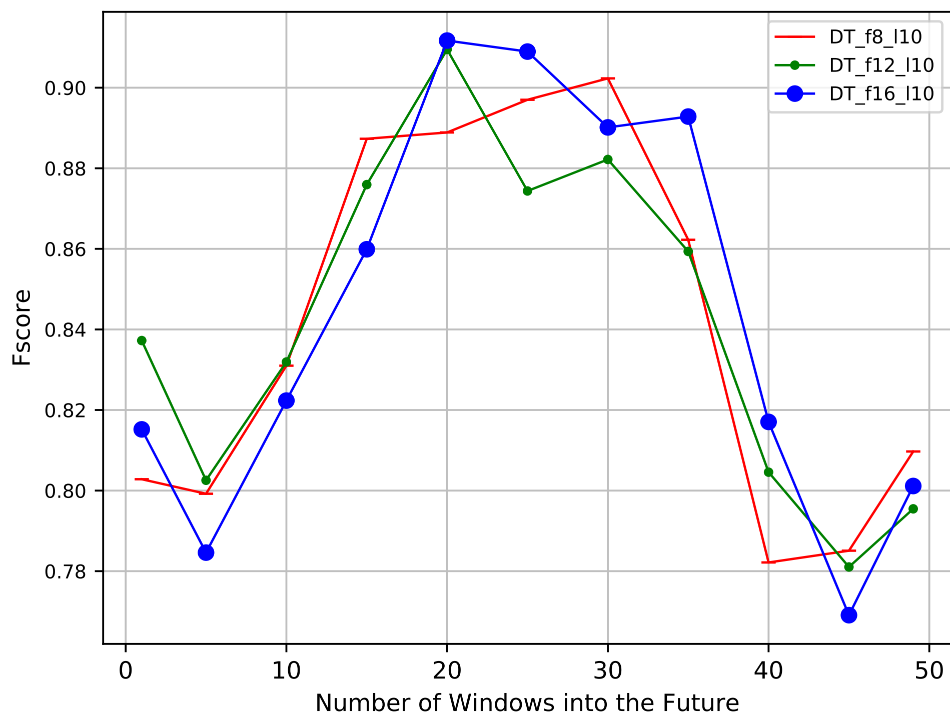


Figure 5.13: F-score for predicting the next 1 to 49 future phases by run-time cross-validated decision trees with 8, 12, and 16 selected features for Python programs. The maximal tree depth is 10.

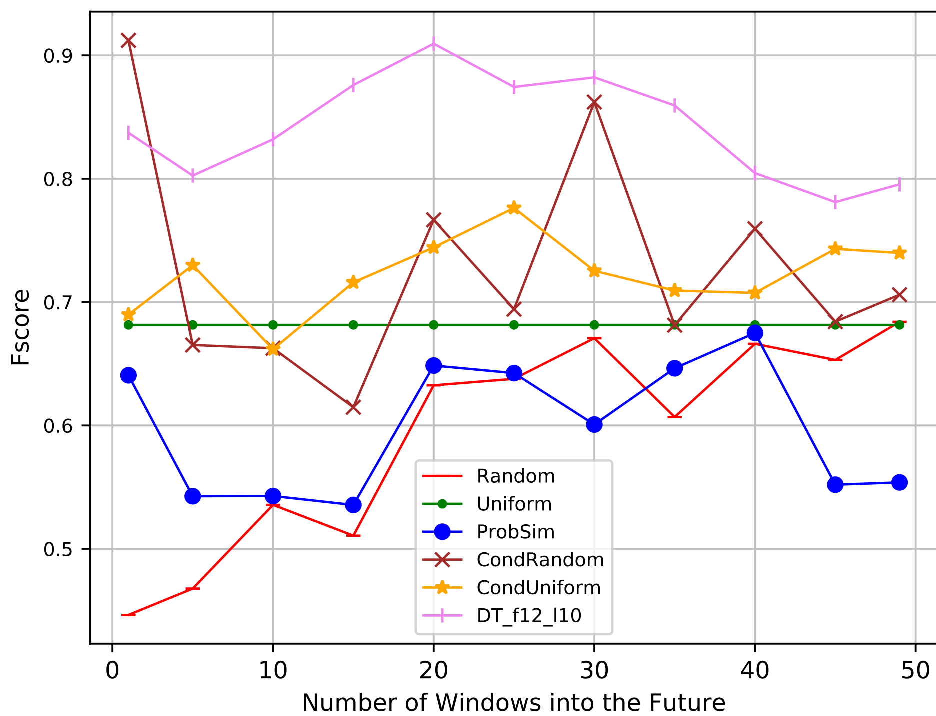


Figure 5.14: F-Score for predicting the next 1 to 49 phases by decision trees, Uniform, Random, CondUniform, CondRandom, and ProbSim for Python programs. We use 12 features and maximal tree depth of 10 in our decision tree model.

trees achieve 78% F-score in next phase prediction for the Java programs, but Figure 5.3 shows there is a significant gap between the performance of our models and of CondUniform. Even though our decision trees ourperform all other models that use only offline information or the information guaranteed to be available at run time, we see the possibility to further improve our models to fill the gap, and we will explore it in future work.

CHAPTER 6

FROM PROGRAM PHASE TO SYSTEM BEHAVIOR PREDICTION

In Chapters 3 and 4, we introduced methods to detect and predict program phases for different program languages. However, what do these phases mean to the system? To answer this question, we use real-time phase prediction to help save energy. In this study we consider only statically compiled executables.

6.1 Introduction

Dynamic voltage and frequency scaling (DVFS) has proven to be an effective method of achieving lower power consumption while simultaneously meeting performance requirements (Horowitz et al., 1994). The key idea behind DVFS is to change the CPU voltage and/or clock frequency in real-time (Choi et al., 2005; Hotta et al., 2006; Azevedo et al., 2002; Shin and Kim, 2001; Grunwald et al., 2000; Herbert and Marculescu, 2009; Dhiman and Rosing, 2007; Rangan et al., 2009; Bhattacharjee and Martonosi, 2009; Xie et al., 2003; Wu et al., 2005b; Li et al., 2005; Wu et al., 2004; Magklis et al., 2006; Wu et al., 2005a) according to the system workload, and thereby reduce energy consumption. Phase information can indicate whether lowering clock frequency might save energy with little impact on performance.

We now present a method for real-time dynamic clock frequency adjustment in statically compiled executable programs, i.e., C/C++, Fortran, etc. After applying a training protocol to a program of interest, our method can adjust clock frequency at run time for that program to lower the energy-delay product (ED) with small performance impact. To accomplish this,

we first trace a number of executions of the program of interest, recording information about each call/return and conditional branch that occurs.

We use the traces described in Chapter 3 to develop a real-time energy-efficient clock frequency adjuster, as follows. We choose (in a manner described later) a small number of branch instructions that we will instrument at run time. Next, we apply a two stage learning model. The first stage determines which set of clock frequencies, among those available, works best for a given trace. The second stage of learning develops a *decision tree* for adjusting the clock frequency, if more than one frequency is advised. At run time, we increment a counter each time we execute an instrumented branch, and we set a timer to trigger evaluation of the decision tree to determine the best clock frequency for the next interval. The timer is set for a certain number of microseconds, depending on the running time of program and the length of the traces, aiming for a certain average number of branches per interval (typically 100,000).

6.2 A Machine Learning Model for ED Reduction

We now turn to using machine learning methods to build a model to control adjusting clock frequency, with the objective of reducing ED. We first consider how to estimate ED itself, arguing that we can do so from measurements of cache misses. We estimate cache misses as part of running the Branchgrind tool, previously described. The second consideration we address is how to choose a small, but effective, subset of branches (features) to instrument at run time. From those features we build decision trees. This section concludes with a theoretical (i.e., offline) evaluation of the performance of the decision trees.

6.2.1 Estimating ED Product from Cache Misses

Ideally we would like to record the ED product for each interval of 100,000 branches found by our tracing tool. This is difficult to do directly because tracing introduces large overhead. Instead, we designed an estimation method to compute the ED product from

cache misses in program runs. The idea is that the time to execute a piece of code can reasonably be estimated as consisting of C CPU cycles plus M memory wait time, where M is the number of cache misses times the memory wait time.[1] The time to run the program at a given frequency f is thus:

$$t_f = M + \frac{C}{f} = \frac{f \cdot M}{f} + \frac{C}{f} \propto \frac{f + \frac{C}{M}}{f} \tag{6.1}$$

But we need a way to relate energy, E, to f.

First, power P is related to the voltage V and clock frequency f according to Equation (6.2):

$$P \propto V^2 f \tag{6.2}$$

However, voltage and frequency have a complex relationship on modern processors. It has been argued (Sakurai and Newton, 1990) that the relationship is well modeled as:

$$f \propto \frac{(V - V_{th})^{\alpha}}{V} \tag{6.3}$$

In current technologies (CMOS at 22 nm and below), we believe that V_{th} is about 0.4v and α is about 1.4. For V in the range V_{th} to 1.0v, f is well approximated as being proportional to $V - V_{th}$, thus:

$$f \propto k(V - V_{th}) \tag{6.4}$$

where k is a proportionality constant, which we have estimated as being approximately 0.8 for a good linear fit. We find it convenient to express V in terms of f, giving:

$$V = \frac{f}{k} + V_{th} \tag{6.5}$$

[1] Admittedly this simple model neglects factors such as overlap of CPU and memory activity, etc., but we claim it is good enough for choosing between a handful of different available CPU clock frequencies.

Energy E is power times time, and delay D is time, leading to Equation (6.6):

$$ED = E \times D = P \times t \times t = Pt_f^2 \tag{6.6}$$

Here the total running time at frequency f is t_f. Recalling that power is proportional to $V^2 f$, our final ED equation is:

$$ED \propto V^2 f t_f^2 = (1.25f + 0.4)^2 f t_f^2 \tag{6.7}$$

Applying Equation (6.1), we get Equation (6.8):

$$\begin{aligned} ED &\propto (1.25f + 0.4)^2 f \times (\frac{f + \frac{M}{C}}{f})^2 \\ &= (1.25f + 0.4)^2 (f + \frac{M}{C})^2 / f \end{aligned} \tag{6.8}$$

It is easy to compute memory time M given a system configuration and the number of cache misses under that configuration as determined by our tracing tool. In addition, by minimizing Equation (6.8), we can get the optimal ED if we can dynamically adjust CPU frequency to an arbitrary value. However, actual systems permit only a certain set of clock frequency multipliers, so we will try to adjust to the best frequency among those available.

6.2.2 Feature Selection

In order to predict and adjust clock frequency for reducing ED, we need to select features that are correlated with the clock frequency that has the lowest ED. To obtain an initial quality ranking for each feature in optimal clock-frequency-based selection, we use a standard train/validate/test protocol (Hastie et al., 2009) to determine the predictive value of individual features on the optimal clock frequency prediction task. We train a single-variable linear regression model for each feature using ridge regression, and assess its predictive performance using the average absolute clock frequency value prediction error

on the validation set. We then rank the features in descending order of predictive accuracy. We include up to F features for some chosen F, working from highest to lowest prediction accuracy. However, we use a correlation threshold to avoid choosing any feature whose correlation with any previously selected feature exceeds the threshold. Furthermore, we exclude features with cost higher than a chosen counting threshold, to avoid high run-time overhead in the final system. (We found that using features whose events occurred frequently resulted in high overhead, so this thresholding is necessary in order to contain run-time cost.) The counting threshold is M/F where M is a chosen total count limit and F is the number of selected features. Our experiments show that the system running time overhead is about 1% when we set M to 100,000,000 for our traces that have 20,000 to 25,000 time steps. After choosing features using the ranking-and-selection process just described, for each chosen feature we consider replacing it with a lower cost and highly correlated proxy feature.

6.2.2.1 History information

In a running program, history from prior time intervals can provide rich information. Considering that our model focuses only on the current and immediate future states of the system, we add history information to the feature space, which does not break the structure of our model. For example, if we wish to add history information for the three most recent time steps, then before running the machine learning model, our system automatically concatenates the feature vector in each time step with the feature vectors of the two previous time steps, thus generating a new feature vector that includes three times the number of features. After we include history information in our feature space, we adjust the feature selection methodology by applying linear regression to the predictive performance model to score the features using their history in each time step.

6.2.3 Offline Evaluation of the Model

We chose decision trees (DTs) [2] as the model to learn for this prediction problem. DTs are efficient to evaluate at run time, the learned models have greater intelligibility to humans than many other models, and there are many convenient tools for developing them. Thus they seemed appropriate to try, and since they ended up performing well there was no need to explore other models, such as neural nets.

To understand the quality of results we might achieve with DTs, we compared them with just using the maximal clock frequency (which is 2.6GHz in our system), hereafter called Default. We also compared with a linear regression model based on the number of cache misses, hereafter called LRCM, and with a linear regression model based on our selected features, hereafter LR. We developed a baseline ideal performance model, described just below, and we rate other models according to their degradation compared with that model.

The baseline model uses the optimal ED that can be achieved in each time interval using the set of available clock frequencies. We use the ED estimation method previously described in Section 6.2.1. Notice that the baseline model uses only the *available* clock frequencies, so it is in principle achievable. Indeed, for all of the traces we excluded from the experiment, Default matched the baseline model, so adjusting the clock frequency cannot offer any improvement for them.

6.2.3.1 ED Improvement Using a Fixed CPU Frequency

Our learning model is separated into two stages. The first, which we call clock-frequency-setting learning, works by finding the best combination of *k* clock frequencies for each trace, for selected values of *k*. Thus we find the best single frequency one could use, the best pair one could use if allowing the system to switch between two frequencies, etc. The second stage, which we call dynamic-adjustment learning, learns a model for dynamically adjusting

[2] We use python scikit-learn 0.18.0 package to implement decision tree

among the clock frequencies selected in the clock-frequency-setting stage. We use DTs in this second stage.

Based on the approach described in Section 6.2.1, we evaluate the ED of the program at various combinations of clock frequency settings. We find the best setting for each trace and number of available clock frequencies (k) pair in the first stage of learning. Figure 6.1 shows the average ED improvement of different numbers of available clock frequencies. The improvement decreases substantially beyond two frequencies: it is 42.4% average improvement (compared with Default) if we adjust to the best single clock frequency at the start of program, but only 5.0% average additional improvement for two clock frequencies (dynamic adjustment). Thus many traces do not benefit from dynamic adjustment. However, our first stage of learning remains useful because it will (a) indicate which traces may benefit from dynamic adjustment, and (b) for those where a single frequency works well, it will indicate that frequency. In our results there are seven traces (two from `bzip`, one from `h264ref`, and four from `gcc`) for which dynamic adjustment may be useful. They offer 12.7% possible average ED improvement, as shown in Figure 6.1.

6.2.3.2 ED Improvement using Decision Trees

In the clock-frequency-setting stage of learning, we improved 29 of 36 traces simply by learning a suitable single clock frequency for each trace, leaving seven traces where dynamic adjustment might offer further useful improvement. As previously mentioned, we use decision trees as our learned model for this stage. Again, our reasons are: (1) DTs, a non-linear model, gave better results than either linear regression model; (2) DTs are a white-box model, that is, that are more interpretable by humans than other non-linear models; and (3) DTs are relatively inexpensive to evaluate. To keep code/table size small and model evaluation cheap, we limited DT depth to 10.

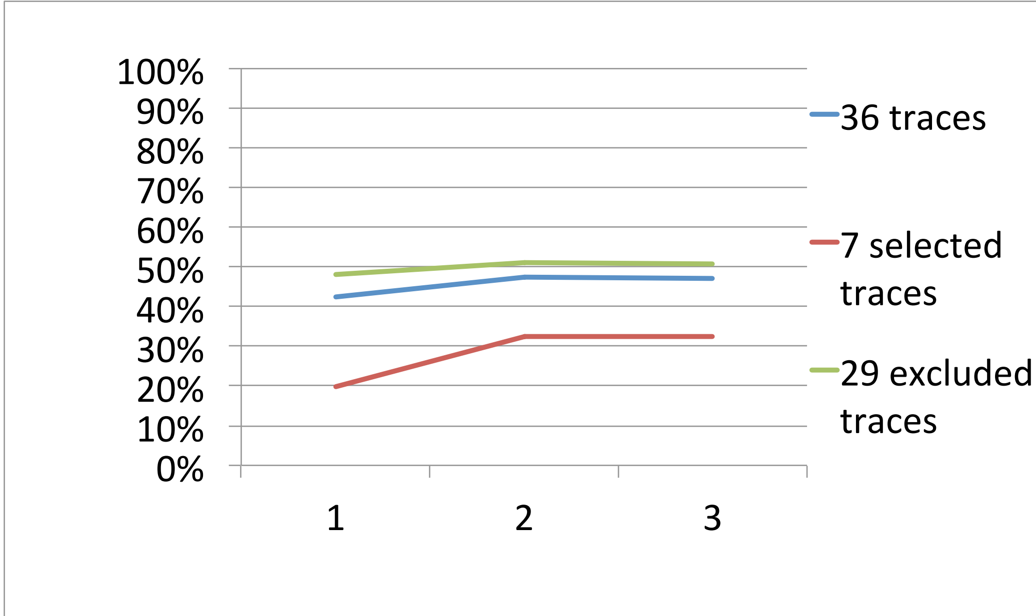


Figure 6.1: Average ED improvement using different numbers of available clock frequencies, excluding traces with less than 0.1% improvement.

6.2.3.3 Offline Cross Validation of Models

The SPEC 2006 benchmark suite (Henning, 2006) includes programs spanning a wide range of behavior and that run long enough to be interesting. They thus provide significant traces for evaluating the models in our experiment. However, real-time use implies developing a model from some number of program runs—the training set—and using that model on *new* program runs. But SPEC 2006 provides only three inputs ("sizes") for each program, not enough for a good training/testing protocol. (Some programs actually run on multiple inputs within one SPEC 2006 "size," however.) For principled training, we split single traces into *partitions* and run cross-validation across the partitions. We randomize the partitioning of the trace: each partition is a random sample of intervals rather than being contiguous in the trace.

We use "leave-one-out" cross-validation. If we have k available partitions, then each cross-validation fold trains on $k-1$ of the partitions and evaluates on the partition not trained

on. We do this leaving out each of the k partitions in turn, and average the results. We set $k = 10$ in the evaluation.

Table 6.1 shows the correlation between features selected from partitions created within the same trace, and the correlation between features selected from different traces of the same program. Unsurprisingly, the features selected for different partitions of the same trace (the In columns) are highly correlated (more than 0.95 correlation). This shows that the model learned from a partition has great potential for accurate fine-grained DVFS modeling of the whole trace. Concerning features selected for different traces of the same program, the correlations (the Out columns) are often high, but occasionally poor. This shows that our feature selection methodology generalizes moderately well across different inputs, but some programs may need training across a large range of inputs for good generalization.

Table 6.1: Correlation between selected features. "In" means the correlation between features selected from partitions of the same trace, and "Out" means the correlation between features selected from different traces of same program. The 6 and 8 after In/Out is the number of selected features.

Trace	In 6	Out 6	In 8	Out 8
`bzip2-ref-1`	0.93	0.99	0.95	0.99
`bzip2-train-1`	0.99	0.13	0.98	0.26
`gcc-ref-1`	0.99	0.98	0.99	0.99
`gcc-ref-2`	0.94	0.99	0.99	0.99
`gcc-ref-3`	0.99	0.98	0.99	0.99
`gcc-train-1`	0.86	0.36	0.93	0.39
`h264ref-ref-1`	0.95	NaN	0.99	NaN
Average	0.95	0.74	0.97	0.77

6.2.3.4 Number of Features and History

For real-time use, exploring the number of features and the history size for our DT-based clock-frequency-setting model is important. Insufficient features or history size may lead to a low quality adjuster, but redundant features and needlessly large history size can cause extra run-time overhead and be harmful to model generalization by overfitting.

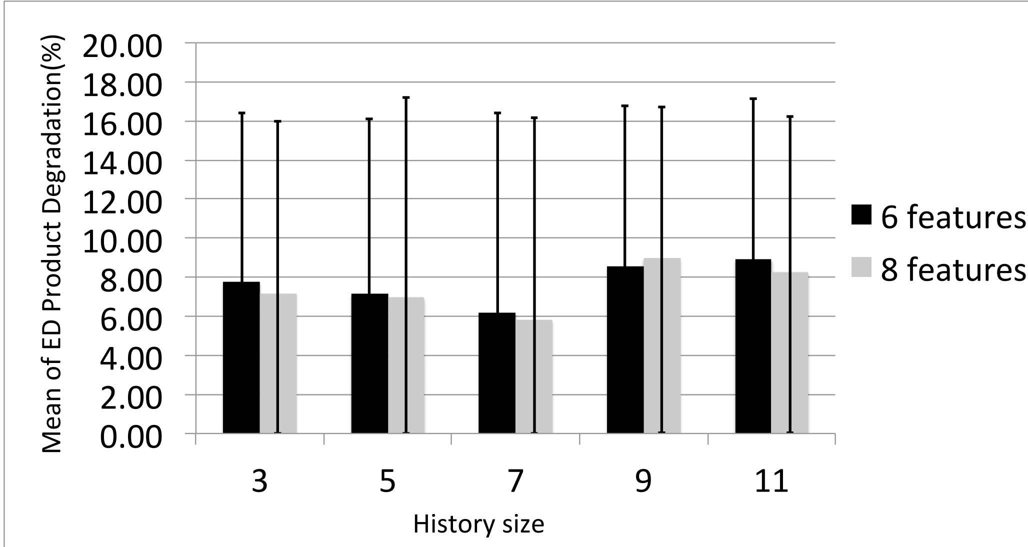


Figure 6.2: Cumulative ED degradation of the DT model over seven traces with different numbers of features and history lengths. The "whiskers" indicate the minimum and maximum ED degradation seen.

We ran experiments to select a suitable F and history size for our model. Figure 6.2 shows that DT leads to good ED with six or eight features and history length of seven. To build the real-time system, we use these two settings for our DT model implementation.

6.2.3.5 Comparison of the Models

For *LRCM* and *LR*, we ran the models for a variety of settings and chose the best one for each as the competitive model. The best setting for *LRCM* in the traces is running linear regression on 5 contiguous intervals' cache miss counts. For *LR*, the linear regression performs best with 10 selected features and history size 3.

Figure 6.3 shows how the various models compare when used to determine clock speed. We see that, even though DT uses fewer features than *LRCM*, it actually performs better on ED, a total degradation of 37% with 6 features and 34% with 8 features versus LR's 315%. This clearly demonstrates that this problem benefits from applying a non-linear model. Furthermore, comparing to other models that do not use any features, we see that the relative order of the schemes by quality is (1) DT with learned clock frequencies (37%

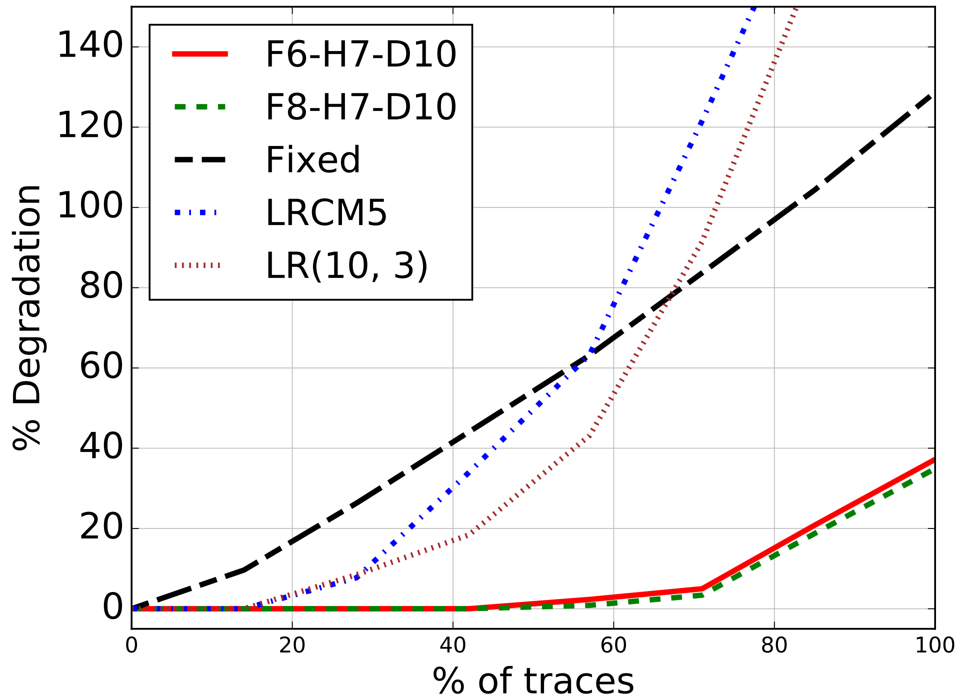


Figure 6.3: Cumulative ED degradation over seven traces of *LRCM*, *LR*, *DT* (with 6 and 8 features), and the model with the best fixed clock frequency.

and 34%), (2) the clock-frequency-setting model, which uses a learned single fixed clock frequency (128%), and (3) *LRCM* (386%). Not only does our model outperform other models for all of these traces, but also the clock-frequency-setting model outperforms *LRCM* and *LR*. Even though *LRCM* and *LR* offer high quality adjustment for 4 or 5 traces, they perform dramatically poorly for the remaining traces.

In short, DT not only gives the best results for an offline analysis unconcerned with real-time performance, but it also gives surprisingly good quality in the face of using such little information.

6.3 Implementing the Dynamic Clock Frequency Adjustment Tool

We implemented a tool to instrument C programs (compiled binaries, actually) for adjusting the clock frequency dynamically. To insert the instrumentation, we implemented

a PEBIL tool (Laurenzano et al., 2010), which does this by using function level code relocation. Each instrumented feature increments a counter for that feature. We also start a regular interval timer before entering the `main` function of the program, which causes periodic interrupts according to the execution time of the program (what the OS calls *virtual time*). Upon a timer interrupt, the program calls a function that obtains the counter values and then, using a trained decision tree, determines the CPU frequency for the next time interval. It then changes the frequency setting if the desired frequency is different from the current one.

6.4 Experiments

We now consider the run-time overhead of our real-time clock frequency adjustment, and its quality as an adjuster.

6.4.1 The Cost of Instrumented Programs

We compare the cost of running with and without the instrumentation that our systems adds. For each target benchmark and input, we have 10 model configurations. Each configuration includes a feature set and a trained decision tree model. We instrument the program with each configuration, then perform 5 runs of the instrumented program. We also perform 5 runs of each program and input with no instrumentation.

We insert instrumentation into a program's executable files in advance, using a PEBIL tool (Laurenzano et al., 2010) described in Section 6.3. For each target benchmark, input, and configuration, we generate two instrumented programs to evaluate how different aspects of our model code effect the run-time overhead. The first instrumented program executes feature counting without calling the DT evaluation function. We call these programs F-only. The second instrumented program includes calling the DT evaluation function, which we call Eval.

Table 6.2: Run-time overhead of instrumented programs with 6 and 8 features compared to the uninstrumented versions.

Trace	F-only 6	Eval 6	F-only 8	Eval 8
bzip2-ref-1	0.1%	1.9%	3.2%	3.1%
bzip2-train-1	2.0%	0.3%	3.3%	0.5%
gcc-ref-1	0.0%	1.0%	0.4%	1.8%
gcc-ref-2	0.4%	1.2%	0.0%	0.8%
gcc-ref-3	4.7%	1.3%	0.6%	0.8%
gcc-train-1	2.7%	2.1%	2.5%	1.4%
h264ref-ref-1	0.3%	0.7%	0.5%	0.1%
Average	1.4%	1.2%	1.4%	1.2%

We find that the average run-time overhead actually does not increase as we instrument more features. (This might be expected since we limit the overall total count.) The smallest number of features that gives good detection quality is 6, resulting in an average overhead of 1.2% (see Table 6.2). Using 8 features gives the same 1.2%. It is curious that Eval sometimes costs less than F-only, etc., but the worst Eval overhead is 3.1%, which we consider an excellent result.

To test the statistical significance of the difference in running times, we performed T-tests comparing the instrumented and uninstrumented time distributions (see Table 6.3). All cases show quite strong significance ($p = 0.07$ means that there is a 7% chance that the two distributions were drawn from the same underlying distribution). We conclude that there is a real performance difference, though it is small.

6.4.2 Performance of the Real-time Decision Tree Model

When facing the same problem as we described in Section 4.2.3, we cannot easily trigger our adjustment function at the same points during execution that the offline function uses. Our solution for it is choosing a timer interrupt interval that should break the trace into about the same number of intervals as were found by the tracing tool.

Table 6.4 shows that the ratio of ED product of the real-time adjustment system to the best single fixed clock frequency model is similar to the performance estimated by our

Table 6.3: T-test of execution time of programs run with and without instrumentation.

Trace	F-only 6	Eval 6	F-only 8	Eval 8
`bzip2-ref-1`	0.34	0.00	0.00	0.00
`bzip2-train-1`	0.00	0.12	0.00	0.00
`gcc-ref-1`	0.00	0.00	0.06	0.00
`gcc-ref-2`	0.16	0.00	0.00	0.00
`gcc-ref-3`	0.00	0.00	0.07	0.00
`gcc-train-1`	0.01	0.17	0.00	0.21
`h264ref-ref-1`	0.00	0.00	0.00	0.02
Average	0.07	0.04	0.02	0.03

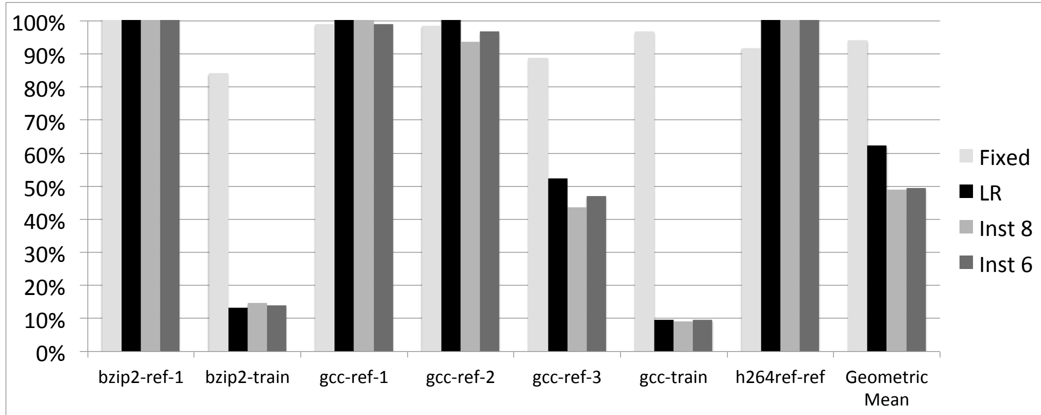
offline DT for 6 features, and slightly worse when using 8 features. We conclude that the trace-window-based timer is an appropriate adjustment function trigger.

Table 6.4: Average ratio of ED product of the real-time adjustment system to the best single fixed clock frequency model.

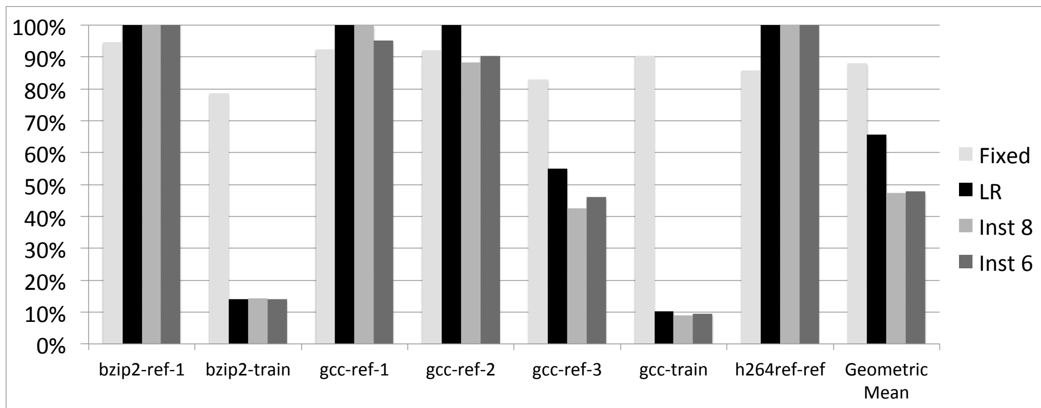
Features	Offline	Real-time
6	75.7%	75.7%
8	75.4%	76.2%

The most interesting comparison is with the ED performance of the other models, but given the different running times and various clock frequencies, how shall we compare? First, we record the system clock frequency each time the timer interrupt is triggered, as well as the number of clock cycles since the last interrupt. This allows us to compute ED using Equation (6.6) of Section 6.2.1. We obtain the ED products shown in Figure 6.4, Figure 6.5, and Figure 6.6, bracketing the value of V_{th}, which is nominally 0.4, with values 0.2 and 0.6, and the value of k, nominally 0.8, with values 0.6 and 1.0.

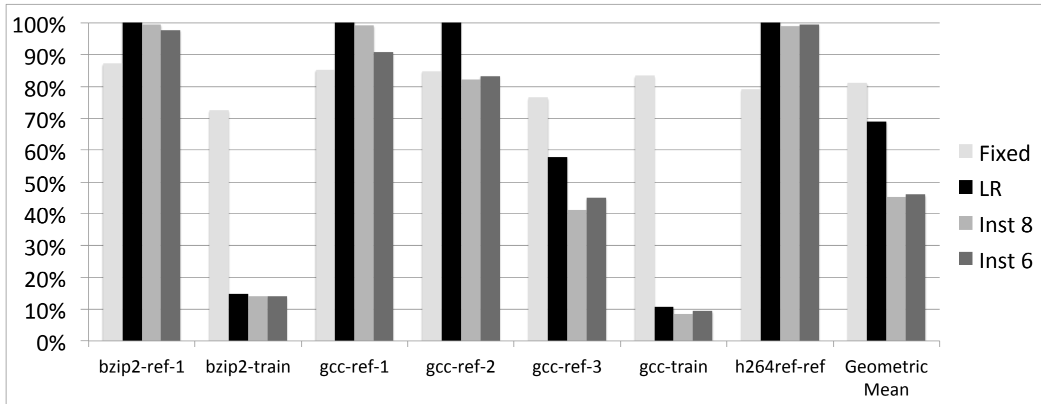
We compare our real-time adjuster's performance to the program running on a fixed single clock frequency learned in the first stage of our model and Choi et al. (2005)'s linear regression model (*LR*). We find that Inst 8 gives good results, better than Inst 6, when comparing with the performance of Default, and better than Fixed. Inst 8 achieves a geometric mean ED improvement of about 54.2% (from 51% to 59%), compared to 53.6%



(a) $V_{th} = 0.2$.

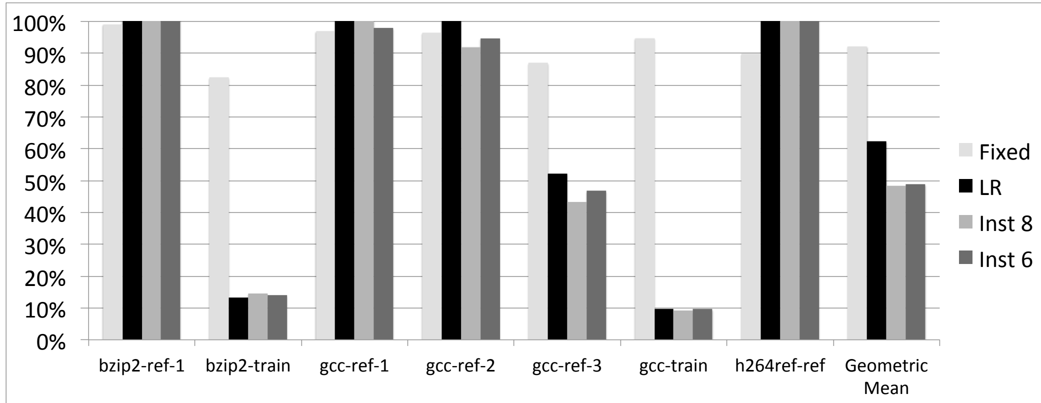


(b) $V_{th} = 0.4$.

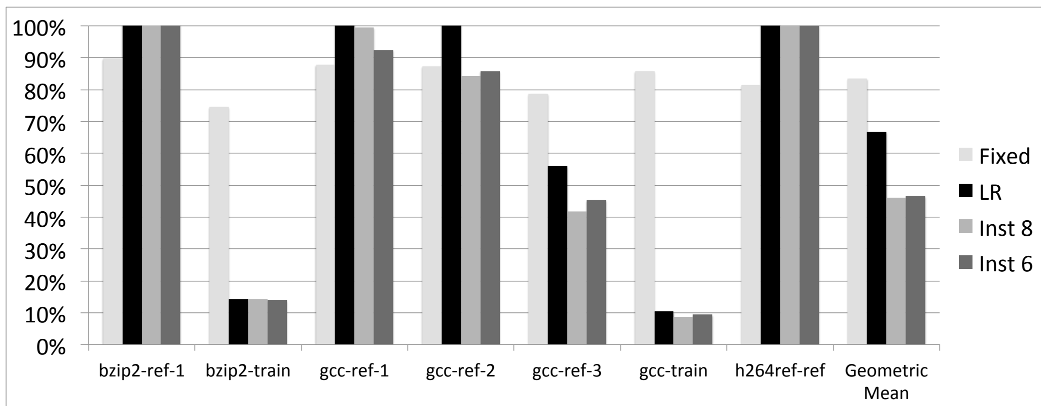


(c) $V_{th} = 0.6$.

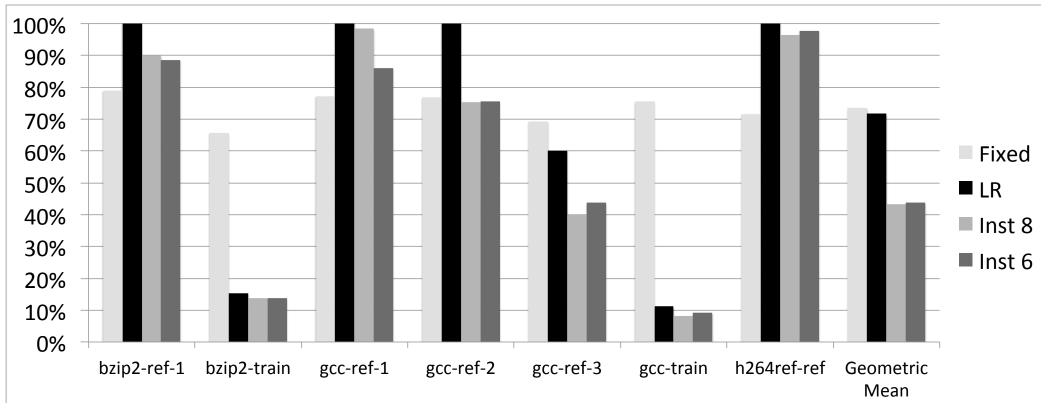
Figure 6.4: Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.6$.



(a) $V_{th} = 0.2$.



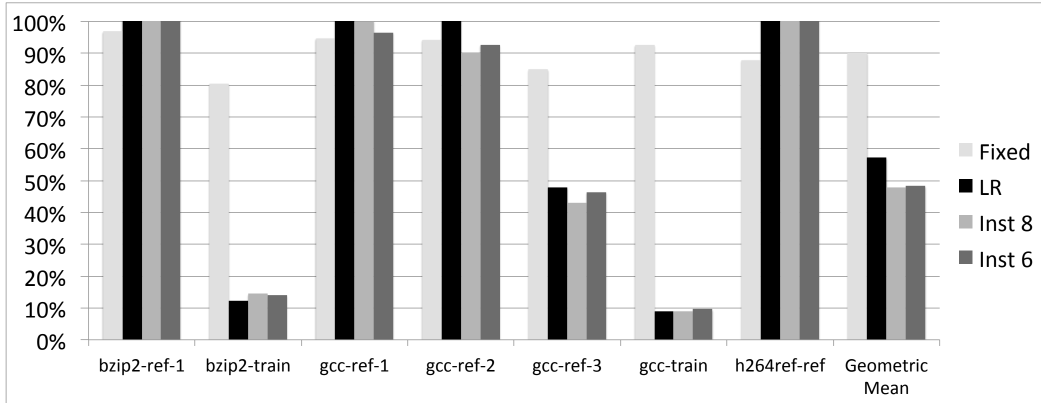
(b) $V_{th} = 0.4$.



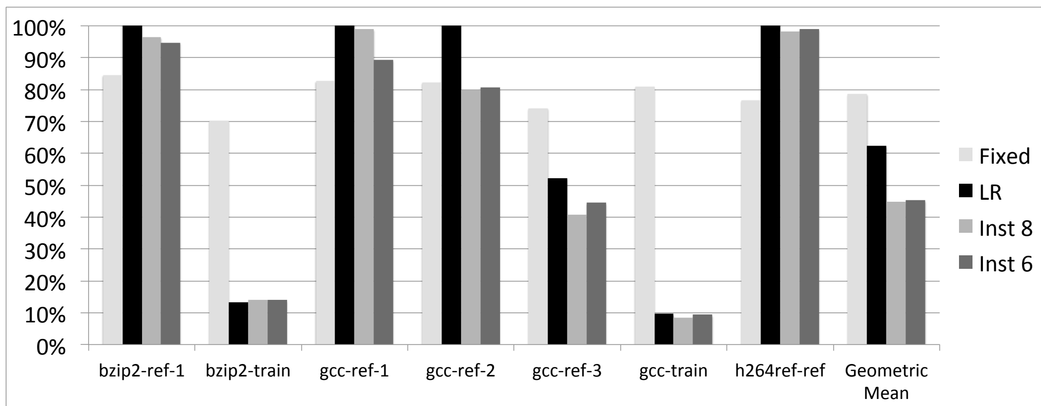
(c) $V_{th} = 0.6$.

Figure 6.5: Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.8$.

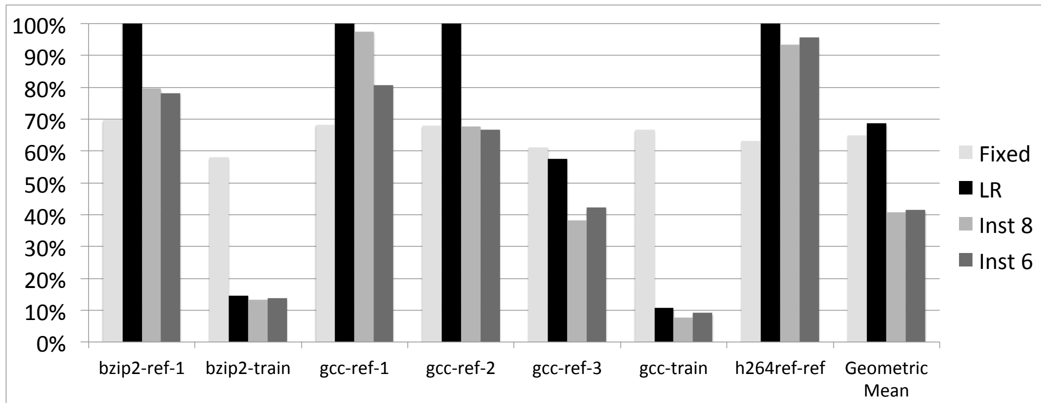
(from 50% to 58%) for Inst 6, 34.9% (from 31% to 43%) for *LR*, and 17.1% (from 5% to 26%) for Fixed, while there is only 1% improvement from instrumenting two more features.



(a) $k = 1.0, V_{th} = 0.2$.



(b) $k = 1.0, V_{th} = 0.4$.



(c) $k = 1.0, V_{th} = 0.6$.

Figure 6.6: Clock frequency adjustment quality of instrumented programs with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 1.0$.

Furthermore, Inst 8 achieves 20.7% above the optimal ED product, compared to 23.7% for

Table 6.5: Average of geometric mean of (ED - optimal)/optimal for nine k and V_{th} settings.

Setting		Fixed	LR	Inst 6	Inst 8
k	V_{th}				
0.6	0.2	264.6%	175.2%	30.5%	30.0%
0.6	0.4	247.1%	184.2%	27.1%	25.3%
0.6	0.6	227.8%	194.0%	23.2%	20.0%
0.8	0.2	258.9%	175.3%	29.4%	28.5%
0.8	0.4	234.5%	187.6%	24.5%	21.9%
0.8	0.6	206.3%	201.7%	18.4%	13.4%
1.0	0.2	253.1%	160.7%	28.3%	27.0%
1.0	0.4	220.9%	175.3%	21.7%	18.0%
1.0	0.6	182.3%	192.8%	10.4%	2.6%
Average		232.8%	182.9%	23.7%	20.7%

Inst 6, 182.9% for *LR*, and 232.8% for Fixed. ("Optimal" is with respect to the chosen clock adjustment interval and the available clock frequencies, based on our model of ED product.)

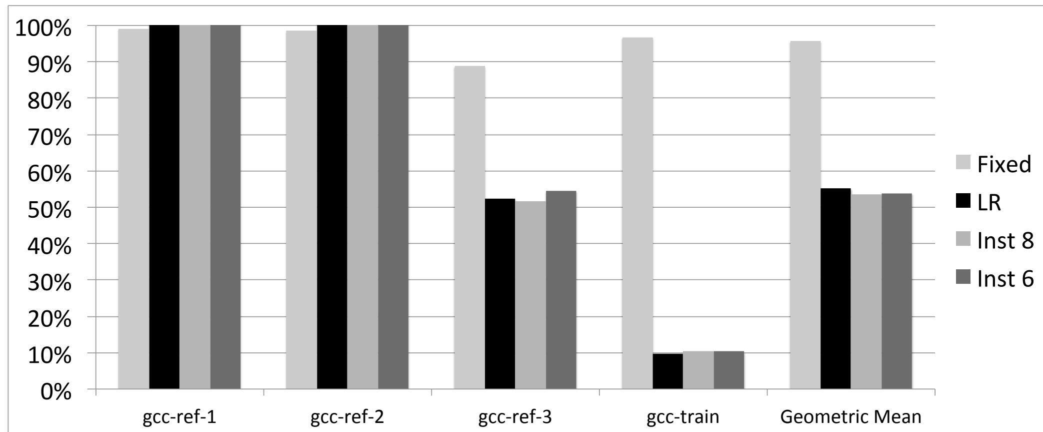
In sum, our real-time clock frequency adjustment system can reduce a program's energy-delay product to within 20-25% of optimal, and does so with small impact on execution time.

6.4.2.1 Generalizability of the Real-time DT Model

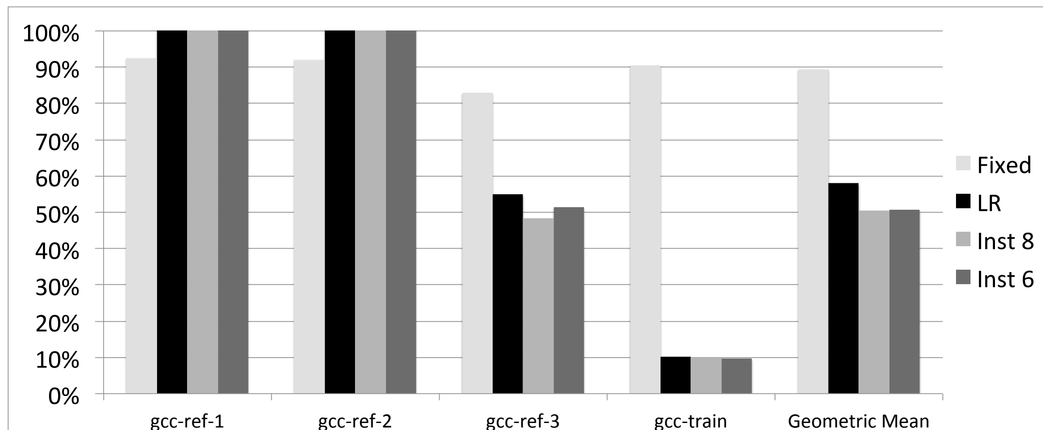
The results show that the ED product improvements that our real-time clock frequency adjustment method achieves are better than those of the other models for each trace. Another interesting evaluation is our model's generalizability, but we cannot evaluate generalizability well for programs with only one or two traces. We chose the four `gcc` traces as a target to evaluate generalization. For each `gcc` input, we instrumented the program based on the selected feature sets and decision trees learned from the other three traces, then ran the instrumented program with the target input. We obtained the ED products shown in Figures 6.7, 6.8, and 6.9.

The results show that Inst 8 and Inst 6 give good results compared with the performance of Default, and better than Fixed. Inst 8 achieves a geometric mean ED product improvement of about 51.9% (from 46% to 60%) and 52.0% (from 46% to 61%) for Inst 6, compared to

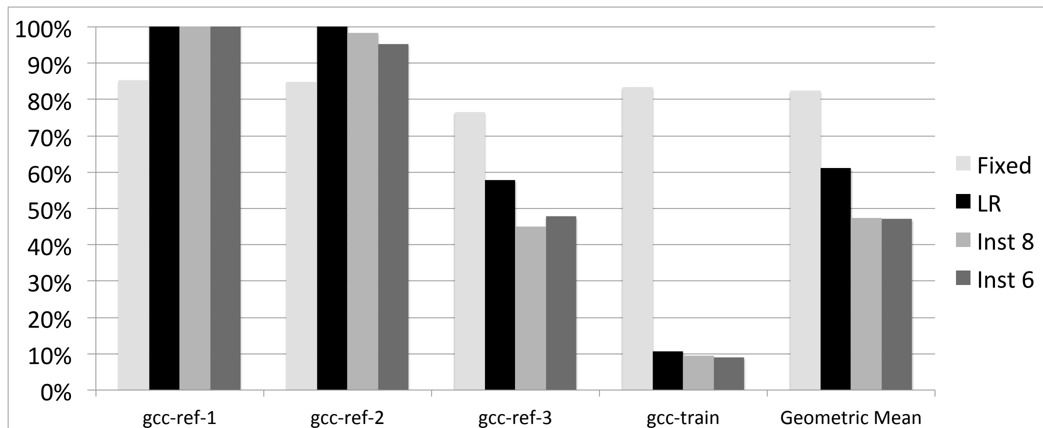
42.4% (from 36% to 49%) for *LR* and 15.8% (from 4% to 34%) for Fixed. Furthermore, the performance of these generalized models (with average 52% improvement for Inst 6 and Inst 8) is close to the models learned in each trace (with average 57 and 58% improvement for Inst 6 and Inst 8). The selected feature sets have high correlation (shown in Table 6.1). Thus, given enough traces to cover the expected behaviors of the program, DTs have high potential to generalize over different inputs of the same program.



(a) $V_{th} = 0.2$.

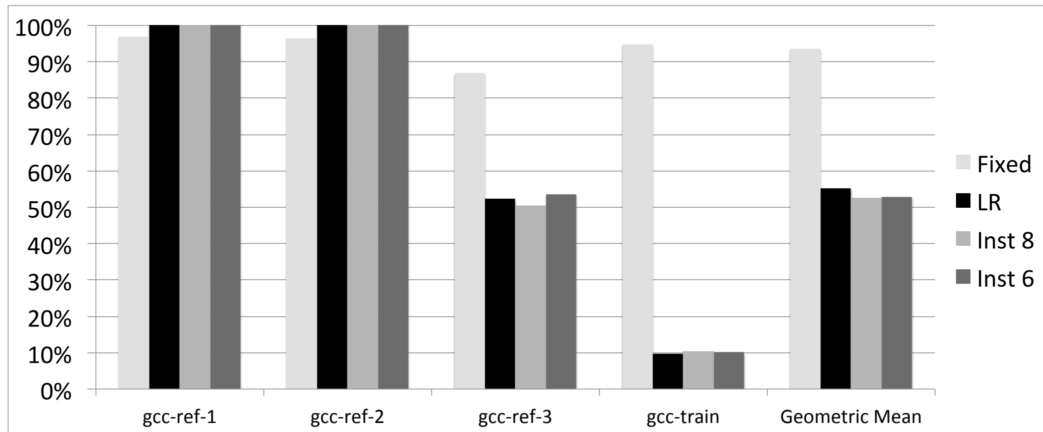


(b) $V_{th} = 0.4$.

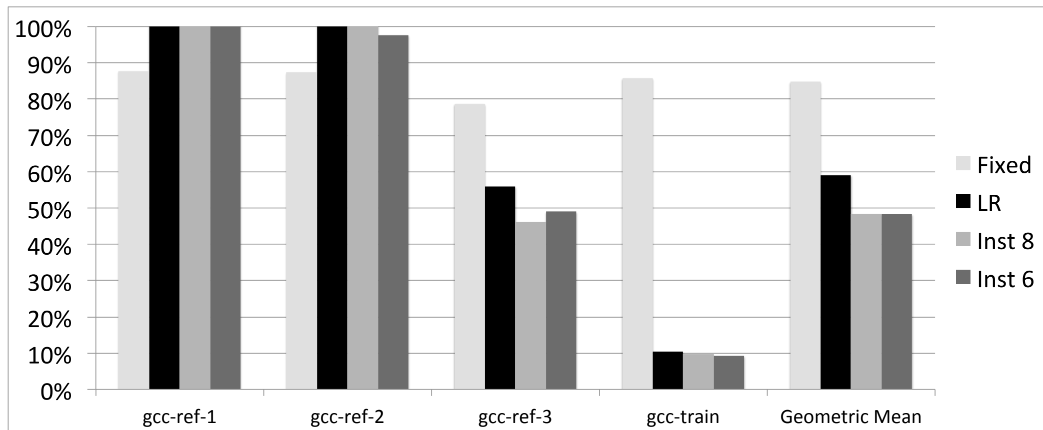


(c) $V_{th} = 0.6$.

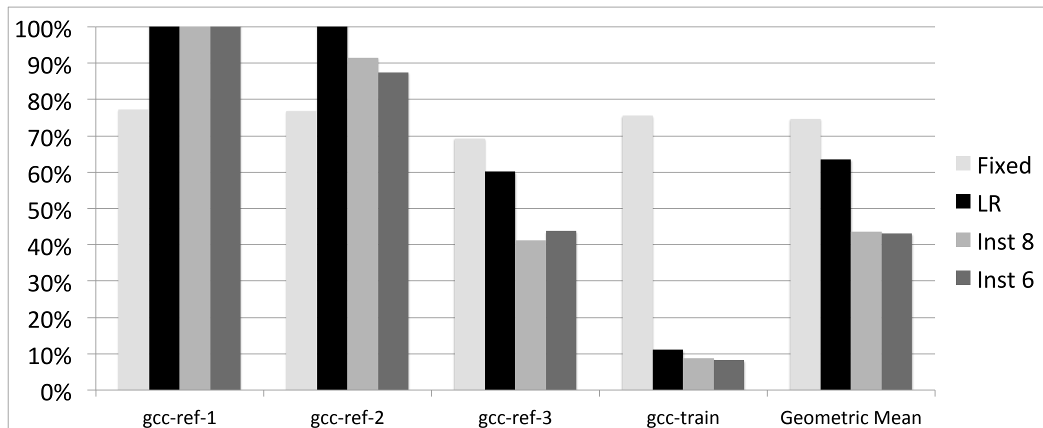
Figure 6.7: Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.6$.



(a) $V_{th} = 0.2$.

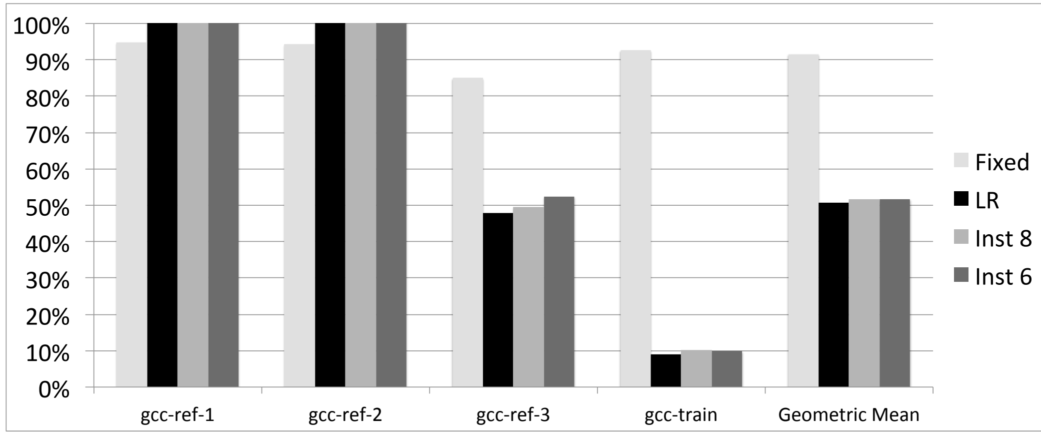


(b) $V_{th} = 0.4$.

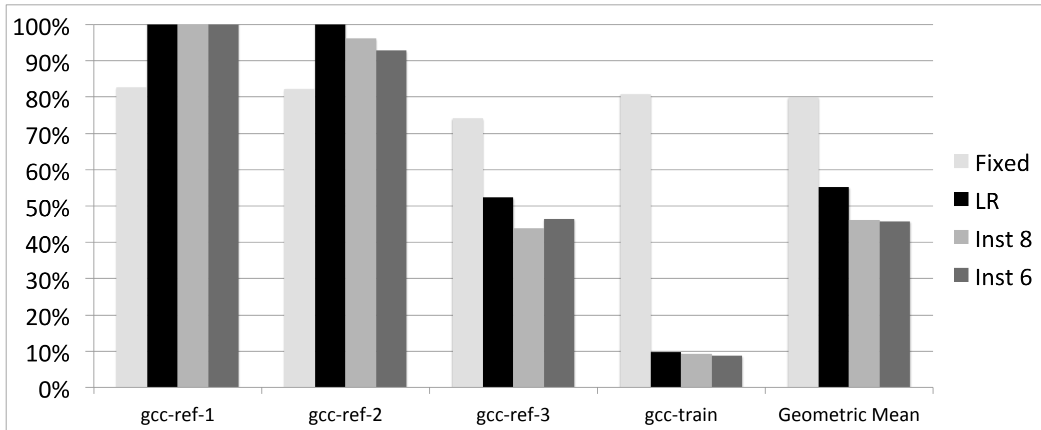


(c) $V_{th} = 0.6$.

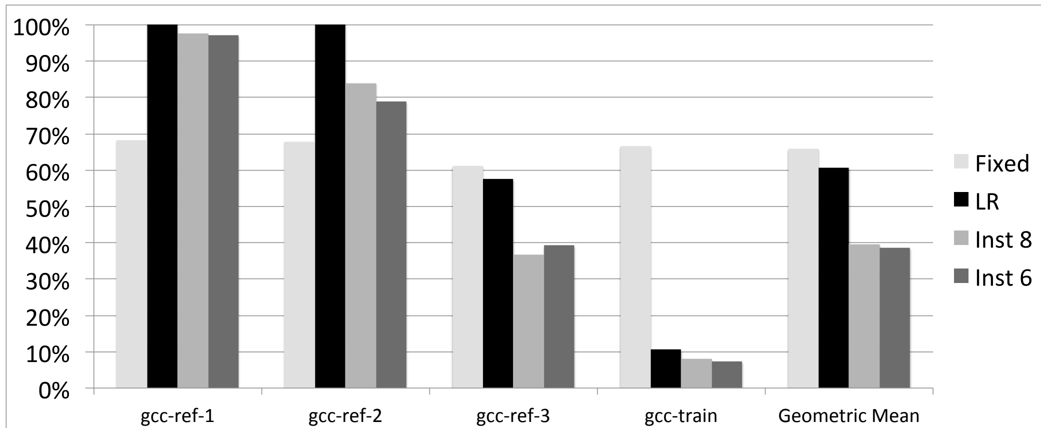
Figure 6.8: Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 0.8$.



(a) $V_{th} = 0.2$.



(b) $V_{th} = 0.4$.



(c) $V_{th} = 0.6$.

Figure 6.9: Clock frequency adjustment quality of instrumented programs based on models learned from other traces with 6 and 8 features, fixed clock frequency, and linear regression, for $k = 1.0$.

CHAPTER 7

CONCLUSION

We have presented a method for run-time phase transition detection and phase prediction in Java, C (etc.), and Python programs. After applying a training protocol to a program of interest, our methods can detect phase changes and predict the future phase k steps later, for $k = 1, ...$, at run time for that program with good precision and recall and without significant run-time overhead.

We trace a number of executions of the program of interest, recording each branch that occurs, accumulating them into feature vectors. We compute "ground truth" by clustering the feature vectors within phase boundaries derived directly from traces according to a precise definition of phases (that of Nagpurkar et al. (2006)). We use the traces and this "ground truth" to develop an efficient offline run-time phase predictor.

We choose a small number of branch instructions that we will instrument at run time and apply machine learning models. For phase transition detection, we use Gaussian mixture models (GMMs) to cluster vectors of branch execution counts, while we apply decision trees built over this small number of features to do the actual phase prediction. We use DaCapo benchmarks, SPEC CPU2006 benchmarks, and Pyperformance benchmarks to test our systems. Our experiments show that, for Java, C, and Python programs, our offline GMM detector can achieve results similar to, and even better than, the detector of Nagpurkar et al. (2006) and a detector that reports transitions randomly with probability equal to the average rate of transitions as determined by a reference definition of phases and transitions. In addition, our offline decision tree model can achieve better results than three other predictors: Uniform, Random, and ProbSim in future phase prediction.

To make run-time measurements, we implemented the phase transition detector and phase predictor for different programming languages. For Java, we build systems as Java agents, using the Java Virtual Machine Tool Interface (JVMTI). For C, we build PEBIL tools. For Python, we build the systems inside CPython. We count the selected branches, and one (extra) feature that has lowest variance across the traces as a timing proxy. When the total count of the timing feature reaches a threshold, we trigger a function that implements the detector's GMM or predictor's decision tree.

Our experimental results show that our methods do not significantly impact performance of the running program, and the detector and predictor correspond well with the underlying definition of phases ("ground truth"). All our GMM-based phase change detectors and decision-tree-based phase predictors for three different programming languages have less than 3% run-time overhead. In addition, the detectors achieve 5% to 10% improvement in SWF score compared to offline ND, and the predictors achieve 79% to 85% F-score for predicting next phase.

Beyond to phase transition detection and phase prediction, we explore run-time energy-efficient clock frequency adjustment for statically compiled executable programs. Our method can adjust clock frequency at run time for that program to significantly lower the energy-delay (ED) product with small performance impact. For the adjustment interval selected (corresponding to about 100,000 branches per interval), our method comes within 20-25% of the optimal ED product, according to our model of ED. Other schemes are on the order of 200% of optimal.

We use the traces to develop an efficient two stage learning model: clock-frequency-setting learning and dynamic-adjustment learning. In clock-frequency-setting learning, we apply brute force analysis to find the best frequency settings for each trace. In dynamic-adjustment learning, we choose a small number of branch instructions that we will instrument at run time, and then induce decision tree models to determine ED product efficient clock settings from vectors of branch execution counts. Our experiments show that our offline

adjuster can achieve results better than linear regression model, even when applying only clock-frequency-setting learning. We find that only a minority of traces benefit from *dynamic* clock frequency adjustment, but that we can achieve useful improvement for them with low overhead.

To make run-time adjustments, we implemented the clock frequency adjuster as a PEBIL tool. We count the instrumented branches, and when a timer is triggered, we evaluate the decision tree on the counts for the recent past. If the setting indicated by the decision tree is different from the current setting, we adjust the clock frequency.

In the future, we will explore and apply other machine learning techniques for phase detection and prediction. For phase transition detection, we aim to enhance our detector for the programs, i.e., `bwaves`(C) and `genshi`(Python), in which our GMMs do not work well. For phase prediction, we will pursue the possibility to further improve our predictor, making it perform similar to, even better than, CondUniform in Java and C programs. In addition, for clock frequency adjustment, in order to verify the correctness of the fomula described in Section 6.2.1, we will implement processor monitoring and cycle-accurate simulation. Furthermore, for cross-validation, in the SPEC CPU2006 benchmarks, if we cannot get a large number of runs that can be futher improved in energy-delay product by our model compared to choosing one fixed clock frequency, we will find more traces where adjustment is useful in future work. One reasonable possibility is to move ahead to SPEC CPU2017, which has additional workloads from the University of Alberta.

BIBLIOGRAPHY

Azevedo, Ana, Issenin, Ilya, Cornea, Radu, Gupta, Rajesh, Dutt, Nikil, Veidenbaum, Alex, and Nicolau, Alexandru. Profile-based dynamic voltage scheduling using program checkpoints. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 168–175. IEEE, 2002.

Benomar, Omar, Sahraoui, Houari, and Poulin, Pierre. Detecting program execution phases using heuristic search. In *Search-Based Software Engineering*, pages 16–30. Springer, 2014.

Bhattacharjee, Abhishek and Martonosi, Margaret. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ACM SIGARCH Computer Architecture News*, pages 290–301. ACM, 2009.

Blackburn, S. M., Garner, R., Hoffman, C., Khan, A., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Guyer, S. Z., Hosking, A., Jump, M., Moss, J. E. B., Stefanovic, D., VanDrunen, T., von Dincklage, D., and Widerman, B. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 2006 ACM International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2006)*, pages 169–190, Portland, OR, October 2006.

Bui, Van and Kim, Martha Allen. Analysis of super fine-grained program phases. Technical report, Columbia University, 2017.

Chiu, Meng-Chieh, Marlin, Benjamin, and Moss, Eliot. Real-time program-specific phase change detection for Java programs. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2016.

Cho, Chang-Burm and Li, Tao. Complexity-based program phase analysis and classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2006*, pages 105–113. ACM, 2006.

Choi, Kihwan, Soma, Ramakrishna, and Pedram, Massoud. Fine-grained dynamic voltage and frequency scaling for precise energy and performance tradeoff based on the ratio of off-chip access to on-chip computation times. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 24(1):18–28, 2005.

Dhiman, Gaurav and Rosing, Tajana Simunic. Dynamic voltage frequency scaling for multi-tasking systems using online learning. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 207–212. IEEE, 2007.

Dhodapkar, Ashutosh S. and Smith, James E. Comparing program phase detection techniques. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, page 217. IEEE Computer Society, 2003.

Dieckmann, Sylvia and Hölzle, Urs. A study of the allocation behavior of the SPECjvm98 Java benchmark. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 92–115, London, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5. URL `http://dl.acm.org/citation.cfm?id=646156.679838`.

Dropsho, Steven G and Mckinley, Kathryn S. Enhancing branch prediction via on-line statistical analysis. 2002.

Duesterwald, Evelyn, Caşcaval, Călin, and Dwarkadas, Sandhya. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003*, pages 220–231. IEEE, 2003.

Gaynor, Alex, Vassalotti, Alexandre, Pitrou, Antoine, Gupta, Anuj, Peterson, Benjamin, Impollonia, Bobby, Cannon, Brett, Winter, Collin, Laing, David, Malcolm, David, Jemerov, Dmitry, Papa, Florin, Brandl, Georg, Abbatiello, James, Yasskin, Jeffrey, Fijalkowski, Maciej, Klecker, Reid, Montanaro, Skip, Behnel, Stefan, Wouters, Thomas, Stinner, Victor, and Ware, Zachary. Pyperformance: Python performance project. https://github.com/python/performance. Accessed: 2016-11-29.

Georges, Andy, Buytaert, Dries, Eeckhout, Lieven, and De Bosschere, Koen. Method-level phase behavior in Java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 270–287, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: 10.1145/1028976.1028999. URL `http://doi.acm.org/10.1145/1028976.1028999`.

Grunwald, Dirk, Morrey III, Charles B., Levis, Philip, Neufeld, Michael, and Farkas, Keith I. Policies for dynamic clock scheduling. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 6. USENIX Association, 2000.

Gu, Dayong and Verbrugge, Clark. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 24–34. ACM, 2008.

Hastie, Trevor, Tibshirani, Robert, and Friedman, Jerome. Unsupervised learning. In *The Elements of Statistical Learning*, pages 485–585. Springer, 2009.

Henning, John L. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

Herbert, Sebastian and Marculescu, Diana. Variation-aware dynamic voltage/frequency scaling. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 301–312. IEEE, 2009.

Horowitz, Mark, Indermaur, Thomas, and Gonzalez, Ricardo. Low-power digital design. In *Low Power Electronics, 1994. Digest of Technical Papers., IEEE Symposium*, pages 8–11. IEEE, 1994.

Hotta, Yoshihiko, Sato, Mitsuhisa, Kimura, Hideaki, Matsuoka, Satoshi, Boku, Taisuke, and Takahashi, Daisuke. Profile-based optimization of power performance by using dynamic voltage scaling on a PC cluster. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006.)*. IEEE, 2006.

Lau, Jeremy, Perelman, Erez, Hamerly, Greg, Sherwood, Timothy, and Calder, Brad. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2005*, pages 135–146. IEEE, 2005.

Laurenzano, Michael A., Tikir, Mustafa M., Carrington, Laura, and Snavely, Allan. PEBIL: Efficient static binary instrumentation for Linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010.

Li, Hai, Cher, Chen-Yong, Roy, Kaushik, and Vijaykumar, T. N. Combined circuit and architectural level variable supply-voltage scaling for low power. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(5):564–576, 2005.

Magklis, Grigorios, Chaparro, Pedro, González, José, and González, Antonio. Independent front-end and back-end dynamic voltage scaling for a GALS microarchitecture. In *Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 49–54. ACM, 2006.

McLachlan, Geoffrey and Peel, David. *Finite mixture models*. John Wiley & Sons, 2004.

Nagpurkar, Priya and Krintz, Chandra. Visualization and analysis of phased behavior in Java programs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, pages 27–33. Trinity College Dublin, 2004.

Nagpurkar, Priya, Krintz, Chandra, Hind, Michael, Sweeney, Peter F., and Rajan, V. T. Online phase detection algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 111–123. IEEE Computer Society, 2006.

Nanz, Sebastian and Furia, Carlo A. A comparative study of programming languages in Rosetta Code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, volume 1, pages 778–788. IEEE, 2015.

Nethercote, Nicholas and Seward, Julian. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 42, pages 89–100. ACM, 2007.

Otte, Michael and Richardson, Scott. An HMM applied to semi-online program phase analysis (CU-CS-1034-07). Technical report, Univ. of Colorado Boulder, 2007.

Peleg, Nitzan and Mendelson, Bilha. Detecting change in program behavior for adaptive optimization. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 150–162. IEEE, 2007.

Pirzadeh, Heidar, Agarwal, Akanksha, and Hamou-Lhadj, Abdelwahab. An approach for detecting execution phases of a system for the purpose of program comprehension. In *Eighth ACIS International Conference on Software Engineering Research, Management, and Applications (SERA), 2010*, pages 207–214. IEEE, 2010.

Pirzadeh, Heidar, Hamou-Lhadj, Abdelwahab, and Shah, Mohak. Exploiting text mining techniques in the analysis of execution traces. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 223–232. IEEE, 2011.

Rangan, Krishna K., Wei, Gu-Yeon, and Brooks, David. Thread motion: fine-grained power management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 37 (3):302–313, 2009.

Ricci, Nathan P., Guyer, Samuel Z., and Moss, J. Eliot B. Tool demonstration: Elephant Tracks—generating program traces with object death records. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java*, pages 39–43, Kongens Lyngby, Denmark, Aug. 2011. ACM.

Ricci, Nathan P., Guyer, Samuel Z., and Moss, J. Eliot B. Elephant Tracks: Portable production of complete and precise GC traces. In Cheng, Perry and Petrank, Erez, editors, *International Symposium on Memory Management, ISMM '13, Seattle, WA, USA - June 20 - 20, 2013*, pages 109–118. ACM, 2013. ISBN 978-1-4503-2100-6. doi: 10.1145/2464157.2466484. URL `http://doi.acm.org/10.1145/2464157.2466484`.

Roh, Yangwoo, Kim, Jaesub, and Park, Kyu Ho. A phase-adaptive garbage collector using dynamic heap partitioning and opportunistic collection. *IEICE TRANSACTIONS on Information and Systems*, 92(10):2053–2063, 2009.

Sakurai, Takayasu and Newton, A. Richard. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, Apr. 1990.

Shen, Xipeng, Zhong, Yutao, and Ding, Chen. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 165–176, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024414. URL `http://doi.acm.org/10.1145/1024393.1024414`.

Sherwood, Timothy, Perelman, Erez, Hamerly, Greg, and Calder, Brad. Automatically characterizing large scale program behavior. *ACM SIGOPS Operating Systems Review*, 36(5):45–57, 2002.

Shin, Dongkun and Kim, Jihong. A profile-based energy-efficient intra-task voltage scheduling algorithm for hard real-time applications. In *Low Power Electronics and Design, 2001.*, pages 271–274. IEEE, 2001.

Singer, Jeremy and Kirkham, Chris. Dynamic analysis of Java program concepts for visualization and profiling. *Science of Computer Programming*, 70(2):111–126, 2008.

Srinivasan, Sudarshan, Kumar, Raghavan, and Kundu, Sandip. Program phase duration prediction and its application to fine-grain power management. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 127–132. IEEE, 2013.

Tajik, Hossein, Donyanavard, Bryan, and Dutt, Nikil. On detecting and using memory phases in multimedia systems. In *Proceedings of the 14th ACM/IEEE Symposium on Embedded Systems for Real-Time Multimedia*, pages 57–66. ACM, 2016.

Wang, Chao, Li, Xi, Dai, Dong, Jia, Gangyong, and Zhou, Xuehai. Phase detection for loop-based programs on multicore architectures. In *2012 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 584–587. IEEE, 2012.

Watanabe, Yui, Ishio, Takashi, and Inoue, Katsuro. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 International Workshop on Dynamic Analysis (WODA'08)*, pages 8–14. ACM, 2008.

Wimmer, Christian, Cintra, Marcelo S., Bebenita, Michael, Chang, Mason, Gal, Andreas, and Franz, Michael. Phase detection using trace compilation. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 172–181. ACM, 2009.

Wu, Qiang, Juang, Philo, Martonosi, Margaret, and Clark, Douglas W. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *ACM SIGARCH Computer Architecture News*, 32(5):248–259, 2004.

Wu, Qiang, Juang, Philo, Martonosi, Margaret, and Clark, Douglas W. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *11th International Symposium on High-Performance Computer Architecture (HPCA 2005)*, pages 178–189. IEEE, 2005a.

Wu, Qiang, Martonosi, Margaret, Clark, Douglas W., Janapa, Vijay Reddi, Connors, Dan, Wu, Youfeng, Lee, Jin, and Brooks, David. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282. IEEE Computer Society, 2005b.

Xian, Feng, Srisa-an, Witawas, and Jiang, Hong. Microphase: An approach to proactively invoking garbage collection for improved performance. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '07, pages 77–96, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297034. URL `http://doi.acm.org/10.1145/1297027.1297034`.

Xie, Fen, Martonosi, Margaret, and Malik, Sharad. Compile-time dynamic voltage scaling settings: Opportunities and limits. *In Proceedings of the 24th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 38(5):49–62, 2003.

Zhang, Weihua, Li, Jiaxin, Li, Yi, and Chen, Haibo. Multilevel phase analysis. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(2):31, 2015.